Documentation of `stattools`

Madleina Caduff

September 16, 2024

Contents

The C++ library `stattools` is a collection of tools for statistical inference. It implements the Markov Chain Monte Carlo (MCMC) algorithm, the Expectation Maximization (EM) algorithm, the Baum-Welch algorithm for Hidden Markov Models (HMM), as well as several optimization methods such as Newton-Raphson, Nelder-Mead and line search. In the following sections, we will provide a detailed description of the implementation of these algorithms in `stattools`. The algorithms themselves are only described in the light of what is necessary for an understanding of `stattools`. For further details, derivations and explanations, we refer to the excellent chapters in Wegmann and Leuenberger (2019), Barber (2012), and Murphy (2012).

1 Markov Chain Monte Carlo

The Markov Chain Monte Carlo (MCMC) method is a widely used algorithm to approximate a distribution by generating samples from it (Barber, 2012; Murphy, 2012). In Bayesian inference, it is commonly used to approximate the posterior distribution $\mathbb{P}(\boldsymbol{\theta}|\boldsymbol{x})$ of a parameter $\boldsymbol{\theta}$. The idea in MCMC is to set up an irreducible, aperiodic Markov chain where the stationary distribution is equal to the distribution of interest.
Various algorithm exist that construct such a chain, most importantly the Metropolis-Hastings algorithm (Metropolis et al., 1953; Hastings, 1970):

1. Choose initial values $\boldsymbol{\theta}_0$ and set $i = 0$.
2. Propose a move $\boldsymbol{\theta}_i \rightarrow \boldsymbol{\theta}'_i$ according to some proposal kernel $q(\boldsymbol{\theta}, \boldsymbol{\theta}')$.
3. Accept move and set $\boldsymbol{\theta}_{i+1} = \boldsymbol{\theta}'_i$ with probability given by the Hastings ratio
$$h = \min\left(1, \frac{\mathbb{P}(\boldsymbol{\theta}')q(\boldsymbol{\theta}'_t, \boldsymbol{\theta}_t)}{\mathbb{P}(\boldsymbol{\theta})q(\boldsymbol{\theta}_t, \boldsymbol{\theta}'_t)}\right),$$
else reject move and set $\boldsymbol{\theta}_{i+1} = \boldsymbol{\theta}_i$.
4. Increment i and go back to step 2.

A special case of the Metropolis-Hastings algorithm is Gibbs sampling. Gibbs sampling circumvents the problem of sampling from a complicated joint distribution $\mathbb{P}(\boldsymbol{\theta})$ by sampling from the conditional distribution $\mathbb{P}(x|\boldsymbol{\theta}_{-i})$, where $\boldsymbol{\theta}_{-i}$ denotes all parameters except the i^{th} element θ_i. For discrete parameters, Gibbs sampling is straightforward to implement. Denoting by $\mathbb{P}(\mathcal{D}|\theta_i, \boldsymbol{\theta}_{-i})$ the likelihood and by $\mathbb{P}(\theta_i|\pi)$ the prior probability, let ϕ_{θ_i} be the product of likelihood and prior:
$$\phi_{\theta_i} = \mathbb{P}(\mathcal{D}|\theta_i, \boldsymbol{\theta}_{-i})\mathbb{P}(\theta_i|\pi).$$
The posterior distribution can be calculated analytically, since the integral for normalization turns into a sum in case of discrete parameters:
$$\mathbb{P}(\theta_i|\mathcal{D}, \pi, \boldsymbol{\theta}_{-i}) = \frac{\phi_{\theta_i}}{\sum_\gamma \phi_\gamma}. \tag{1}$$
where the sum in the denominator integrates over all possible values of θ_i. In addition, Gibbs sampling is commonly used when the prior distribution is conjugate with respect to the likelihood, such that the posterior distribution can be calculated analytically (Murphy, 2012).
In a typical setup, multiple parameters are inferred in an MCMC, and Metropolis-Hastings or Gibbs sampling algorithm can be mixed depending on the particular choice of prior distributions. The handling of these parameters - be it the choice of a proposal kernel, prior distribution, updating scheme, initialization, file handling etc. - can be cumbersome to implement. `stattools` provides a powerful and flexible implementation of the MCMC algorithm that makes it easy and efficient to program MCMCs without needing to start from scratch every time.

1.1 Statistical model

A statistical model defines the vector of parameters $\boldsymbol{\theta}$ that are estimated from a vector of observations, $\mathcal{D}$ and the joint probability distribution $\mathbb{P}(\mathcal{D}, \boldsymbol{\theta})$. Because a MCMC is a Bayesian method, we define $\mathbb{P}(\mathcal{D}, \boldsymbol{\theta}) = \mathbb{P}(\mathcal{D}|\boldsymbol{\theta})\mathbb{P}(\boldsymbol{\theta})$, where $\mathbb{P}(\mathcal{D}|\boldsymbol{\theta})$ is the likelihood and $\mathbb{P}(\boldsymbol{\theta})$ is the prior probability. The classes of `stattools` are based on this definition of a statistical model:

- Two fundamental classes are `TParameter` and `TObservation`. Class `TParameter` implements a standard MCMC parameter $\boldsymbol{\theta}$ where the goal is to obtain posterior samples from. Class `TObservation` implements an observation $\mathcal{D}$ that stores observed data.
- The relationship between observations and parameters (as well as between parameters in a hierarchical model) are given by probability distributions or deterministic functions. Classes deriving from the base class `prior::TBase` implement common probability distributions and deterministic functions.

A statistical model can be visualized graphically with using a so-called directed acyclic graph (DAG). In a DAG, the parameters and observations are shown as nodes, and directed edges connecting two nodes represent an assumption about non-independence. We will extend the concept of a DAG to illustrate the relationship between the `stattools` classes, as shown in Figure 1. In line with this graphical representation, we will refer to parameters and observations as *nodes* and to the relationship between the nodes as *boxes*.

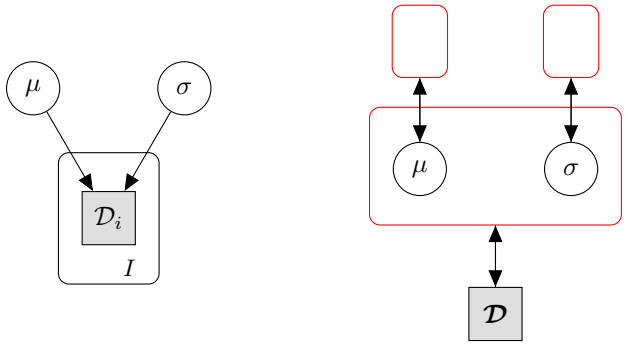

Figure 1: *Directed acyclic graph (DAG) of a simple model where the data $\mathcal{D}_i$ with $i = 1, \ldots, I$ follows a normal distribution $\mathcal{D}_i \sim \mathcal{N}(\mu, \sigma)$. The shaded nodes represent observations. Shown on the left is the classic representation of this model using a DAG. Shown on the right is the `stattools` representation of this model. The red boxes represent the probability distributions, with the large box around μ and σ being the normal distribution and the two small boxes on top being two prior distributions $\mathbb{P}(\mu)$ and $\mathbb{P}(\sigma)$. The edges represent memory pointers in two directions, pointing from the box above to the node below and vice-versa.*

1.2 Nodes

Class `TNodeTyped` is a pure virtual base class for `TObservation` and `TParameter` and defines the common interface of observations and parameters. The most important member variables are:

- `_storage`: A storage object. This stores the current and old values of the parameter and manages its dimensions.
- `_boxAbove`: A pointer to the box above.

Importantly, `TNodeTyped` is templated with a typename `Type`. This allows an efficient and flexible memory management. In most cases, the pre-defined types from the library `coretools` are used, e.g. `Probability`, `StrictlyPositive` etc. However, a developer is free to implement custom types. In addition, `TNodeTyped` is also templated by the number of dimensions. Note that this does not correspond to the size of the parameter (which might not be known at compile time), but to the dimensionality, e.g. two-dimensional. `TNodeTyped` inherits from a non-templated base class, `TNodeBase`, which implements the interface needed to interact with the parameters/observations from the "outside", e.g. for file handling or coordinating the MCMC updates.
We will now go through the different member variables and functions and describe them in more detail. Often, the member variables themselves are classes that contain objects of other classes. We will therefore start at the most basic classes and then explain how other classes make use of them.
First of all, it it obvious that every parameter and observation must store its value(s).

1.2.1 `TValueUpdated`

The class `TValueUpdated` is used to store values of parameters. It is templated with `Type` and stores the current (`_value`) and old (`_value_old`) value of the parameter in order to compute the Hastings ratio. It overloads the assignment `operator=()`, which swaps `_value_old` and `_value` and then sets `_value` to the value provided. It further overloads the cast `operator Type()` to return the current value.
In constrast to parameters, observations do not need to store old and new value since they are not updated. Observations simply store instances of `Type`. The interface for getting and setting however is the same regardless which value type is used: The assignment `operator=()` and the cast `operator Type()` are defined for both.

1.2.2 `TMultiDimensionalStorage`

Parameters and observations can be single values, one-dimensional vectors or n-dimensional matrices. All these data structures can be stored as a linear vector by transforming a linear index to a multi-dimensional coordinate and vice versa. For example, a 2-dimensional matrix is stored in a linear fashion by pasting row by row together (row-major layout). The main advantages of this design are that 1) this scales to any dimension, as we know how to linearize a n-dimensional index and back, 2) this provides a coherent interface, independent of the dimension and 3) if a smart layout is chosen (cache-friendly), it is faster than looking up values in a nested structure.
This concept is implemented in class `TMultiDimensionalStorage`. It is templated with the type (`Type`) and the number of dimensions (`NumDim`) of the underlying storage. It has three important member variables:

- A vector of type `Type` that stores the values.
- An instance of `TDimension`, storing the dimensions of these values.
- An array of pointers to `TNamesEmpty`, where each element in the array stores the names of one dimension.

`TMultiDimensionalStorage` implements the operator `operator=()` that provides access to the values of the parameter. To get linear indices from n-dimensional coordinates, `TMultiDimensionalStorage` also provides wrapper functions of `TDimension` (e.g. `getIndex()`, `getRange()`, `getFull()`, `get1DSlice()` etc.). Hence, `TMultiDimensionalStorage` manages both values and

their dimensionality.

Class `TMultiDimensionalStorage` provides different options to fill values. If the dimensions are known at run-time, the function `resize()` allocates the memory which can then be set with the `[]` operator. However, the dimensions might not be known in advance. For example, when reading in data from a file, we typically know the number of columns from the header, but we do not know the number of rows until we have reached the end of the file. We provide special functions for such cases:

- Before reading the data, the function `prepareFillData()` must be called. This function prepares the memory required to fill the data. The basic idea is that for most input files, all except one dimension are known. For example, in a VCF file, we know from the header how many individuals there will be. However, we will not know how many loci there will be until we parsed the entire file. We might have a guess, but it can be less or more. This function threrefore expects two arguments. The first argument is a guess of the length of the first dimension (i.e. the most outer dimension when linearizing the data, see chapter about dimensions). The second argument is an array of all known dimensions. The guess of the length of the first dimension makes the memory-filling process more efficient, because we reserve the memory in advance (with the vector function `reserve()`) and then fill it (with the vector function `emplace_back()`). If we need more memory than guessed, this is not a problem, we will simply `emplace_back()` to the vector.
- Then, we can fill the data using `emplace_back()`. Note that data must be filled in an ordered manner (i.e. the way it is supposed to be stored), no re-ordering will be possible afterwards.
- When all data has been filled, the function `finalizeFillData()` must be called. This function calculates the effective length of the first dimension based on the number of elements that were filled and the known dimensions; and resizes the vector of stored values to its actual size (to release redundant memory).

In the following section, we will dive deeper into the concept of linearizing n-dimensional arrays and explain how this is implemented in `TDimension`.

1.2.2.1 `TDimension`

`TDimension` is templated with the number of dimensions, as this is known at compile time. It stores a array `_dimensions` of size `NumDim` that represents the length of each dimension.
When initializing a `TDimension` object, the total size of the array (= the product of all dimensions) is automatically calculated and stored in variable `_totalSize`.
The linear index s is calculated from a D-dimensional array of coordinates $\boldsymbol{n} = n_1, \ldots, n_D$ as

$$s = \sum_{i=1}^{D} \left(\prod_{j=i+1}^{D} N_j \right) n_i, \tag{2}$$

where N_j corresponds to the length of dimension j. To optimize speed, this double-loop is written out for 1-, 2- and 3-dimensional cases, respectively:

$$s = n_1 \text{ and } s = n_2 + n_1 N_2 \text{ and } s = n_3 + N_3(n_2 + n_1 N_2).$$

The D-dimensional array of coordinates $\boldsymbol{n} = n_1, \ldots, n_D$ is calculated from a linear index s as follows:

1. Start at the most inner dimension $i = D$.
2. Calculate the index in the current dimension i with $n_i = s \bmod N_i$.
3. Update the linear index by removing the effect of the current dimension: $s' = \frac{s-n_i}{N_i}$.
4. Repeat steps 2-3 for next dimension until the most outer dimension is reached.

Again, this algorithm can be written out for speed for 1- and 2-dimensional arrays:

$$n_1 = s,$$

and

$$n_2 = s \bmod N_2 \text{ and } n_1 = \frac{(s - n_2)}{N_2}.$$

Most other functions of class `TDimension` use function linearizeIndex to linearize a vector of coordinates, and then return an object of class `TRange`. `TRange` is a very simple class that consists of three members: `begin`, `end` and `increment`. These correspond to the first index in the linear array, the last (not included) index in the linear array, and the increment one has to use to get the correct elements. It is then straightforward to use `TRange` in a for-loop like this:

```
for (size_t i = range.begin; i < range.end; i += range.increment){
  // ...
}
```

The following functions are implemented in `TDimension`:

- `getIndex()`: takes a vector of coordinates, and returns the linear index.
- `getDiagonal()`: only for 2-dimensional square matrices, throws otherwise. Returns a range object for all linear indices of the elements on diagonal.
- `getRange()`: takes two vectors of coordinates (start and end) and returns a range object of all elements between those.
- `get1DSlice()`: takes a number `dim` and a vector of coordinates, `start`. It will start at the linear index of `start`, follow the given dimension `dim` until its end, and return the corresponding range object. A classic example would be a 2-dimensional matrix, where we want to start at element $\{4, 0\}$ (fifth row, first column), and go over all elements of dimension 1 (= columns), to get one full row (end: $\{4, ncol\}$).
- `getFull()`: returns a range of all linear indices from first to last element, with increment 1.

1.2.2.2 `TNamesEmpty`

As mentioned above, class `TMultiDimensionalStorage` stores a vector of pointers to `TNamesEmpty`, such that each dimension has a pointer to the dimension names. For example, a three-dimensional parameter will have a vector with three pointers to `TNamesEmpty`, corresponding to the dimension names of the first, the second and the third dimension. `TNamesEmpty` is a pure virtual base class that provides a common interface to all deriving name classes. The `[]` operator can be used to get the name as a string for an element with a certain index of a dimension. The following classes derive from it:

- `TNamesStrings`. This class is nothing more but wrapper of a vector of strings: it provides functions to resize the vector and to add or set elements to it. The `[]` operator simply returns the element of the vector with the corresponding index. This class is e.g. useful to store sample names or names of covariates.

- `TNamesIndices`. This class stores only the offset for indices (by default 1, but can be set to zero). The `[]` operator adds its argument to the offset and returns this result as a string. This is the most simple class, and is typically used for simulating parameters (when there are no specific names), or for parameters whose elements correspond to "categories", e.g. the event probabilities from a categorical distribution.

- `TNamesIndicesAlphabetUpperCase`. This class stores nothing and returns the (numeric) index given to the `[]` operator as a alphabetic representation in upper case letters of that index in Excel column name stype (i.e. $0 \rightarrow$ A, $1 \rightarrow$ B, ..., $25 \rightarrow$ Z, $26 \rightarrow$ AA, $27 \rightarrow$ AB, $51 \rightarrow$ AZ, $52 \rightarrow$ BA etc.)

- `TNamesIndicesAlphabetLowerCase`. This class stores nothing, but returns the (numeric) index given to the `[]` operator as a alphabetic representation in lower case letters of that index in Excel column name stype (i.e. $0 \rightarrow$ a, $1 \rightarrow$ b, ..., $25 \rightarrow$ z, $26 \rightarrow$ aa, $27 \rightarrow$ ab, $51 \rightarrow$ az, $52 \rightarrow$ ba etc.)

- `TNamesPositions`. This class stores a pointer to `TPositionsRaw`, which is a class that stores a vector of chunk names (as strings) and a vector of positions (as `uint32_t`). The `[]` operator returns a string consisting of chunk name and position (concatenated by some delimiter). This class is useful to store positional information, e.g. chromosome (as chunks) and loci (positions).

The idea behind these classes is that the developer reads the observations and creates a name class, depending on which kind of names need to be stored. The name class for each dimension is then given over the function `setDimensionName()` to the observation. For example, when reading a VCF-file containing genotype likelihoods, the developer will need to create and fill three such name classes: one for the loci (i.e. `TNamesPositions`), one for the individuals (i.e. `TNamesStrings`) and one for the genotype (i.e. `TNamesIndices`). The developer will then pass them to the instance of the observation class storing these genotype likelihoods. After this, when the storage of the DAG is initialized, the box specified on these data knows how to match its parameter dimensions to the observation dimensions. For example, if one would specify a normal mixed model directly on these genotype likelihoods, the hidden state would be per locus, and would therefore retrieve the pointer to the loci-names from the observations below. When writing to a file, each parameter will hence have the correct dimension names.

1.3 Observations

We define observations as the node(s) located at the bottom of the DAG that contain observed data and that are not to be estimated. The class `TObservation` represents such an observation. In its constructor, it gets an instance of `TMultiDimensionalStorage` that contains the data and that is not modified afterwards.
`TObservation` is templated with its type, the number of dimensions and the type of the box above. It inherits from `TObservationBase`, which is not templated and provides a pure virtual interface. It further inherits from `TNodeTyped`.

1.4 Parameters

We define parameters as the node(s) located on the upper levels of the DAG that are to be inferred. The class `TParameter` represents such a parameter. Most importantly, `TParameter` has functionalities to update itself with Metropolis-Hastings or Gibbs sampling.
`TParameter` is templated with typename `Spec` that specifies the compile-time features of the parameter, as well as with the type of the box above. `TParameter` inherits from `TParameterBase`, which is not templated and provides a pure virtual interface. It further inherits from `TNodeTyped`.

1.4.1 `ParamSpec`

This class specifies the compile-time features of a parameter. It is templated with the following mandatory arguments:

- `typename Type`: The type that defines the interval of the parameter. Can be any of the weak types from `coretools`, e.g. `Unbounded`, `Probability`, `StrictlyPositive` etc.
- `Hash<size_t H>`: A unique hash, generated from the name of the parameter. Used to make each parameter uniquely identifiable.
- `typename TypeBoxAbove`: The full type of the box above the parameter.

`ParamSpec` is futher templated with a template parameter pack (`...`) that represent optional arguments. Since all of these optional arguments have a unique type identifier, they can be given in any order. If they are not specified, a default value is used.

- `NumDim<size_t N>`: The number of dimensions. Default is `N = 1`.
- `typename Constraint`: A constraint. Currently supported constraints are `Unconstrained`, `SumOne` and `LengthOne`. Default is `Unconstrained`.
- `Parallelize<MarkovOrder Order>` Indicates if the update of the parameter can be parallelized, and with which Markov order. The template argument `MarkovOrder` is a `enum` with possible values `allDependent`, `allIndependent` and `different`. If `allDependent`, all elements in the vector all dependent on each other, such that parallelization is not possible. If `allIndependent`, all elements in the vector are independent of each other, such that the update can be fully parallelized among the indices. If `different`, there are some dependencies, but they are possibly only known at runtime and must then be provided in the constructor of `TParameter`. Default is `MarkovOrder::allDependent`, resulting in no parallelization.
- `Weights<size_t N, UpdateWeights...W>` Update weights, one per dimension. Currently supported weights are `regular`, `irregular`, `geometricUniform`, `log10StatePosterior` and `powerStatePosterior`. Default is `W = regular` for each dimension.
- `RJMCMC<typename SpecModelParameter>` Whether or not this parameter is an RJ-MCMC parameter. A RJ-MCMC parameter is a parameter that is present or absent in a model, depending on an indicator variable. Default is `NoRJMCMC`.

1.4.2 Parameter constraints

Certain parameter vectors have constraints on their values. We consider two cases. First, a unit vector $\boldsymbol{x} = \{x_1, \ldots, x_I\}$ has length one, i.e. $\sqrt{\sum_{i=1}^{I} x_i^2} = 1$. Second, a normalized vector sums to one, i.e. $\sum_{i=1}^{I} x_i = 1$.

The elements of such parameters can not be updated independently in the MCMC without violating the constraints. There are two options to update constrained parameters: joint and pair-wise. In a joint update, all elements of the parameter are slightly modified and then re-normalized to satisfy the constraint. In a pair-wise update, two elements are updated together such that the constraint is still satisfied.
In the following paragraphs, we will discuss the different updating schemes.

1.4.2.1 Pairwise update of unit vectors

Let us denote by $\boldsymbol{x} = \{x_1, \ldots, x_I\}$ a unit vector such that $\sqrt{\sum_{i=1}^{I} x_i^2} = 1$. Let us further denote by i, j the two indices that are being updated. A pair-wise update of x_i and x_j will always result in a valid unit vector $\boldsymbol{x}'$ if

$$x_i^2 + x_j^2 = x_i'^2 + x_j'^2. \tag{3}$$

This condition can be solved by representing the problem as a circle where the radius is given by

$$r_{ij}(\boldsymbol{x}) = x_i^2 + x_j^2.$$

Every value on the circumference of this circle will satisfy the condition given by equation (3). The proposal kernel to get a new value $p_{ij}(\boldsymbol{x})' = (x_i', x_j')$ should be symmetric to avoid extra computations in the Hastings ratio. A computationally cheap and symmetric proposal kernel is to sample a random point along the tangent line to the circle and then normalize its norm to $r_{ij}(\boldsymbol{x})$ to project it on the circle.
The slope of the tangent line is given by the vector

$$s_{ij}(\boldsymbol{x}) = \begin{pmatrix} 1 \\ -\frac{x_i}{x_j} \end{pmatrix},$$

which is normalized to unit length:

$$\hat{s}_{ij}(\boldsymbol{x}) = \frac{s_{ij}(\boldsymbol{x})}{||s_{ij}(\boldsymbol{x})||}$$

We then propose a random jump α according to some proposal kernel and calculate the new values $p_{ij}(\boldsymbol{x})'$ as

$$p_{ij}(\boldsymbol{x})' = \gamma \left(p_{ij}(\boldsymbol{x}) + \alpha \hat{s}_{ij}(\boldsymbol{x}) \right),$$

where γ is the normalizing ratio

$$\gamma = \sqrt{\frac{||p_{ij}(\boldsymbol{x})||}{||p_{ij}(\boldsymbol{x})'||}} = \sqrt{\frac{x_i^2 + x_j^2}{x_i'^2 + x_j'^2}}.$$

1.4.2.2 Pairwise update of normalized vector

Let us denote by $\boldsymbol{x} = \{x_1, \ldots, x_I\}$ a vector with length one such that $\sum_{i=1}^{I} x_i = 1$. Let us further denote by i, j the two indices that are being updated. A pair-wise update of x_i and x_j will always result in a valid vector $\boldsymbol{x}'$ of length one if

$$x_i + x_j = x_i' + x_j'. \tag{4}$$

Since x_i and/or x_j can be very small, we scale the proposal width by the sum $x_i + x_j$. Specifically, our goal is to propose a jump $u \sim \alpha\sqrt{x_i^2 + x_j^2}$ where $\alpha \sim \mathcal{N}(0, \sigma)$ and $\sqrt{x_i^2 + x_j^2}$ is the length of

the line constituted by all valid x_i, x_j combinations. Given jump u, the new values are simply $x_i' = x_i + u$ and $x_j' = x_j - u$. However, we need to make sure to mirror at the maximum given by $x_i + x_j$. To achieve this, we re-formulate the above as:

$$x_i' \sim \mathcal{N}(x_i, \sigma\sqrt{x_i^2 + x_j^2}),$$

where we mirror at $x_i + x_j$. The new value of x_j' is then simply given by $x_j' = x_i + x_j - x_i'$.

1.4.3 Update weights

In general, each element of a parameter is updated exactly once per MCMC iteration. However, for certain parameters, we might have a guess about which elements are more relevant than others. These elements should preferrably be updated more often, therefore allocating computational resources to the elements that need fine-tuned updating. This ideally improves convergence of the chain, allowing for shorter MCMCs.
The idea is to allocate update weights to each dimension of a parameter. For example, the first dimension of a parameter is updated according to some specific weights, whereas the second dimension is updated "regularly", i.e. each element once. The enum `UpdateWeights` specifies all possible weighting schemes. The parameters of the weighting schemes can be passed as vectors of strings, if necessary, but there are sensible default values for all schemes.

1.4.3.1 `UpdateWeights::regular`

Returns equal weights for all indices.

1.4.3.2 `UpdateWeights::irregular`

Returns the (untransformed) weights provided by the developer. This is useful if some tool-specific algorithm is used to calculate the weights.

1.4.3.3 `UpdateWeights::geometricUniform`

The input statistics are ranks r_i for all indices $i = 1, \ldots, N$, for instance obtained by the ranking of a statistical test. The weight Q_i of index i is then given by $Q_i = \frac{3}{10}U_i + \frac{7}{10}G_i$, where U_i is a discrete uniform distribution between 1 and n, and $G_i = (1-p)^{r_i-1}p$ is the geometric distribution. We determine the parameter p of the geometric distribution based on two probabilities t and x, which specify the top fraction t of all elements that receive a fraction x of the effort in updating. By default, $t = 0.2$ and $x = 0.9$, meaning that the top 20% elements receive 90% of the updating effort. For $n \to \infty$, the parameter p can be computed as

$$p = 1 - \exp\left(\frac{\log(1-x)}{tN}\right),$$

by solving the cumulative distribution function for t elements for p. It can be shown that this approximation is valid if $n > 150$. Figure 2A visualizes the weight distribution for different values of t and x.

1.4.3.4 `UpdateWeights::log10StatePosterior`

The input statistics are state posterior probabilities p_i for all indices $i = 1, \ldots, N$. The weight Q_i of index i is then given by the absolute logarithmic value of the complement of the state posterior probability:

$$Q_i = -\log_{10}(1 - p_i),$$

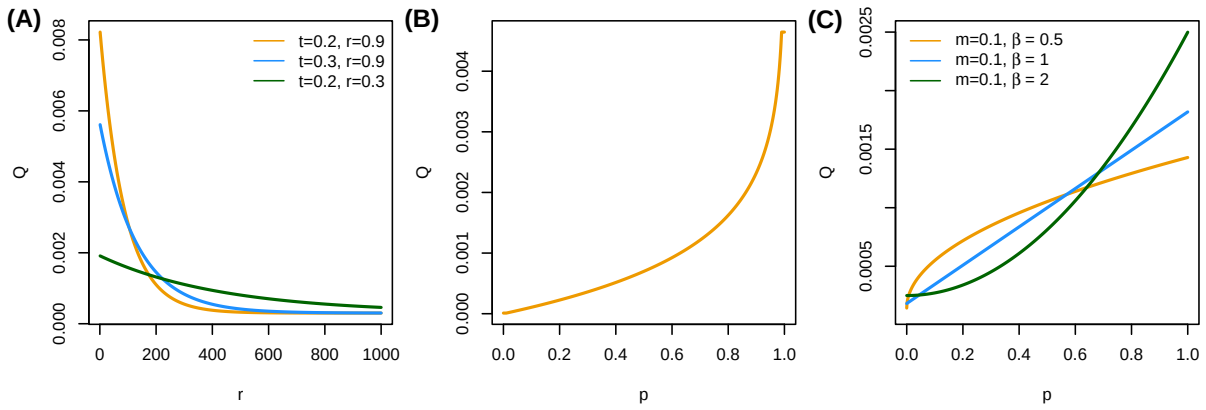

Figure 2: *Update weights as a function of the input statistics. (A) The weights Q as a function of the input ranks r for* `UpdateWeights::geometricUniform`*. (B) The weights Q as a function of the input state posterior probabilities p for* `UpdateWeights::log10StatePosterior`*. (C) The weights Q as a function of the input state posterior probabilities p for* `UpdateWeights::powerStatePosterior`*.*

where we cap p_i at 0.01 and 0.99, respectively. The Q_i are normalized such that they sum to one, $\sum_i Q_i = 1$. With this weight distribution, indices with high state posterior probabilities will receive higher update weights (Figure 2B).

1.4.3.5 `UpdateWeights::powerStatePosterior`

The input statistics are state posterior probabilities p_i for all indices $i = 1, \ldots, N$. The weight Q_i of index i is then given by a power function with the state posterior probability as a base:

$$Q_i = m + (1 - m)p_i^{\beta},$$

where m is an intercept in [0,1] and β is the slope (Figure 2C). If $\beta = 1$, a linear function results. If $\beta < 1$, a concave function with a flat slope results that levels off for $p_i \to 1$. If $\beta > 1$, a convex function with a steep slope results. By default, $\beta = 0.5$ and $m = 0.1$. The Q_i are normalized such that they sum to one, $\sum_i Q_i = 1$.

1.4.3.6 `TUpdateWeights`

For each dimension of a parameter, an enum `UpdateWeights` has to be specified at compile time. The update weight of a (multi-dimensional) index is given by the product of the update weights for all dimensions: $Q_i = Q_{\{j_1,\ldots,J_D\}} = \prod_{d=1}^{D} Q_{j_d}$, where $\{j_1, \ldots, J_D\}$ corresponds to the subscripts of the linear index i in all $d = 1, \ldots, D$ dimensions. The class `TUpdateWeights` provides several functions that calculate those linear weights and calculate the cumulative of the linear weights, which are ultimately needed for choosing an index to update.

1.4.4 Choosing indices to update

As discussed previously, a parameter can be updated regularly, i.e. each index exactly once per iteration, or else according to update weights. In addition, constraints on a parameter dimension may require special update types, such as an update in pairs or a joint update of the entire dimension. Finally, all update functions should be safe to parallelize, i.e. one index should never be updated by two threads at the same time. To achieve this, multiple classes handle each

specific combination of cases. Each class implements the function `numUpdates()` that specifies the number of updates to perform per iteration. In addition, each class implements the function `pick()` that takes the current iteration and thread as arguments, and returns an instance of `TRange` that specifies one or multiple indices to updated.

1.4.4.1 `TUpdateSingle`

This class handles updates when there are no constraints and when `UpdateWeights::regular` is used for all dimensions. In this case, each index should be updated exactly once per iteration. The initialization function of this class takes an instance of `TMarkovOrder`. This class `TMarkovOrder` is described in section 1.4.5 and defines a range of parameter indices that can be updated independently and thus can be parallelized. There are possibly multiple instances of `TUpdateSingle`. Each instance will loop over the indices within its internal range. The function `pick()` then transform the internal index to the index of the full parameter space as described in 1.4.5.

1.4.4.2 `TUpdateSingleWeighted`

This class handles updates when there are no constraints and when at least one dimension uses non-regular update weights. The class randomly selects an index to update based on the cumulative update weights. If the update is parallelized, we need to make sure that there is no race conditions, i.e. that one index is never updated by two threads simultaneously. To achieve this, we assign a range of indices to one specific thread $t = 1, \dots, T$ by splitting the vector of cumulative weights into T vectors, one per thread. The function `pick()` then draws an index from the cumulative weights of the current thread.
To split the vector, we use a simple splitting scheme that aims at assigning approx. the same cumulative weight to each thread, but making sure that each thread gets at least one index to update. We note that this might not result into the exact weights. For example, if there are three elements with the cumulative weights $\{0.1, 0.2, 1.0\}$ and three threads are used for updating, each element will be assigned to one thread and updated equally often. However, for more indices and a less extreme weight distribution, this becomes less of an issue. Ultimately, the update weights are rough estimates too and should not be problematic if a parameter is updated a bit more or less than the update weights predict.
In addition, as in class `TUpdateSingle`, the initialization function takes an instance of `TMarkovOrder` to define a subset of indices to update. The weights given to the class are then subsetted to the indices from this sequence.

1.4.4.3 `TUpdatePair`

This class handles updates when `UpdateWeights::regular` is used for all dimensions and when there are constraints such that parameters must be updated in pairs. The template argument `AlongDim` specifies the dimension along which the constraint holds. The class `TIndexSampler` is used to generate random index pairs with the use of a single random number. This random number corresponds to one random position i and it pairs up with index $j = i + 1$. From this, all other pairs are created by shifting $i' = i - 1$ and $j' = j + 1$ repetitively, where the indices are set to $N - 1$ when crossing zero and to zero when crossing $N - 1$. The pairwise indices thus run along one dimension, which represents a one-dimensional slice. In the case of D dimensions with length N_d each, there are $K = \prod_{d \neq AlongDim}^{D} N_d$ such slices. The class `TIndexSamplerMultiDim` transforms a linear index in the parameter space into the index relevant for the pairwise index sampler and then back-transforms the index of the pair, corresponding to the index in `AlongDim`,

into two linear indices in the parameter space. This is thread-safe, since each parameter index occurs at most once within all possible index pairs.

1.4.4.4 `TUpdatePairWeighted`

This class handles updates when at least one dimension uses non-regular update weights and when there are constraints such that parameters must be updated in pairs. The template argument `AlongDim` specifies the dimension along which the constraint holds. In this case, we can not simply assign indices to threads as before, since then certain indices will never be assigned to the same pair. To prevent race conditions, we store all the indices that are currently updated in a vector, and only allow a thread to update an index if it is not present in this vector. This section of the code is marked with `pragma critical` such that only one thread can access it at a time. The first index of a pair is drawn at random from the cumulative update weights. The second index of the pair is then also drawn at random from the cumulative update weights, but ensuring that i) it is located in the same slice as the first index and ii) it is not the same as the first index. To achieve this, we re-scale the cumulative weights in the dimension `AlongDim` by excluding the weight of the first index, and drawing at random from these weights.

1.4.4.5 `TUpdateJoint`

This class handles updates when `UpdateWeights::regular` is used for all dimensions and when there are constraints such that all parameters of a certain dimension, given by the template argument `AlongDim`, must be updated jointly. In this case, each one-dimensional slice along `AlongDim` should be updated exactly once per iteration. Note that in this case we don't allow for update weights.

1.4.5 Parallelization of updates

When parameter indices can be updated independently, their update can be parallelized. The template argument `MarkovOrder` of the parameter specification is used to define the dependency assumptions. The three possible enums `allDependent`, `allIndependent` and `different` correspond to no, full and partial parallelization, respectively.
For the `different` case, the Markov order must be given at runtime to the constructor of `TParameter`. In general, a Markov order of X means that every $(X+1)^{th}$ element can be updated independently. For example, a Markov order of zero corresponds to full independence of elements as every element can be updated independently. A Markov order of one corresponds to the classic HMM case where every second element can be updated independently, as the update of one index only depends on the direct neighbours. The largest possible Markov order is the size of the parameter minus one, which corresponds to full dependency.
This concept can be generalized to multiple dimensions where each dimension is assigned a Markov order.
Having specified the Markov order, the challenge is then to parallelize the loop over indices while respecting the Markov order. To simplify the problem, we currently only allow for parallelization in unconstrained parameters, i.e. if either of the classes `TUpdateSingle` or `TUpdateSingleWeighted` are used to pick indices to update. We then define a vector of pickers that corresponds to multiple instances of these two classes. Each picker is assigned to a subset of all parameter indices that can be fully parallelized. In the function `update()` of `TParameter`, we first loop over all pickers, which is never parallelized. We then loop over all indices within one picker, which is fully parallelized using `OpenMP`.

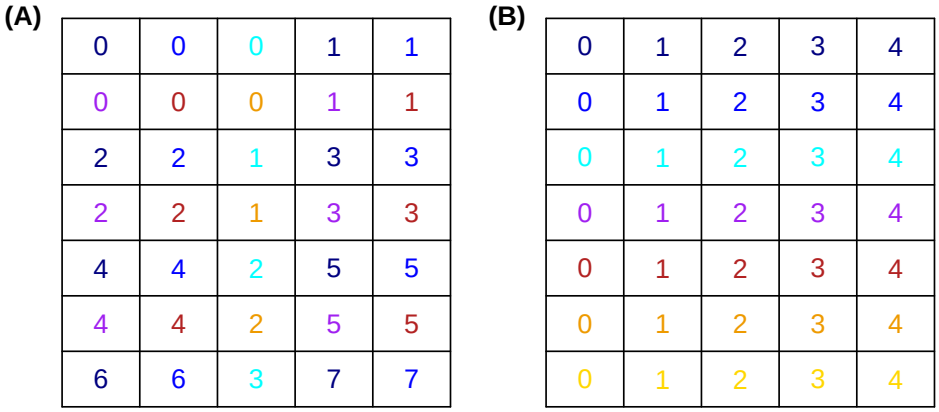

Figure 3: *Two examples of Markov orders for a two-dimensional parameter with seven rows and five columns. The colors represent the index pickers, and the numbers represent the index within the index picker. (A): Markov order = (1, 2) results in six index pickers of different dimensions. (B): Markov order = (6, 0) results in seven index pickers, one for each row.*

The class `TMarkovOrder` manages the compression of full indices to picker indices and their expansion back to full indices. Let us define by $\boldsymbol{m} = (m_1, \ldots, m_D)$ the D-dimensional Markov order and by $\boldsymbol{f} = (f_1, \ldots, f_D)$ the dimensions of the parameter.
The number of pickers P is given by

$$P = \prod_{d=1}^{D} n_d,$$

where $n_d = m_d + 1$ denotes the Markov order m_d increased by one. Each picker $p = 1, \ldots, P$ is then assigned a starting value. This is obtained by calculating the D-dimensional coordinates $\boldsymbol{s}_p$ of the linear index p in the space spanned by $\boldsymbol{n} = (n_1, \ldots, n_D)$ as explained in section 1.2.2.1. This is visualized in Figure 3. The internal dimensions $\boldsymbol{q}_p = (q_{p1}, \ldots, q_{pD})$ of each picker p are then given by

$$q_{pd} = \left\lceil \frac{f_d - s_{pd}}{n_d} \right\rceil,$$

for each dimension d. This simply corresponds to the total length of the dimension when starting at s_{pd}, divided by the step size n_d, and then rounding up.
To transform internal index i of picker p to the index j in the full parameter space, we first transform the linear index i to the D-dimensional coordinates $\boldsymbol{k} = (k_1, \ldots, k_D)$ in the space spanned by $\boldsymbol{q}_p$. We then expand these coordinates to the full space using

$$k'_d = s_{pd} + k_d * n_d, \tag{5}$$

for each dimension d. This simply corresponds to multiplying the index k_d by the step size n_d and adding the starting value s_{pd} to it. The linear index j is then obtained by linearizing the coordinates $\boldsymbol{k}'$ using (2).
We consider the special case that all elements are independent. In this case, there is a single picker and the equation (5) simplifies $j = i$.
Finally, it would be quite inefficient if all indices had their own picker when all indices are dependent, i.e. if the enum `allDependent` is used (which is the default). In this case, class

`TParameter` sets the Markov order to all zeros, which results in a single picker. Using compile-time statements, it is ensured that the number of threads for parallelization is restricted to one, which makes the use of this single picker safe to use.

1.4.6 Proposing new values

We have now seen the different mechanisms on how indices are selected for an update. In the following section, we will describe on how the update is effectively performed for the selected indices. The base class for this is `TUpdateTypedBase` that manages the Metropolis-Hastings update of the parameter value. It holds a pointer to a proposal kernel that proposes a new value based on some proposal width.
Class `TUpdateTypedBase` is templated and its type matches the type of the parameter. The proposal widths, however, are of the underlying type (and not of the type itself). The reason is that for certain custom types, e.g. for strictly negative values, the proposal width could only hold negative values - but this is clearly not useful, since we would have to implement a special proposing scheme for such types.
After each burnin, `TUpdateTypedBase` is responsible for:

1. Adjust the jump size j by calculating

   $$j' = ju,$$

   with

   $$u = \left(\frac{(a+1)}{r(T+1)} \right) \tag{6}$$

   where a is the number of accepted updates, T is the number of total updates and r is the ideal acceptance rate. We add one to avoid issues if the parameter has never been accepted. By default, r is set to 0.44, since this appears to be the optimal acceptance rate for the Metropolis-within-Gibbs algorithm where a single coordinate is updated at a time (Brooks et al., 2011). However, r can be overridden by providing the command-line argument `accRate`.

2. Clear all counters (the number of accepted and total updates).

Its derived classes implement different options on how the proposal widths are handled.

1.4.6.1 `TUpdateShared`

The class `TUpdateShared` inherits from `TUpdateTypedBase` and uses the same proposal width for all indices of parameters. This is useful if the parameter values are approx. the same order of magnitude.

1.4.6.2 `TUpdateUnique`

The class `TUpdateUniqueRegular` inherits from `TUpdateTypedBase` and uses a proposal width that is specific to each parameter index. This is useful if the parameter values are of different orders of magnitude, such that using the same proposal width for all will result in poor mixing for some elements. The class is templated with `Regular` that indicates if the update weights are all regular. If all weights are regular, each index is updated equally often, so the total number of updates can be a single number. If the weights are not regular, indices might be updated more often than others, therefore, the total counter of updates must run per index.

1.4.7 `TPropKernelBase`

The actual proposal of a new value uses a certain proposal distribution. This is handled by the class `TPropKernelBase` and its derived classes.
Proposal kernels 1) propose a value j according to some distribution, 2) add it to x, and 3) mirror. All proposal kernels must restrict the value of j, in order to prevent too large jumps that could cause overshooting or numerical issues. We therefore make sure that the proposal width δ is always smaller than half of the value's range r, $\delta \leq \frac{r}{2}$. All of the following proposal kernels except `TPropKernelScalingLogNormal` are symmetric and cancel out in the Hastings ratio.

1.4.7.1 `TPropKernelUniform`

Uniform proposal kernel where $j \sim \mathcal{U}(\delta)$. A jump is obtained with $j = R\delta - \frac{\delta}{2}$, where R is a random uniform number between 0 and 1. This is enabled for floating point types.

1.4.7.2 `TPropKernelNormal`

Normal proposal kernel where $j \sim \mathcal{N}(0, \delta^2)$. A jump is obtained by drawing random normal values inside a while-loop until $-\frac{r}{2} \leq j \leq \frac{r}{2}$. This is enabled for floating point types.

1.4.7.3 `TPropKernelRandomInteger`

Integer proposal kernel with random sampling. The new value is randomly sampled from the entire range of the type. This class thus does not have a proposal width that is adjusted. It is designed for integral types with small ranges (e.g. categorical variables) where a proposal width is not meaningful (e.g. a update $0 \to 1$ should be equally likely as an update $0 \to 10$). Per default, this proposal kernel is used for types with a maximum value smaller or equal than 10.

1.4.7.4 `TPropKernelInteger`

Integer proposal kernel where $j \sim \mathcal{U}(-\delta, \delta)$. A jump is obtained by randomly picking any value in the range $0, 1, 2, \ldots, 2\delta$. The values are then shifted by δ in order to be symmetric. If $j = \delta$ is picked, we draw again, since this would result in no update after shifting. If mirroring results in the same value as before, we also draw again. Note that this update has a bad performance for integers with small ranges (e.g. categorical variables), since it requires mirroring which might be quite an overhead for small ranges. For these cases, we instead recommend `TPropKernelRandomInteger`.

1.4.7.5 `TPropKernelBool`

Boolean proposal kernel: $x' = 1 - x$. We always propose the opposite value for bools, so jump sizes j do not matter. This is enabled for bools only.

1.4.7.6 `TPropKernelScalingLogNormal`

A special log-normal proposal kernel that scales with magnitude of value. In general, a desired feature of a proposal kernel is that its width scales with the value x of the parameter that is updated: If x is very small, the proposal width should also be very small. If x is very large, the proposal width should also be very large. However, tuning the proposal width until it matches the value of x might take quite a few iterations, especially if the values of x are very large.

A typical solution for this is to update x in the log-space. However, updating x in the log-space is sometimes unpractical, as this naturally imposes a exponential prior with rate 1 on the parameter, which is sometimes too strong.
We therefore implemented a proposal kernel that scales with the value of x. We obtain a new value x' by multiplying a jump value j:

$$x' = jx,$$

where j follows a log-normal distribution: $j \sim \text{Lognormal}(0, \sigma^2)$. In fact, one can show that $x' = jx \sim \text{Lognormal}(\log x, \sigma^2)$.
Unfortunately, this proposal kernel is not reversible:

$$\begin{aligned} \mathbb{P}(x'|x) &= \mathbb{P}(x|x') \\ \frac{1}{x'\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(\log x' - \log x)^2}{2\sigma^2}\right) &= \frac{1}{x\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(\log x - \log x')^2}{2\sigma^2}\right) \\ x &= x' \end{aligned}$$

The ratio of the proposal kernels does not cancel out in the Hastings ratio. We need to correct the Hastings ratio with:

$$h = \frac{\mathbb{P}(x|x')}{\mathbb{P}(x'|x)} = \frac{x'}{x}.$$

It is important to note that this proposal kernel does only make sense for strictly positive or strictly negative values, since a sign change is not possible (j will always be positive, therefore x' will always keep the sign of x).
Since the proposal kernel is not reversible, using it to infer bounded parameters by mirroring at those boundaries is also more complicated. Mirroring for reversible proposal kernels is valid because

$$\mathbb{P}(x|x') + \mathbb{P}(2a - x|x') + \mathbb{P}(2b - x|x') = \mathbb{P}(x'|x) + \mathbb{P}(2a - x'|x) + \mathbb{P}(2b - x'|x)$$

In our case, however, the distances to the proposed values are not the same. We could correct for this in the Hastings ratio, too, but we decided not to implement this, since it was not needed in our use-case. The proposal kernel is anyways automatically "mirrorred" at zero, as explained above.
The procedure to adjust the proposal width is also different than for the standard proposal kernels. The variance of j is given by the variance of a log-normal distribution with $\mu = 0$:

$$\text{Var}(j) = \left(e^{\sigma^2} - 1\right) e^{\sigma^2} = \left(e^{\sigma^2}\right)^2 - e^{\sigma^2}.$$

Our goal is to adjust the variance of the log-normal distribution, σ^2, according to the factor u from equation 6:

$$u\left(\left(e^{\sigma^2}\right)^2 - e^{\sigma^2}\right) = \left(e^{\sigma^{2\prime}}\right)^2 - e^{\sigma^{2\prime}}.$$

Solving for $\sigma^{2\prime}$ results in:

$$\sigma^{2\prime} = \log\frac{1}{2} + \log\left(1 + \sqrt{1 + 4u\left(e^{2\sigma^2} - e^{\sigma^2}\right)}\right).$$

1.4.8 `TMirror`

Let us have a closer look on how we make sure that the proposed values are inside a specific interval. All types in the framework are weak types from `coretools` that are defined on a specific interval. By default, this interval is given by the numeric limits of the underlying type. A couple of classes exist that implement the most common non-default intervals, such as positive, strictly positive, the open interval between 0 and 1, the closed interval between 0 and 1, etc. All of these types can return their minimum and maximum value, denoted here by a and b, respectively.
In its most basic form, mirroring works like this:

$$\tilde{x} = x + j; \quad x' = \begin{cases} \tilde{x} & \text{if } a \leq \tilde{x} \leq b \\ 2a - \tilde{x} & \text{if } \tilde{x} < a \\ 2b - \tilde{x} & \text{if } \tilde{x} > b \end{cases}, \tag{7}$$

where $\tilde{x}$ is an initial proposal, and x' is the value returned from the mirroring function. This only holds if $2a - \tilde{x} < b$ and $2b - \tilde{x} > a$ in all cases, so we need to make sure that $\tilde{x}$ is not too large (i.e. by restricting the maximal jump value j to half of the range). However there are different challenges associated with mirroring. A first challenge for both signed and unsigned types arises from the fact that we will run into numeric problems when using equation (7) for boundaries where a and/or b are given by the numeric limits of that type. If we first add $x + j$ and then mirror, we might have crossed the numeric limits already, but never notice it, as numeric overflow results in undefined behaviour. Hence, we must *first* check if we cross the boundaries, and only *after* do the addition. Note that the following rules apply to both default and non-default min and max.
We exceed the minimum if

$$-j > x - a, \tag{8}$$

because the distance between the minimum and the current value is smaller than the jump size. If this is the case, we need to mirror, using the logic of (7). However, calculating $2a$ might also result in numeric overflow, so we use the un-simplified equation $a - j - (x - a)$ in order to mirror. The same logic applies for mirroring at the maximum b. We exceed the maximum if

$$j > b - x, \tag{9}$$

because the distance between the maximum and the current value is smaller than the jump size. We then mirror with $b - (j - (b - x))$, as $2b$ would again result in numeric overflow.
For signed types, an additional challenge arises from the fact that adding/subtracting value with different signs can cause numeric overflows, since the value to store this range might be larger than the numeric maximum of the type. This issue is addressed in class `TMirrorSigned`, a class for mirroring signed types. If $x \geq 0$ and $a < 0$, we can safely compute $x - a$ if

$$x \leq a - \text{numeric\_min}.$$

If this is not the case, we do not need to care about mirroring anyways, as we restrict j to half of the range, and will therefore never make such a big jump. For mirroring at the maximum, we would again run into numeric overflow if $b \gg 0$ (e.g. the numeric maximum) and $x < 0$. We can safely compute $b - x$ if

$$-x \leq \text{numeric\_max} - b.$$

If this is not the case, we don't need to care about mirroring as well.

For unsigned types (e.g. uint8_t, uint16_t etc.), we do not have these range-overflow issues, as the numeric minimum is always 0. Therefore, class `TMirrorUnsigned` provides an implementation for unsigned types. The challenge for unsigned types is that we can not directly pass the jump size as an unsigned integer to the function, since unsigned values can't be negative, and if we passed j as an unsigned type, we would always increase x - clearly not what we want. We therefore pass both the jump size as well as the shift to the function for unsigned types (both as unsigned integers), and calculate whether we will jump to the right or the left and mirroring accordingly.

A last caveat are integer types. If we applied the above procedure to an integer type, we would need to correct for a non-symmetric proposal kernel in the Hastings ratio because of mirroring. As an example, consider an integral type with an interval $[0, 2]$ with possible jumps $\{-2, -1, 0, 1, 2\}$. There are two possibilities to propose the update $0 \to 1$: Either by taking jump 1 or by taking jump -1 and then mirroring at $a = 0$. However, there is only one possibility to propose the reversed update $1 \to 0$, as this is only possible by taking jump -1, but not by mirroring.
We solve this issue by mirroring at $a - \frac{1}{2}$ and at $b + \frac{1}{2}$ instead of mirroring at a and b. This results in a symmetric, reversible proposal kernel. Since $2\left(a - \frac{1}{2}\right) - \tilde{x} = 2a - \tilde{x} - 1$ and analogously $2\left(b + \frac{1}{2}\right) - \tilde{x} = 2b - \tilde{x} + 1$, we can simply subtract or add one after mirroring at the left or the right boundary, respectively.

1.4.9 RJ-MCMC

Reversible-jump MCMC (RJ-MCMC, Green (1995)) allows for MCMC updates of parameters that differ in dimensionality. In such cases, the Hastings ratio must account for the change in dimensionality as comparing densities of parameters of different dimensions is meaningless. This is relevant in the case of model choice, e.g. in mixture models. We refer to Wegmann and Leuenberger (2019) for a general description of RJ-MCMC. In `stattools`, we currently allow for a mixed model that consists of two nested submodels $\mathcal{M}_1 \subset \mathcal{M}_2$ where both models share the parameters $\boldsymbol{\theta}$ but model $\mathcal{M}_2$ has additional parameters $\boldsymbol{\sigma}$. To compensate for the difference in dimensionality, we use a random vector $\boldsymbol{u}$ with density $\pi(\boldsymbol{u})$ and of the same dimension as $\boldsymbol{\sigma}$. For the diffeomorphism Φ_{12} we choose the identity map $(\boldsymbol{\theta}, \boldsymbol{u}) \mapsto (\boldsymbol{\theta}, \boldsymbol{\sigma} = \boldsymbol{u})$ whose Jacobian determinant is 1 and thus cancels out in the Hastings ratio. The simplified Hastings ratio is then given by

$$h(\boldsymbol{\theta} \to (\boldsymbol{\theta}, \boldsymbol{\sigma})) = \min\left\{1, \frac{\mathbb{P}(\mathcal{D}|\boldsymbol{\theta}, \boldsymbol{\sigma})\mathbb{P}(\boldsymbol{\sigma})\mathbb{P}(\mathcal{M}_2)q_{21}}{\mathbb{P}(\mathcal{D}|\boldsymbol{\theta})\mathbb{P}(\mathcal{M}_1)\pi(\boldsymbol{\sigma})q_{12}}\right\}$$

for the jump from $\mathcal{M}_1$ to $\mathcal{M}_2$ and

$$h((\boldsymbol{\theta}, \boldsymbol{\sigma}) \to \boldsymbol{\theta}) = \min\left\{1, \frac{\mathbb{P}(\mathcal{D}|\boldsymbol{\theta})\mathbb{P}(\mathcal{M}_1)\pi(\boldsymbol{\sigma})q_{12}}{\mathbb{P}(\mathcal{D}|\boldsymbol{\theta}, \boldsymbol{\sigma})\mathbb{P}(\boldsymbol{\sigma})\mathbb{P}(\mathcal{M}_2)q_{21}}\right\}$$

for the reverse jump. Here, q_{12} denotes the proposal probability for the jump from $\mathcal{M}_1$ to $\mathcal{M}_2$, and q_{21} denotes the reverse probability. We fix $q_{12} = 1$, meaning that we will always propose a jump $\mathcal{M}_1 \to \mathcal{M}_2$. The reverse probability is set to $q_{21} = 0.1$ by default (and can be changed in the `TParameterDefinition`), meaning that in 10% of the cases we will propose a jump $\mathcal{M}_2 \to \mathcal{M}_1$. In all other cases, we will remain in model $\mathcal{M}_2$ and perform a standard Metropolis-Hastings update of parameter $\boldsymbol{\sigma}$ instead. We generally recommend to set q_{21} to rather small values, as this choice allows for a sampling rate of $\boldsymbol{\sigma}$ that is comparable to that of other parameters. If q_{21} was set to high values most updates would correspond to a model jump, and the MCMC would need to run much longer in order to generate sufficiently many posterior samples of $\boldsymbol{\sigma}$ for parameters where the $\mathcal{M}_2$ is preferred. $q_{21} = 0.1$ is an arbitrary choice but performed well in our application of RJ-MCMC.

The RJ-MCMC update is implemented as follows. For each RJ-MCMC parameter σ_i with $i = 1, \ldots, I$ there exists an indicator variable m_i that is one if σ_i is in model $\mathcal{M}_2$ and zero otherwise. Both parameters $\boldsymbol{\sigma} = (\sigma_1, \ldots, \sigma_I)$ and $\boldsymbol{m} = (m_1, \ldots, m_I)$ are instances of `TParameter`. In the parameter specification `ParamSpec` of $\boldsymbol{\sigma}$, the user must indicate that this is an RJ-MCMC variable with the parameter specification of $\boldsymbol{m}$ as a template argument. In addition, the proposal distribution for proposing new values of $\boldsymbol{\sigma}$ after jumping from $\mathcal{M}_1 \to \mathcal{M}_2$ must be specified. This can be any distribution from `coretools::probdist`. For example, for a $\boldsymbol{\sigma}$ that is bounded in the interval $(0, 1)$, the `ParamSpec` for $\boldsymbol{\sigma}$ should specify `RJMCMC<SpecM, coretools::probdist::TBetaDistr>`. Note that the model parameter $\boldsymbol{m}$ does not need the RJ-MCMC specification in `ParamSpec`.
The function `calculateLLRatio(SpecSigma)` for $\boldsymbol{\sigma}$ is then used to calculate the likelihood ratio of the standard Metropolis-Hastings update in case of no jump between submodels.
The function `calculateLLRatio(SpecM)` for $\boldsymbol{m}$ is used to calculate the likelihood ratio of the jump between models. The user needs to figure out which model jump was performed. The jump $\mathcal{M}_2 \to \mathcal{M}_1$ would result in `m->value(i) == 0`, meaning that the likelihood ratio $\frac{\mathbb{P}(\mathcal{D}|\boldsymbol{\theta})}{\mathbb{P}(\mathcal{D}|\boldsymbol{\theta},\boldsymbol{\sigma})}$ must be calculated. Analogously, the jump $\mathcal{M}_1 0 \to \mathcal{M}_2$ would result in `m->value(i) == 1`, meaning that the likelihood ratio $\frac{\mathbb{P}(\mathcal{D}|\boldsymbol{\theta},\boldsymbol{\sigma})}{\mathbb{P}(\mathcal{D}|\boldsymbol{\theta})}$ must be calculated. The other parts of the RJ-MCMC Hastings ratio are evaluated internally in `stattools`.

1.5 Prior distributions

In a Bayesian setting, every parameter is assumed to follow a prior distribution. The prior distribution reflects the knowledge about the parameter before any observations are taken into account. The prior can be an uniform distribution, giving equal weight to every value, or something more sophisticated like a normal, exponential or beta distribution. The latter distributions are parametrized with prior parameters (e.g. mean and variance for the normal distribution).
The relationship between an observation and a parameter is formally called the likelihood. However, from a programming point of view, the implementation of likelihood and prior is the same, as they both define a probabilistic relationship. The following classes, although named "prior", equally apply to likelihood functions if they relate an observation to a parameter.
In `stattools`, the prior distributions are implemented in the classes deriving from `prior::TBase`. A couple of prior classes have fixed prior parameters that are not inferred. These all include the name "Fixed". Other prior classes infer their prior parameters (and are hence called "Inferred"). For efficient calculations, the prior classes must know 1) the type and the number of dimensions of the parameter they are defined on, and 2) the full parameter specification (`ParamSpec`) of all prior parameters. The number of templates is thus specific for each prior class. There are certain restrictions on these parameter types, since most priors are only valid on specific intervals. For example, for `prior::TNormalInferred` mentioned before, `Type` and `SpecMean::value_type` must be unconstrained floating point numbers (constraints are not allowed, since this would represent a truncated normal prior that needs to normalize its densities). `SpecVar::value_type` on the other hand must be a strictly positive floating point number. To enforce these conditions, the prior classes are all templated with static asserts, such that compile errors result if the conditions are not met.

1.5.0.1 Initialization

Prior classes that have inferred prior parameters expect pointers for each such parameter in the constructor. Priors with fixed prior parameters accept a string that contains the values for the prior parameters (seperated by commas). This is handy since users are allowed to change these values over the command-line or via files.

A prior that manages inferred prior parameters usually has certain expectations on the dimensionality of these prior parameters. For instance, a normal prior knows that the parameter for the mean must be a single element, whereas a multivariate normal prior knows that the parameter for the mean must be D-dimensional. The function `initialize()` of a prior goes over all parameters on which the prior is defined and extracts the necessary information about their dimensions (e.g. the multivariate normal prior will derive the number of dimensions D from the parameter below). The function then calls `initStorage()` of each prior parameter, and passes a vector of dimensions, for example `{D}` to it. The parameter then allocates the requested memory. An important member of class `TPriorBase` is a vector of pointers to the data of the parameters and observations it is defined on (member variable `_storageBelow`). Usually, this vector will have only one element, as most parameters have their own prior. However, there might be cases where multiple parameters/observations share the same prior and where the prior parameters should be leared based on all of these parameters/observations.

1.5.1 Prior ratios

An important part of a prior is the calculation of prior probabilities. The Hastings ratio for an update of parameter $x \to x'$ is calculated as

$$h_x = \min\left(1, \frac{\mathbb{P}(\boldsymbol{\mathcal{D}}|x')\mathbb{P}(x')q(x', x)}{\mathbb{P}(\boldsymbol{\mathcal{D}}|x)\mathbb{P}(x)q(x, x')}\right).$$

Here, $\mathbb{P}(\boldsymbol{\mathcal{D}}|x)$ is the likelihood, $\mathbb{P}(x)$ is the prior probability, and $q(x, x')$ is the proposal kernel. The prior ratio p_x, which is the focus of this section, is therefore part of the full Hastings ratio and defined as:

$$p_x = \frac{\mathbb{P}(x')}{\mathbb{P}(x)}.$$

As these probabilities can become very small, we usually calculate the log prior ratio:

$$\log p_x = \log \mathbb{P}(x') - \log \mathbb{P}(x).$$

Here, the prior density $\mathbb{P}(x)$ obviously depends on the prior distribution chosen. The interface to get $\log \mathbb{P}(x')$, $\log \mathbb{P}(x)$ and $\log p_x$ however is the same for all prior distributions:

- `getDensity(const Storage & x, size_t i)`: Calculates $\mathbb{P}(x'_i)$.
- `getLogDensity(const Storage & x, size_t i)`: Calculates $\log \mathbb{P}(x'_i)$.
- `getLogDensityRatio(const UpdatedStorage & x, size_t i)`: Calculates $\log p_{x_i}$.

The functions `getDensity()` and `getLogDensityRatio()` are pure virtual and must be overridden by any prior inheriting from `TPriorBase`. The function `getLogDensity()` by default just calculates `log(getDensity())`, but can also be overridden if necessary.
Multiple classes inherit from `TPriorBase` and implement specific prior distributions. We will now discuss those in more detail.

1.5.2 Priors with fixed parameters

The following paragraphs describe prior distributions where the prior parameters are fixed, i.e. they are not inferred in the MCMC.

1.5.2.1 `prior::TUniformFixed`

A uniform prior does not have any parameters, except for a minimal and maximal value for which the log density is defined. These values are directly obtained from the underlying type. The log prior density of a value x with minimum a and maximum b is:

$$\log \mathbb{P}(x|a,b) = \log\left(\frac{1}{b-a}\right).$$

The log prior ratio is zero, as every value is equally likely:

$$\log p_x = \log\left(\frac{\mathbb{P}(x'|a,b)}{\mathbb{P}(x|a,b)}\right) = 0.$$

1.5.2.2 `prior::TNormalFixed`

A normal distribution is parametrized by:

- mean μ (`_mean`).
- variance σ^2 (`_var`), where $\sigma^2 > 0$.

The log prior density of a value x is:

$$\log \mathbb{P}(x|\mu,\sigma^2) = -\frac{1}{2}\log(\sigma^2) - \frac{1}{2}\log(2\pi) - \frac{1}{2\sigma^2}(x-\mu)^2. \tag{10}$$

The log prior ratio is:

$$\log p_x = \log\left(\frac{\mathbb{P}(x'|\mu,\sigma^2)}{\mathbb{P}(x|\mu,\sigma^2)}\right) = \frac{1}{2\sigma^2}\left((x-\mu)^2 - (x'-\mu)^2\right). \tag{11}$$

1.5.2.3 `prior::TExponentialFixed`

An exponential distribution is parametrized by:

- rate λ (`_lambda`), where $\lambda > 0$.

The log prior density of a value x is:

$$\log \mathbb{P}(x|\lambda) = \log(\lambda) - \lambda x. \tag{12}$$

The log prior ratio is:

$$\log p_x = \log\left(\frac{\mathbb{P}(x'|\lambda)}{\mathbb{P}(x|\lambda)}\right) = \lambda(x - x'). \tag{13}$$

1.5.2.4 `prior::TGammaFixed`

An gamma distribution is parametrized by:

- shape α (`_alpha`), where $\alpha > 0$.
- rate β (`_beta`), where $\beta > 0$.

The log prior density of a value x is:

$$\log \mathbb{P}(x|\alpha,\beta) = \alpha\log(\beta) - \log\Gamma(\alpha) + (\alpha-1)\log(x) - \beta x. \tag{14}$$

The log prior ratio is:

$$\log p_x = \log\left(\frac{\mathbb{P}(x'|\alpha,\beta)}{\mathbb{P}(x|\alpha,\beta)}\right) = (\alpha-1)\log\left(\frac{x'}{x}\right) + \beta(x - x'). \tag{15}$$

1.5.2.5 `prior::TChisqFixed`

A chi-square distribution is parametrized by

- the degrees of freedom k (`_k`), where $k > 0$.

The log prior density of a value x is:

$$\log \mathbb{P}(x|k) = -\frac{k}{2}\log 2 - \ln \Gamma\left(\frac{k}{2}\right) + \left(\frac{k}{2} - 1\right)\log x - \frac{x}{2}. \tag{16}$$

The log prior ratio is:

$$\log p_x = \log\left(\frac{\mathbb{P}(x'|k)}{\mathbb{P}(x|k)}\right) = \left(\frac{k}{2} - 1\right)(\log x' - \log x) + \frac{x - x'}{2}. \tag{17}$$

1.5.2.6 `prior::TMultivariateChisqFixed`

In this prior, $\boldsymbol{x}$ is a vector of N values, and on every element x_i a chisq prior with a different degree of freedom k_i is specified. A classic example is that the degrees of freedom are given by the index i with $N + 1 - i$, such that the first element in the vector has N degrees of freedom, the second has $N - 1$, ..., and the last has 1 degree of freedom.
Hence, this prior is parametrized by

- A vector of degrees of freedom, $\boldsymbol{k}$ (`_k`), where all $k_i > 0$.

The log prior density and ratio of a value x are given by (16) and (17), with the only difference that k depends on the index i.

1.5.2.7 `prior::TMultivariateChiFixed`

In this prior, $\boldsymbol{x}$ is a vector of N values, and on every element x_i a chi prior with a different degree of freedom k_i is specified. A classic example is that the degrees of freedom are given by the index i with $N + 1 - i$, such that the first element in the vector has N degrees of freedom, the second has $N - 1$, ..., and the last has 1 degree of freedom.
Hence, this prior is parametrized by

- A vector of degrees of freedom, $\boldsymbol{k}$ (`_k`), where all $k_i > 0$.

The log prior density of a value x is:

$$\log \mathbb{P}(x|k) = -\left(\frac{k}{2} - 1\right)\log 2 - \ln \Gamma\left(\frac{k}{2}\right) + (k-1)\log x - \frac{x^2}{2}.$$

The log prior ratio is:

$$\log p_x = \log\left(\frac{\mathbb{P}(x'|k)}{\mathbb{P}(x|k)}\right) = (k-1)(\log x' - \log x) + \frac{x^2 - x'^2}{2}.$$

1.5.2.8 `prior::TBetaFixed`

A beta distribution is parametrized by

- shape α (`_alpha`), where $\alpha > 0$.
- shape β (`_beta`), where $\beta > 0$.

The log prior density of a value x is:

$$\log \mathbb{P}(x|\alpha,\beta) = \ln\Gamma(\alpha+\beta) - \ln\Gamma(\alpha) - \ln\Gamma(\beta) + (\alpha-1)\log x + (\beta-1)\log(1-x), \tag{18}$$

where $\ln\Gamma$ is the log-gamma function. The log prior ratio is:

$$\log p_x = \log\left(\frac{\mathbb{P}(x'|\alpha,\beta)}{\mathbb{P}(x|\alpha,\beta)}\right) = (\alpha-1)(\log x' - \log x) + (\beta-1)(\log(1-x') - \log(1-x)). \tag{19}$$

1.5.2.9 `prior::TBinomialFixed`

A binomial distribution is parametrized by

- the success probability for each trial, p (`_p`), where $0 < p < 1$.

The data on which a binomial prior is defined must be 2-dimensional, as we allow for different number of trials n. The function `setFixedPriorParameters()` checks if 1) the data is one-dimensional with a total size of 2 (n and k) or if 2) the data is two-dimensional with the number of columns (the second dimension) being exactly two (n and k). If neither of these is the case, an error is thrown.
The log prior density of a value k is:

$$\log \mathbb{P}(k|n,p) = \log\binom{n}{k} + k\log p + (n-k)\log(1-p). \tag{20}$$

The log prior ratio is:

$$\log p_k = \log\left(\frac{\mathbb{P}(k'|n,p)}{\mathbb{P}(k|n,p)}\right) = \log\binom{n}{k'} - \log\binom{n}{k} + (k'-k)\log p + (k-k')\log(1-p). \tag{21}$$

1.5.2.10 `prior::TNegativeBinomialFixed`

The negative binomial distribution models the number of failures which occur in a sequence of Bernoulli trials before a target number of successes is reached. Note that there exist different formulations for this distribution, as failure and success are interchangable. We implemented the formulation that is used in `R` and thus assume that the number of successes n until the experiment is stopped is non-random (observed), while the number of failures x is a parameter that can potentially be updated, analogous to k from the binomial distribution.
An negative binomial distribution is parametrized by:

- the success probability for each trial, p (`_p`), where $0 < p < 1$.

The data on which a negative binomial prior is defined must be 2-dimensional, as we allow for different number of trials n. The function `setFixedPriorParameters()` checks if 1) the data is one-dimensional with a total size of 2 (n and x) or if 2) the data is two-dimensional with the number of columns (the second dimension) being exactly two (n and x). If neither of these is the case, an error is thrown.
The log prior density of a value x is:

$$\log \mathbb{P}(x|p) = \log\Gamma(x+n) - \log\Gamma(n) - \log(x!) + x\log(1-p) + n\log(p) \tag{22}$$

The log prior ratio is:

$$\log p_x = \log\left(\frac{\mathbb{P}(x'|p)}{\mathbb{P}(x|p)}\right) = \log\Gamma(x'+n) - \log\Gamma(x+n) + \log(x!) - \log(x'!) + (x'-x)\log(1-p); \tag{23}$$

Here, the term $\log(x!) - \log(x'!)$ can be simplified to a falling factorial $\sum_{k=0}^{n-1}(y-k)$ with $n = |x'-x|$ and $y = \max(x', x)$. This term is added if $x > x'$ and substracted if $x \leq x'$.

1.5.2.11 `prior::TBernoulliFixed`

A Bernoulli distribution parametrized by

- probability π (`_pi`), where $0 < \pi < 1$.

The log prior density of a value x is:

$$\log \mathbb{P}(x|\pi) = x \log \pi + (1 - x) \log(1 - \pi). \tag{24}$$

The log prior ratio is:

$$\log p_x = \log \left(\frac{\mathbb{P}(x'|\pi)}{\mathbb{P}(x|\pi)} \right) = (x' - x) \log \pi + (x - x') \log(1 - \pi). \tag{25}$$

1.5.2.12 `prior::TPoissonFixed`

A Poisson distribution parametrized by

- rate λ (`_lambda`), where $\lambda > 0$.

The log prior density of a value x is:

$$\log \mathbb{P}(x|\lambda) = x \log(\lambda) - \lambda - \log(x!). \tag{26}$$

The log prior ratio is:

$$\log p_x = \log \left(\frac{\mathbb{P}(x'|\lambda)}{\mathbb{P}(x|\lambda)} \right) = (x' - x) \log(\lambda) + \log(x!) - \log(x'!). \tag{27}$$

Here, the term $\log(x!) - \log(x'!)$ can be simplified to a falling factorial $\sum_{k=0}^{n-1}(y-k)$ with $n = |x'-x|$ and $y = \max(x', x)$. This term is added if $x > x'$ and substracted if $x \leq x'$.

1.5.2.13 `prior::TDirichletFixed`

A Dirichlet distribution parametrized by

- a vector of K concentration parameters $\boldsymbol{\alpha}$, where all $\alpha_k > 0$.

The log prior density of values $\boldsymbol{x}$ is:

$$\log \mathbb{P}(\boldsymbol{x}|\boldsymbol{\alpha}) = -\log \mathrm{B}(\boldsymbol{\alpha}) + \sum_{k=1}^{K} (\alpha_k - 1) \log x_k, \tag{28}$$

where the normalizing constant $\mathrm{B}(\boldsymbol{\alpha})$ is a Beta function given by

$$\mathrm{B}(\boldsymbol{\alpha}) = \frac{\prod_{k=1}^{K} \Gamma(\alpha_k)}{\Gamma(\sum_{k=1}^{K} \alpha_k)}.$$

The log prior ratio is:

$$\log p_{\boldsymbol{x}} = \log \left(\frac{\mathbb{P}(\boldsymbol{x}'|\boldsymbol{\alpha})}{\mathbb{P}(\boldsymbol{x}|\boldsymbol{\alpha})} \right) = \sum_{k=1}^{K} (\alpha_k - 1) \log \left(\frac{x'_k}{x_k} \right). \tag{29}$$

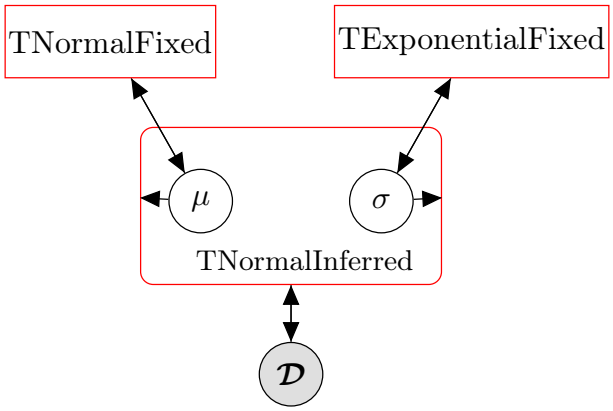

Figure 4: *Example DAG. The prior on data $\mathcal{D}$ is a normal distribution where the parameters μ and σ are inferred. $\mathcal{D}$ is an instance of* `TObservation`*, and μ and σ are instances of* `TParameter`*. The prior distributions on μ and σ are fixed, since the prior parameters of these distributions can not be estimated from a single value. All existing pointers are shown with directed edges.*

1.5.3 Priors with inferred prior parameters

The following paragraphs describe prior distributions where the prior parameters are inferred. In inferred prior classes, the prior parameters are pointers to `TParameter`. A simple example is shown in Figure 4, where a `TObservation` $\boldsymbol{\mathcal{D}}$ has a pointer to a `prior::TNormalInferred`. This `prior::TNormalInferred` has a pointer to both μ and σ, which are of type `TParameter`. Each prior parameter, i.e. μ and σ, has a pointer to a fixed prior distribution. In addition, each prior parameters also has a pointer to the box around them, i.e. to `prior::TNormalInferred`.
While in most cases a prior is solely defined on one parameter or observation $\boldsymbol{x}$, it is possible that it is defined on a vector of P parameters. In such cases, we need to account for all parameters $\boldsymbol{x}_p$ when updating the prior parameters. In the following paragraphs, we will write out the likelihood ratios for the single-parameter case only. The likelihood ratio for a prior parameter θ naturally extends to

$$l_\theta = \sum_{p=1}^{P} \log\left(\frac{\mathbb{P}(\boldsymbol{x}_p|\theta')}{\mathbb{P}(\boldsymbol{x}_p|\theta)}\right).$$

for the multi-parameter case. The loop over all nodes below is hidden in `prior::TPriorBase` and must not be considered by the developer.

1.5.3.1 `prior::TNormalInferred`

A normal distribution is parametrized by

- mean μ (`_mean`).
- variance σ^2 (`_var`).

The log prior density and ratio are given by (10) and (11), respectively.
The log likelihood ratio l_μ for an update of μ is:

$$l_\mu = \log\left(\frac{\mathbb{P}(\boldsymbol{x}|\mu', \sigma^2)}{\mathbb{P}(\boldsymbol{x}|\mu, \sigma^2)}\right) = \frac{1}{2\sigma^2}\sum_{i=1}^{N}\left((x_i-\mu)^2 - (x_i-\mu')^2\right).$$

The log likelihood ratio l_{σ^2} for an update of σ^2 is:

$$l_{\sigma^2} = \log\left(\frac{\mathbb{P}(\boldsymbol{x}|\mu,\sigma^{2\prime})}{\mathbb{P}(\boldsymbol{x}|\mu,\sigma^2)}\right) = \frac{N}{2}\log\left(\frac{\sigma^2}{\sigma^{2\prime}}\right) + \frac{1}{2}\left(\frac{1}{\sigma^2} - \frac{1}{\sigma^{2\prime}}\right)\sum_{i=1}^{N}(x_i-\mu)^2.$$

For initialization of the prior parameters, we calculate the maximum likelihood estimate (MLE) μ and σ^2. The MLE $\hat{\mu}$ is

$$\hat{\mu} = \frac{1}{N}\sum_{i=1}^{N} x_i. \tag{30}$$

If the prior is defined on a vector of P parameters $\boldsymbol{X}$, the MLE $\hat{\mu}$ is:

$$\hat{\mu} = \frac{1}{\sum_{p=1}^{P} N_p}\sum_{p=1}^{P}\sum_{i=1}^{N_p} x_{p_i}.$$

The MLE $\hat{\sigma^2}$ is

$$\hat{\sigma^2} = \frac{1}{N}\sum_{i=1}^{N}(x_i-\hat{\mu})^2. \tag{31}$$

If the prior is defined on a vector of P parameters $\boldsymbol{X}$, the MLE $\hat{\sigma^2}$ is:

$$\hat{\sigma^2} = \frac{1}{\sum_{p=1}^{P} N_p}\sum_{p=1}^{P}\sum_{i=1}^{N_p}(x_{p_i}-\hat{\mu})^2.$$

1.5.3.2 `prior::TExponentialInferred`

An exponential distribution is parametrized by

- rate λ (`_lambda`).

The log prior density and ratio are given by (12) and (13), respectively.
The log likelihood ratio l_λ for an update of λ is:

$$l_\lambda = \log\left(\frac{\mathbb{P}(\boldsymbol{x}|\lambda')}{\mathbb{P}(\boldsymbol{x}|\lambda)}\right) = N\log\left(\frac{\lambda'}{\lambda}\right) + (\lambda-\lambda')\sum_{i=1}^{N} x_i.$$

For the initialization of the prior parameters, we calculate the MLE for $\hat{\lambda}$:

$$\hat{\lambda} = \frac{N}{\sum_{i=1}^{N} x_i},$$

If the prior is defined on a vector of parameters, the MLE is:

$$\hat{\lambda} = \frac{\sum_{p=1}^{P} N_p}{\sum_{p=1}^{P}\sum_{i=1}^{N_p} x_{p_i}}.$$

1.5.3.3 `prior::TPoissonInferred`

An Poisson distribution is parametrized by

- rate λ (`_lambda`).

The log prior density and ratio are given by (26) and (27), respectively.
The log likelihood ratio l_λ for an update of λ is:

$$l_\lambda = \log\left(\frac{\mathbb{P}(\boldsymbol{x}|\lambda')}{\mathbb{P}(\boldsymbol{x}|\lambda)}\right) = \log\left(\frac{\lambda'}{\lambda}\right)\sum_{i=1}^{N} x_i + (\lambda - \lambda')N.$$

For the initialization of the prior parameters, we calculate the MLE for $\hat{\lambda}$:

$$\hat{\lambda} = \frac{\sum_{i=1}^{N} x_i}{N},$$

If the prior is defined on a vector of parameters, the MLE is:

$$\hat{\lambda} = \frac{\sum_{p=1}^{P}\sum_{i=1}^{N_p} x_{p_i}}{\sum_{p=1}^{P} N_p}.$$

1.5.3.4 `prior::TBetaInferred`

A beta distribution is parametrized by

- shape α (`_alpha`).
- shape β (`_beta`).

The log prior density and ratio are given by (18) and (19), respectively.
The log likelihood ratio l_α for an update of α is

$$l_\alpha = \log\left(\frac{\mathbb{P}(\boldsymbol{x}|\alpha',\beta)}{\mathbb{P}(\boldsymbol{x}|\alpha,\beta)}\right) = N\left(\ln\Gamma(\alpha'+\beta) - \ln\Gamma(\alpha+\beta) + \ln\Gamma(\alpha) - \ln\Gamma(\alpha')\right) + (\alpha'-\alpha)\sum_{i=1}^{N}\log x_i.$$

The log likelihood ratio l_β for an update of β is

$$l_\beta = \log\left(\frac{\mathbb{P}(\boldsymbol{x}|\alpha,\beta')}{\mathbb{P}(\boldsymbol{x}|\alpha,\beta)}\right) = N\left(\ln\Gamma(\alpha+\beta') - \ln\Gamma(\alpha+\beta) + \ln\Gamma(\beta) - \ln\Gamma(\beta')\right) + (\beta'-\beta)\sum_{i=1}^{N}\log(1-x_i).$$

For the initialization of the prior parameters, we calculate the method of moments estimates for α and β, since the MLE does not have a closed form. These estimates rely on the mean μ and variance σ^2 of $\boldsymbol{x}$:

$$\begin{aligned}\mu &= \frac{1}{N}\sum_{i=1}^{N} x_i,\\ \sigma^2 &= \frac{1}{N}\sum_{i=1}^{N}(x_i-\mu)^2.\end{aligned}$$

Let us define

$$\nu = \hat{\alpha} + \hat{\beta} = \frac{\mu(1-\mu)}{\sigma^2} - 1, \text{ where } \nu > 0, \text{ therefore: } \sigma^2 < \mu(1-\mu).$$

The method of moments estimates for $\hat{\alpha}$ and $\hat{\beta}$ then are:

$$\begin{aligned}\hat{\alpha} &= \mu\nu,\\ \hat{\beta} &= (1-\mu)\nu.\end{aligned}$$

This naturally extends to the case where the prior is defined on a vector of parameters, as the only difference is the equation for μ and σ^2:

$$\begin{aligned}\mu &= \frac{1}{\sum_{p=1}^{P} N_p}\sum_{p=1}^{P}\sum_{i=1}^{N_p} x_{p_i},\\ \sigma^2 &= \frac{1}{\sum_{p=1}^{P} N_p}\sum_{p=1}^{P}\sum_{i=1}^{N_p}(x_{p_i}-\mu)^2.\end{aligned}$$

1.5.3.5 `prior::TGammaInferred`

A gamma distribution is parametrized by:

- shape α (`_alpha`).
- rate β (`_beta`).

The log prior density and ratio are given by (14) and (15), respectively.
The log likelihood ratio l_α for an update of α is:

$$l_\alpha = \log\left(\frac{\mathbb{P}(\boldsymbol{x}|\alpha',\beta)}{\mathbb{P}(\boldsymbol{x}|\alpha,\beta)}\right) = N\left(\log\beta(\alpha'-\alpha)+\log\Gamma(\alpha)-\log\Gamma(\alpha')\right)+(\alpha'-\alpha)\sum_{i=1}^{N}\log x_i.$$

The log likelihood ratio l_β for an update of β is:

$$l_\beta = \log\left(\frac{\mathbb{P}(\boldsymbol{x}|\alpha,\beta')}{\mathbb{P}(\boldsymbol{x}|\alpha,\beta)}\right) = N\alpha\log\left(\frac{\beta'}{\beta}\right)+(\beta-\beta')\sum_{i=1}^{N} x_i.$$

There is no closed-form solution for the MLE of α and β. Instead, we use mixed type log-moment estimators that have a closed-form and are derived from the likelihood of the generalized gamma distribution.
Let us denote by $s = N\sum_{i=1}^{N} x_i\log x_i - \sum_{i=1}^{N}\log x_i\sum_{i=1}^{N} x_i$. The estimate for α is:

$$\hat{\alpha} = \frac{N\sum_{i=1}^{N} x_i}{s},$$

If the prior is defined on a vector of parameters, the estimate is:

$$\hat{\alpha} = \frac{\sum_{p=1}^{P} N_p\sum_{i=1}^{N_p} x_i}{\sum_{p=1}^{P} s_p},$$

The estimate for β is:

$$\hat{\beta} = \frac{N^2}{s},$$

If the prior is defined on a vector of parameters, the estimate is:

$$\hat{\beta} = \frac{\sum_{p=1}^{P} N_p^2}{\sum_{p=1}^{P} s_p}.$$

1.5.3.6 `prior::TBinomialInferred`

A Binomial distribution is parametrized by

- the success probability for each trial, p (`_p`).

The data this prior is defined on must be 2-dimensional, where the number of columns (i.e. the second dimension) is of length two, corresponding to n_i and k_i of the binomial distribution.
The log prior density and ratio are given by (20) and (21), respectively.
The log likelihood ratio l_p for an update of p is:

$$l_p = \log\left(\frac{\mathbb{P}(\boldsymbol{k}|\boldsymbol{n}, p')}{\mathbb{P}(\boldsymbol{k}|\boldsymbol{n}, p)}\right) = \log\left(\frac{p'}{p}\right)\sum_{i=1}^{N} k_i + \log\left(\frac{1-p'}{1-p}\right)\sum_{i=1}^{N}(n_i - k_i)\,.$$

For the initialization of the prior parameters, we calculate the MLE for p with

$$\hat{p} = \frac{\sum_{i=1}^{N} k_i}{\sum_{i=1}^{N} n_i}.$$

If the prior is defined on a vector of parameters, this gets:

$$\hat{p} = \frac{\sum_{p=1}^{P}\sum_{i=1}^{N_p} x_{pi}}{\sum_{p=1}^{P}\sum_{i=1}^{N} n_{pi}}.$$

1.5.3.7 `prior::TNegativeBinomialInferred`

A negative binomial distribution is parametrized by

- the success probability for each trial, p (`_p`).

The same conditions apply to the data as in the fixed prior, see above.
The log prior density and ratio are given by (22) and (23), respectively.
The log likelihood ratio l_p for an update of p is:

$$l_p = \log\left(\frac{\mathbb{P}(\boldsymbol{x}|\boldsymbol{n}, p')}{\mathbb{P}(\boldsymbol{x}|\boldsymbol{n}, p)}\right) = \log\left(\frac{p'}{p}\right)\sum_{i=1}^{N} n_i + \log\left(\frac{1-p'}{1-p}\right)\sum_{i=1}^{N} x_i.$$

For the initialization of the prior parameters, we calculate the MLE for p with

$$\hat{p} = \frac{\sum_{i=1}^{N} n_i}{\sum_{i=1}^{N}(n_i + x_i)}.$$

If the prior is defined on a vector of parameters, this gets:

$$\hat{p} = \frac{\sum_{p=1}^{P}\sum_{i=1}^{N_p} n_{pi}}{\sum_{p=1}^{P}\sum_{i=1}^{N}(n_{pi} + x_{pi})}.$$

1.5.3.8 `prior::TBernoulliInferred`

A Bernoulli distribution is parametrized by

- a probability π (`_pi`).

The log prior density and ratio are given by (24) and (25), respectively.
The log likelihood ratio l_π for an update of π is:

$$l_\pi = \log\left(\frac{\mathbb{P}(\boldsymbol{x}|\pi')}{\mathbb{P}(\boldsymbol{x}|\pi)}\right) = \log\left(\frac{\pi'}{\pi}\right)\sum_{i=1}^{N} x_i + \log\left(\frac{1-\pi'}{1-\pi}\right)\sum_{i=1}^{N}(1-x_i).$$

For the initialization of the prior parameters, we calculate the MLE for π with

$$\hat{\pi} = \frac{\sum_{i=1}^{N} x_i}{N}.$$

If the prior is defined on a vector of parameters, this gets:

$$\hat{\pi} = \frac{\sum_{p=1}^{P}\sum_{i=1}^{N_p} x_{p_i}}{\sum_{p=1}^{P} N_p}.$$

1.5.3.9 `prior::TGibbsBetaBernoulliInferred`

If the likelihood is given by a Bernoulli distribution as

$$\mathbb{P}(x|\pi) = \pi^x(1-\pi)^{1-x},$$

and the prior is given by a Beta distribution as

$$\mathbb{P}(\pi|\alpha,\beta) = \pi^{\alpha-1}(1-\pi)^{\beta-1}\frac{1}{\mathrm{B}(\alpha,\beta)},$$

then it is possible to sample directly from the posterior $\mathbb{P}(\pi|x)$, since the Bernoulli and the Beta distribution are conjugate. The posterior is proportional to

$$\mathbb{P}(\pi|x) \propto \pi^{k+\alpha-1}(1-\pi)^{N-k+\beta-1},$$

where $k = \sum_{i=1}^{N} x_i$ denotes the number of successes. This is again a Beta-distribution, such that

$$\mathbb{P}(\pi|x) \sim \mathrm{B}(k+\alpha, N-k+\beta).$$

In such cases, the parameter π does not need to be updated with the Metropolis-Hastings algorithm, but can be instead directly sampled from its posterior distribution. This is more efficient, since every update is accepted (compared to accepting only roughly 30% of all updates in Metropolis-Hastings).
This prior inherits from `prior::TBernoulliInferred` and overrides solely the methods that update π: Instead of updating π in Metropolis-Hastings, the function `doGibbs()` samples from the posterior. Because it needs to know α and β from the prior on π, the class additionally expects a pointer to the `prior::TBetaFixed` prior on π.

1.5.3.10 `prior::TCategoricalInferred`

A categorical distribution is a discrete probability distribution that describes the possible results of a parameter $\boldsymbol{x}$ that can take on one of K possible categories, with the probability of each category separately specified by a probability π_k. This prior is thus parametrized by

- probabilities $\boldsymbol{\pi}$ (`_pis`) of size K.

The probabilities $\boldsymbol{\pi}$ must sum to one: $\sum_k \pi_k = 1$. The initial values are thus normalized to satisfy this condition.
The log prior density of a value x_i is:

$$\log \mathbb{P}(x_i = k|\boldsymbol{\pi}) = \pi_k.$$

The log prior ratio is

$$\log p_x = \log\left(\frac{\mathbb{P}(x = j|\boldsymbol{\pi})}{\mathbb{P}(x = k|\boldsymbol{\pi})}\right) = \log\left(\frac{\pi_j}{\pi_k}\right).$$

The log likelihood ratio l_{π_k} for an update of π_k is:

$$l_{\pi_k} = \log\left(\frac{\mathbb{P}(\boldsymbol{x}|\pi'_k)}{\mathbb{P}(\boldsymbol{x}|\pi_k)}\right) = \log\left(\frac{\pi'_k}{\pi_k}\right)\sum_{i=1}^{N}\mathcal{I}(x_i = k).$$

For the initialization of the prior parameters, we calculate the MLE for each $\hat{\pi}_k$:

$$\hat{\pi}_k = \frac{1}{N}\sum_{i=1}^{N}\mathcal{I}(x_i = k).$$

If the prior is defined on a vector of parameters, the MLE is:

$$\hat{\pi} = \frac{1}{\sum_{p=1}^{P} N_p}\sum_{p=1}^{P}\sum_{i=1}^{N_p}\mathcal{I}(x_{p_i} = k).$$

1.5.3.11 `prior::TDirichletVarInferred`

A Dirichlet with inferred variance distribution parametrized by

- a vector of K concentration parameters $\tilde{\boldsymbol{\alpha}}$ (fix).
- a variance parameter σ (inferred).

The values of $\tilde{\boldsymbol{\alpha}}$ should be given as a comma-seperated string to the function `setFixedPrior-Parameters()`, whereas σ is a pointer to `TParameter`. The concentration parameters that are then used for the Dirichlet distribution are given by

$$\boldsymbol{\alpha} = \frac{\tilde{\boldsymbol{\alpha}}}{\sigma}.$$

The idea of this class is that the expected value of the parameter below does not depend on σ but only on the normalized concentration parameter:

$$\mathbb{E}[X_i] = \frac{\alpha_i}{\sum_{k=1}^{K}\alpha_k}.$$

However, the variance of the parameter below may vary depending on σ:

$$\mathrm{Var}[X_i] = \frac{a_i(1-a_i)}{\alpha_0+1},$$

where $a_i = \frac{\alpha_i}{\alpha_0}$ and $\alpha_0 = \sum_{k=1}^{K} \alpha_k$. The forumlation is such that smaller values of σ result in smaller $\mathrm{Var}[X_i]$.
The log prior density and ratio are given by 28 and 29 as described above for `TDirichletFixed`. The log likelihood ratio l_σ for an update of σ is:

$$l_\sigma = \log\left(\frac{\mathbb{P}(\boldsymbol{x}|\boldsymbol{\alpha}')}{\mathbb{P}(\boldsymbol{x}|\boldsymbol{\alpha})}\right) = \log \mathrm{B}(\boldsymbol{\alpha}) - \log \mathrm{B}(\boldsymbol{\alpha}') + \sum_{k=1}^{K}(\alpha'_k - \alpha_k)\log x_k.$$

For the initialization of σ prior to the MCMC, we do a line search to find the value of σ maximizing the log-likelihood as given by equation 28.

1.5.3.12 `prior::THMMPrior`

The general framework for HMMs are described in section 3. Here, we will describe how the HMM parameters are updated in the MCMC. The class `prior::THMMPrior` serves as a base class for several specific HMM priors. As the MCMC updates are very similar among all HMM priors, we will describe the general procedure here assuming that the transition probabilities are modelled using a parameter θ and that the hidden states are represented by z_i with $i = 0, \ldots, N$. The distances are given by δ_i.
The HMM prior is a classic example of a non-iid prior, as the prior density of one z_i depends on its neighbours z_{i-1} and z_{i+1}. The log prior density of a value z_i is

$$\log \mathbb{P}(z_i|\boldsymbol{z}_{0:N}, \boldsymbol{\delta}_{0:N}, \theta) = \log \mathbb{P}(z_i|z_{i-1}, \delta_i, \theta) + \log \mathbb{P}(z_{i+1}|z_i, \delta_{i+1}, \theta).$$

These probabilites correspond to the transition probabilities and are given by the element $[Q(\delta_i)]_{z_{i-1}z_i}$ from the transition matrix. Note that the log prior density breaks down to

$$\log \mathbb{P}(z_0|\boldsymbol{z}_{0:N}, \boldsymbol{\delta}_{0:N}, \theta) = \log \mathbb{P}(z_0|\theta) + \log \mathbb{P}(z_1|z_0, \delta_1, \theta),$$

and

$$\log \mathbb{P}(z_N|\boldsymbol{z}_{0:N}, \boldsymbol{\delta}_{0:N}, \theta) = \log \mathbb{P}(z_N|z_{N-1}, \delta_N, \theta),$$

for $z = 0$ and $z = N$, respectively.
The log prior ratio for an update of z_i is:

$$\begin{aligned}\log p_{z_i} &= \log\left(\frac{\mathbb{P}(z'_i|\boldsymbol{z}_{0:N}, \boldsymbol{\delta}_{0:N}, \theta)}{\mathbb{P}(z_i|\boldsymbol{z}_{0:N}, \boldsymbol{\delta}_{0:N}, \theta)}\right) \\ &= \log\left(\frac{\mathbb{P}(z'_i|z_{i-1}, \delta_i, \theta)}{\mathbb{P}(z_i|z_{i-1}, \delta_i, \theta)}\right) + \log\left(\frac{\mathbb{P}(z_{i+1}|z'_i, \delta_{i+1}, \theta)}{\mathbb{P}(z_{i+1}|z_i, \delta_{i+1}, \theta)}\right),\end{aligned}$$

Note that the log prior ratio breaks down to

$$\log p_{z_0} = \log\left(\frac{\mathbb{P}(z'_0|\theta)}{\mathbb{P}(z_0|\theta)}\right) + \log\left(\frac{\mathbb{P}(z_1|z'_0, \delta_1, \theta)}{\mathbb{P}(z_1|z_0, \delta_1, \theta)}\right),$$

and

$$\log p_{z_N} = \log\left(\frac{\mathbb{P}(z'_N|z_{N-1}, \delta_N, \theta)}{\mathbb{P}(z_N|z_{N-1}, \delta_N, \theta)}\right),$$

for $z = 0$ and $z = N$, respectively.

The log likelihood ratio l_θ for an update of θ is

$$l_\theta = \log\left(\frac{\mathbb{P}(\boldsymbol{z}_{0:N}|\boldsymbol{\delta}_{0:N},\theta')}{\mathbb{P}(\boldsymbol{z}_{0:N}|\boldsymbol{\delta}_{0:N},\theta)}\right) = \log\left(\frac{\mathbb{P}(z_0|\theta')}{\mathbb{P}(z_0|\theta)}\right) + \sum_{i=1}^{N}\log\left(\frac{\mathbb{P}(z_i|z_{i-1},\delta_i,\theta')}{\mathbb{P}(z_i|z_{i-1},\delta_i,\theta)}\right). \tag{32}$$

For the initialization of the prior parameters, we use the Baum-Welch algorithm described in section 3.

1.5.3.13 `prior::THMMBoolInferred`

The class `prior::THMMBoolInferred` applies for HMMs where the hidden state z_i is a binary variable with $S = 2$ states: 0 and 1. The transition matrix for this prior is parametrized by two parameters π and γ, as described in section 3.1. The class inherits from `prior::THMMPrior`, and equation 32 is used to calculate the log likelihood ratio for an update of $\theta = \pi, \gamma$.

1.5.3.14 `prior::THMMBoolGeneratingMatrixInferred`

The class `prior::THMMBoolGeneratingMatrixInferred` applies for HMMs where the hidden state z_i is a binary variable with $S = 2$ states: 0 and 1. The transition matrix for this prior is parametrized by two parameters $\log\Lambda_1$ and $\log\Lambda_2$, as described in section 3.1. The class inherits from `prior::THMMPrior`, and equation 32 is used to calculate the log likelihood ratio for an update of $\theta = \log\Lambda_1, \log\Lambda_2$.
In simulations, we noted that for large distances (i.e. when the transition probabilities approach the stationary probabilities), different combinations of $\log\Lambda_1$ and $\log\Lambda_2$ result in exactly the same transition probabilities. The likelihood surface therefore results in a ridge. We then encountered the issue that during initialization, the Nelder-Mead algorithm optimizes $\log\Lambda_1$ and $\log\Lambda_2$ along that ridge, until the values get so large that the matrix exponential has numeric inaccuracies, and odd transition probabilities result. This is an undesired behaviour. We therefore implemented the prior `prior::THMMBoolInferred`, which circumvents this issue with a more elegant parametrization, and recommend to use that class for HMM prior on booleans.

1.5.3.15 `prior::THMMLadderInferred`

The class `prior::THMMLadderInferred` applies for HMMs where the generator matrix has a ladder-type structure that only allows for transitions between neighbouring states. The transition matrix for this prior is parametrized by two parameters κ, as described in section 3.1.4. The class inherits from `prior::THMMPrior`, and equation 32 is used to calculate the log likelihood ratio for an update of $\theta = \kappa$.

1.5.3.16 `prior::THMMCombinedScaledLadderInferred`

The class `prior::THMMCombinedScaledLadderInferred` implements one or more HMMs where each chain has a transition probability given by a scaled ladder-type generator matrix. The concept of decomposing a HMM on combined states into multiple chains with specific transition probabilities is described in section 3.6.3. This class is templated and takes any scaled latter-type transition matrix, with or without attractor. Specifically, the template argument `TransitionMatrixType` can either be `TTransitionMatrixScaledLadder`, `TTransitionMatrixScaledLadderAttractorShift` or `TTransitionMatrixScaledLadderAttractorShift2`, which are described in section 3.1. If the transition matrix has an attractor, the attractor for each chain must be specified in the constructor. These attractors are fixed and will not be updated in the MCMC.

Apart from (optional) attractors, the transition matrix is parametrized by three parameters, $\boldsymbol{\kappa}$, $\boldsymbol{\nu}$ and $\boldsymbol{\mu}$. These parameters are vectors where the size is given by the number of chains. If a chain c consists of 3 states, the parameter μ_c is superficial and is not included. All $\boldsymbol{\kappa}$, $\boldsymbol{\nu}$ and $\boldsymbol{\mu}$ must be strictly positive. However, the parametrization of the scaled ladder only makes sense if $\boldsymbol{\nu}$ and $\boldsymbol{\mu}$ are in [0,1], since only then the middle state is more frequent than all other states. We therefore recommend to put a sparse prior on them. In the initialization with EM, we can not specify a prior, but we still want a spare solution. Therefore, we make sure $\boldsymbol{\nu}$ and $\boldsymbol{\mu}$ are in [0,1] by applying the logistic function after optimization by Nelder-Mead.
The class inherits from `prior::THMMPrior`, and equation 32 is used to calculate the log likelihood ratio for an update of $\theta = \boldsymbol{\kappa}, \boldsymbol{\nu}, \boldsymbol{\mu}$.

1.5.3.17 `prior::THMMCategoricalInferred`

The class `prior::THMMCategoricalInferred` applies for HMMs where the transition matrix is of type `TTransitionMatrixCategorical`. The hidden states are categorical with a total of D states. The transition matrix for this prior is parametrized by three parameters π, γ and $\boldsymbol{\rho}$, where $\boldsymbol{\rho} = \rho_1, \ldots, \rho_D$ is a vector with the constraint $\sum_d \rho_d = 1$. The class inherits from `prior::THMMPrior`, and equation 32 is used to calculate the log likelihood ratio for an update of $\theta = \pi, \gamma, \boldsymbol{\rho}$.

1.5.3.18 `prior::TTwoNormalsMixedModelInferred`

This class implements a special form of a mixture model of $K = 2$ normal distributions:

$$\mathbb{P}(x_i) = (1 - z_i)\Phi(\mu, \sigma_0^2) + z_i\Phi(\mu, \sigma_0^2 + \sigma_1^2).$$

Here, z_i denotes an binary indicator variable representing the two models, and $\Phi(\mu, \sigma^2)$ denotes the normal density with mean μ and variance σ^2. In this parametrization, the two mixture components share a common mean μ and the variance is larger in case $z_i = 1$ than in the case $z_i = 0$.
This prior is thus parametrized by

- A mean (for both components), μ (`_mus`).
- A variance for the null-model, σ_0^2, and for the one-model σ_1^2 (`_vars`).
- Indicator variables $\boldsymbol{z}$ (`_z`). These are linear with size N.

The log prior density of a value x_i is:

$$\log \mathbb{P}(x_i|\mu, \sigma_0^2, \sigma_1^2, z_i) = \begin{cases} -\frac{1}{2}\log(\sigma_0^2) - \frac{1}{2}\log(2\pi) - \frac{1}{2\sigma_0^2}(x_i - \mu)^2 \text{ if } z_i = 0, \\ -\frac{1}{2}\log(\sigma_0^2 + \sigma_1^2) - \frac{1}{2}\log(2\pi) - \frac{1}{2(\sigma_0^2+\sigma_1^2)}(x_i - \mu)^2 \text{ if } z_i = 1. \end{cases} \tag{33}$$

The log prior ratio is:

$$\log p_x = \log\left(\frac{\mathbb{P}(x_i'|\mu, \sigma_0^2, \sigma_1^2, z_i)}{\mathbb{P}(x_i|\mu, \sigma_0^2, \sigma_1^2, z_i)}\right) = \begin{cases} \frac{1}{2\sigma_0^2}\left((x_i - \mu)^2 - (x_i' - \mu)^2\right) \text{ if } z_i = 0 \\ \frac{1}{2(\sigma_0^2+\sigma_1^2)}\left((x_i - \mu)^2 - (x_i' - \mu)^2\right) \text{ if } z_i = 1. \end{cases}$$

The log likelihood ratio l_μ for an update of μ is

$$\begin{aligned} l_\mu &= \log\left(\frac{\mathbb{P}(\boldsymbol{x}|\mu', \sigma_0^2, \sigma_1^2, \boldsymbol{z})}{\mathbb{P}(\boldsymbol{x}|\mu, \sigma_0^2, \sigma_1^2, \boldsymbol{z})}\right) \\ &= \frac{1}{2\sigma_0^2}\sum_{i=1}^{N}(1 - z_i)\left((x_i - \mu)^2 - (x_i - \mu')^2\right) + \frac{1}{2\left(\sigma_0^2 + \sigma_1^2\right)}\sum_{i=1}^{N} z_i\left((x_i - \mu)^2 - (x_i - \mu')^2\right). \end{aligned}$$

The log likelihood ratio $l_{\sigma_0^2}$ for an update of parameter σ_0^2 is

$$\begin{aligned} l_{\sigma_0^2} &= \log\left(\frac{\mathbb{P}(\boldsymbol{x}|\mu,\sigma_0^{2'},\sigma_1^2,\boldsymbol{z})}{\mathbb{P}(\boldsymbol{x}|\mu,\sigma_0^2,\sigma_1^2,\boldsymbol{z})}\right) \\ &= \frac{1}{2}\log\left(\frac{\sigma_0^2}{\sigma_0^{2'}}\right)\sum_{i=1}^{N}(1-z_i) + \frac{1}{2}\log\left(\frac{\sigma_0^2+\sigma_1^2}{\sigma_0^{2'}+\sigma_1^2}\right)\sum_{i=1}^{N} z_i + \\ &\quad \frac{1}{2}\left(\frac{1}{\sigma_0^2}-\frac{1}{\sigma_0^{2'}}\right)\sum_{i=1}^{N}(1-z_i)(x_i-\mu)^2 + \frac{1}{2}\left(\frac{1}{\sigma_0^2+\sigma_1^2}-\frac{1}{\sigma_0^{2'}+\sigma_1^2}\right)\sum_{i=1}^{N} z_i(x_i-\mu)^2. \end{aligned}$$

The log likelihood ratio $l_{\sigma_1^2}$ for an update of σ_1^2 is

$$\begin{aligned} l_{\sigma_1^2} &= \log\left(\frac{\mathbb{P}(\boldsymbol{x}|\mu,\sigma_0^2,\sigma_1^{2'},\boldsymbol{z})}{\mathbb{P}(\boldsymbol{x}|\mu,\sigma_0^2,\sigma_1^2,\boldsymbol{z})}\right) \\ &= \frac{1}{2}\log\left(\frac{\sigma_0^2+\sigma_1^2}{\sigma_0^2+\sigma_1^{2'}}\right)\sum_{i=1}^{N} z_i + \frac{1}{2}\left(\frac{1}{\sigma_0^2+\sigma_1^2}-\frac{1}{\sigma_0^2+\sigma_1^{2'}}\right)\sum_{i=1}^{N} z_i(x_i-\mu)^2. \end{aligned}$$

The posterior distribution of z_i can be calculated analytically, because its values are limited to few discrete states. We therefore directly sample new values of z_i from the posterior distribution as specified in (1). The likelihood for sampling a specific z_i is then given by $\mathbb{P}(x_i|\mu,\sigma_{z_i}^2)$, and the prior by $\mathbb{P}(z_i|\pi)$. Note that this hold regardless of the prior distribution, e.g. HMMs.
We use an Expectation-Maximization (EM) algorithm for the initialization of the prior parameters. To simplify the EM, we assume that the variance of the null model (σ_0^2) is independent of the variance of the one-model. The EM algorithm then runs exactly as described on page 82, except for μ, which is now independent of component k. We can therefore initialize μ with its MLE using (30), and don't need to update it during the EM. The initialization of the EM therefore also proceeds differently:

1. Calculate the overall MLE mean $\hat{\mu}$ and variance $\hat{\sigma}^2$ of all $\boldsymbol{x}$ using (30) and (31), respectively.
2. Calculate the log density $\log\mathbb{P}(x_{pi}|\hat{\mu},\hat{\sigma}^2)$ for each data point x_{pi} with (10). Then, calculate the average log density over all parameters as $\bar{\boldsymbol{d}} = \frac{1}{P}\sum_{p=1}^{P}\log\mathbb{P}(x_{pi}|\hat{\mu},\hat{\sigma}^2)$.
3. Data points with a low density do not fit the overall variance well, because they are far away from the mean. Therefore, sort the densities $\bar{\boldsymbol{d}}$ in ascending order. Assign the first 10% of the data points $\boldsymbol{x}$ with the lowest density to $z=1$ and the rest to $z=0$.
4. Calculate the MLE variances $\hat{\sigma}_0^2$ and $\hat{\sigma}_1^2$ based on this classification.
5. Initialize the latent variable z_i to the state that maximizes the emission probabilities.
6. Initialize σ_0^2 and σ_1^2 by re-calculating the MLE variance based on this classification.

We have now estimated mean and variance of null- and one-model independently, ignoring the condition that the variance of the one-model must be larger than the variance of the null-model. Therefore, we now set $\hat{\sigma_0^2} = \min(\sigma_0^2,\sigma_1^2)$ and $\hat{\sigma_1^2} = \max(\sigma_0^2,\sigma_1^2) - \hat{\sigma_0^2}$, and assign $\boldsymbol{z}$ and the prior on $\boldsymbol{z}$ accordingly.

1.5.3.19 `prior::TMultivariateNormalInferred`

This class implements a multivariate normal prior

$$\boldsymbol{x}_i \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma}),$$

where $\boldsymbol{x}_i$ is a D-dimensional vector, parametrized by a D-dimensional mean vector $\boldsymbol{\mu}$ and a DxD variance-covariance matrix $\boldsymbol{\Sigma}$. Since the inverse of $\boldsymbol{\Sigma}$ is positive definite and symmetric, it can be parametrized with aid of the Cholesky factorization:

$$\boldsymbol{\Sigma}^{-1} = \boldsymbol{M}\boldsymbol{M}^T, \tag{34}$$

where $\boldsymbol{M}$ is the lower triangular matrix:

$$\boldsymbol{M} = \frac{1}{m} \cdot \begin{pmatrix} m_{11} & 0 & \dots & 0 \\ m_{21} & m_{22} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ m_{D1} & m_{D2} & \dots & m_{DD} \end{pmatrix},$$

for a positive scale factor $m > 0$. This factorization allows for independent updates of the elements in the matrix $\boldsymbol{M}$ while preserving the intrinsic properties of a variance-covariance matrix. The density of the multivariate normal distribution is:

$$\begin{aligned} \mathbb{P}(\boldsymbol{x}_i|\boldsymbol{\mu}, \boldsymbol{\Sigma}) &= \frac{1}{\sqrt{(2\pi)^D \det \boldsymbol{\Sigma}}} \exp\left(-\frac{1}{2}(\boldsymbol{x}_i - \boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1}(\boldsymbol{x}_i - \boldsymbol{\mu})\right) \\ &= \frac{\frac{1}{m^D}|\prod_{r=1}^D m_{rr}|}{\sqrt{(2\pi)^D}} \exp\left(-\frac{1}{2m^2}\sum_{s=1}^D \left(\sum_{r=s}^D m_{rs}(x_{ir} - \mu_r)\right)^2\right), \end{aligned} \tag{35}$$

because the determinant of $\boldsymbol{\Sigma}$ is given by

$$\det \boldsymbol{\Sigma} = \left(\frac{1}{m^D}\prod_{r=1}^D m_{rr}\right)^{-2},$$

and

$$(\boldsymbol{x}_i - \boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1}(\boldsymbol{x}_i - \boldsymbol{\mu}) = ||\boldsymbol{M}^T(\boldsymbol{x}_i - \boldsymbol{\mu})||^2 = \frac{1}{m^2}\sum_{s=1}^D \left(\sum_{r=s}^D m_{rs}(x_{ir} - \mu_r)\right)^2.$$

New proposals of the variance-covariance matrix $\boldsymbol{\Sigma}$ are obtained by proposing new elements m_{rr}, m_{rs} and m. Although this is not part of this prior, commonly recommended prior distributions for these parameters are $m \sim \text{Exp}(\lambda_m)$, $m_{rr}^2 \sim \chi_{\delta_r}^2$ where $\delta_r = D + 1 - r$ and $m_{rs} \sim \mathcal{N}(0, 1)$. With m fixed and these prior distributions for m_{rr} and m_{rs}, $\boldsymbol{\Sigma}^{-1}$ is sampled from a Wishart distribution with D degrees of freedom and scale m (Anderson, 2003).
A multivariate normal distribution is hence parametrized by

- Means $\boldsymbol{\mu}$ of size D (`_mus`).
- A scale factor m of size 1 (`_m`).
- Diagonal elements $\boldsymbol{m}_{rr}$ of size D (`_Mrr`).

- Off-diagonal elements $\boldsymbol{m}_{rs}$ of size T (`_Mrs`). T is given by:

$$T = \frac{D(D-1)}{2}. \tag{36}$$

From T, we can also calculate D with:

$$D = \frac{1+\sqrt{1+8T}}{2}.$$

To get from a coordinate r, s the linear index p, we calculate

$$p = \frac{r(r-1)}{2} + s.$$

The parameter on which this prior is defined, $\boldsymbol{X}$, must have dimensions $N \times D$, i.e. the number of dimensions of the multivariate normal prior should be the columns of $\boldsymbol{X}$. This is beneficial for cache handling: When calculating the prior densities, we always need one row $\boldsymbol{x}_i$ from $\boldsymbol{X}$, which corresponds to one consecutive chunk of memory because we store matrices in a row-major order.
The multivariate normal prior is a non-iid prior, as the prior density of one x_{id} depends on all $\boldsymbol{x}_i$. The log prior density of a value x_{id} with parameters $\boldsymbol{\mu}$, $\boldsymbol{M} = \{m, \boldsymbol{m}_{rr}, \boldsymbol{m}_{rs}\}$ is

$$\log \mathbb{P}(x_{id}|\boldsymbol{\mu}, \boldsymbol{M}) = -D \log m + \sum_{r=1}^{D} m_{rr} - \frac{D}{2}\log(2\pi) - \frac{1}{2m^2}\sum_{s=1}^{D}\left(\sum_{r=s}^{D} m_{rs}(x_{ir} - \mu_r)\right)^2.$$

The log prior ratio is:

$$\begin{aligned}
\log p_{x_{id}} = \log\left(\frac{\mathbb{P}(x'_{id}|\boldsymbol{\mu}, \boldsymbol{M})}{\mathbb{P}(x_{id}|\boldsymbol{\mu}, \boldsymbol{M})}\right) &= \frac{1}{2m^2}\sum_{s=1}^{D}\left(\left(\sum_{r=s}^{D} m_{rs}(x_{ir} - \mu_r)\right)^2 - \left(\sum_{r=s}^{D} m_{rs}(x'_{ir} - \mu_r)\right)^2\right) \\
&= \frac{1}{2m^2}\Bigg(\left(x_{id}^2 - x_{id}'^2 + 2\mu_d(x'_{id} - x_{id})\right)\sum_{r=1}^{d} m_{dr}^2 + \\
&\quad 2(x_{id} - x'_{id})\sum_{s=1}^{S} m_{ds}\sum_{r=s, r\neq d}^{D} m_{rs}(x_{ir} - \mu_r)\Bigg).
\end{aligned}$$

where $S = \min(d, D-1)$.
The log likelihood ratio l_{μ_d} for an update μ_d is:

$$\begin{aligned}
l_{\mu_d} = \log\left(\frac{\mathbb{P}(\boldsymbol{X}|\mu'_d, \boldsymbol{M})}{\mathbb{P}(\boldsymbol{X}|\mu_d, \boldsymbol{M})}\right) &= \frac{1}{2m^2}\sum_{i=1}^{N}\sum_{s=1}^{D}\left(\left(\sum_{r=s}^{D} m_{rs}(x_{ir} - \mu_r)\right)^2 - \left(\sum_{r=s}^{D} m_{rs}(x_{ir} - \mu'_r)\right)^2\right) \\
&= \frac{1}{2m^2}\Bigg(N(\mu_d^2 - \mu_d'^2)\sum_{r=1}^{d} m_{dr}^2 + 2(\mu'_d - \mu_d)\left(\sum_{r=1}^{d} m_{dr}^2\right)\left(\sum_{i=1}^{N} x_{id}\right) + \\
&\quad 2(\mu'_d - \mu_d)\sum_{i=1}^{N}\sum_{s=1}^{S} m_{ds}\sum_{r=s, r\neq d}^{D} m_{rs}(x_{ir} - \mu_r)\Bigg).
\end{aligned}$$

The log likelihood ratio l_m for an update of m is:

$$\begin{aligned}
l_m &= \log\left(\frac{\mathbb{P}(\boldsymbol{X}|\boldsymbol{\mu}, \boldsymbol{M}')}{\mathbb{P}(\boldsymbol{X}|\boldsymbol{\mu}, \boldsymbol{M})}\right) \\
&= ND(\log m - \log m') + \frac{1}{2}\left(\frac{1}{m^2} - \frac{1}{m'^2}\right)\sum_{i=1}^{N}\sum_{s=1}^{D}\left(\sum_{r=s}^{D} m_{rs}(x_{ir} - \mu_r)\right)^2.
\end{aligned}$$

The log likelihood ratio $l_{m_{rr}}$ for an update of parameter m_{rr} is:

$$\begin{aligned}
l_{m_{rr}} &= \log\left(\frac{\mathbb{P}(\boldsymbol{X}|\boldsymbol{\mu},\boldsymbol{M}')}{\mathbb{P}(\boldsymbol{X}|\boldsymbol{\mu},\boldsymbol{M})}\right) \\
&= N(\log m'_{rr} - \log m_{rr}) + \frac{1}{2m^2}\sum_{i=1}^{N}\sum_{s=1}^{D}\left(\left(\sum_{r=s}^{D} m_{rs}(x_{ir}-\mu_r)\right)^2 - \left(\sum_{r=s}^{D} m'_{rs}(x_{ir}-\mu_r)\right)^2\right) \\
&= N(\log m'_{rr} - \log m_{rr}) + \frac{1}{2m^2}\Big((m_{rr}^2 - m_{rr}'^2)\sum_{i=1}^{N}(x_{ir}-\mu_r)^2 + \\
&\quad 2(m_{rr}-m'_{rr})\sum_{i=1}^{N}(x_{ir}-\mu_r)\sum_{d=r+1}^{D} m_{dr}(x_{id}-\mu_d)\Big).
\end{aligned}$$

The log likelihood ratio $l_{m_{rs}}$ for an update of parameter m_{rs} is:

$$\begin{aligned}
l_{m_{rs}} &= \log\left(\frac{\mathbb{P}(\boldsymbol{X}|\boldsymbol{\mu},\boldsymbol{M}')}{\mathbb{P}(\boldsymbol{X}|\boldsymbol{\mu},\boldsymbol{M})}\right) \\
&= \frac{1}{2m^2}\sum_{i=1}^{N}\sum_{s=1}^{D}\left(\left(\sum_{r=s}^{D} m_{rs}(x_{ir}-\mu_r)\right)^2 - \left(\sum_{r=s}^{D} m'_{rs}(x_{ir}-\mu_r)\right)^2\right) \\
&= \frac{1}{2m^2}\left((m_{rs}^2 - m_{rs}'^2)\sum_{i=1}^{N}(x_{ir}-\mu_r)^2 + 2(m_{rs}-m'_{rs})\sum_{i=1}^{N}(x_{ir}-\mu_r)\sum_{d=s,d\neq r}^{D} m_{ds}(x_{id}-\mu_d)\right).
\end{aligned}$$

Note that if $D = 1$, parameter m_{rs} is non-existant and m_{rr} is superficial and fixed to 1, such that only m is estimated.
For the initialization of the prior parameters, we calculate the MLE of $\boldsymbol{\mu}$ and $\boldsymbol{M}$. The MLE for one $\hat{\mu}_d$ is given by:

$$\hat{\mu}_d = \frac{1}{N}\sum_{i=1}^{N} x_{id}. \tag{37}$$

If the prior is defined on a vector of P parameters, the MLE is given by

$$\hat{\mu}_d = \frac{1}{\sum_{p=1}^{P} N_p}\sum_{p=1}^{P}\sum_{i=1}^{N_p} x_{id}^{(p)}.$$

The MLE for $\boldsymbol{\Sigma}$ is given by:

$$\hat{\boldsymbol{\Sigma}} = \frac{1}{N}\sum_{i=1}^{N}(\boldsymbol{x}_i - \hat{\boldsymbol{\mu}})(\boldsymbol{x}_i - \hat{\boldsymbol{\mu}})^T. \tag{38}$$

If the prior is defined on a vector of parameters, the MLE is given by

$$\hat{\boldsymbol{\Sigma}} = \frac{1}{\sum_{p=1}^{P} N_p}\sum_{p=1}^{P}\sum_{i=1}^{N}(\boldsymbol{x}_i^{(p)} - \hat{\boldsymbol{\mu}})(\boldsymbol{x}_i^{(p)} - \hat{\boldsymbol{\mu}})^T.$$

We can calculate $\boldsymbol{M}$ by Cholesky factorization from the inverse of $\boldsymbol{\Sigma}$ with 34. We then set $m = 1$ and m_{rr} and m_{rs} to the respective elements in $\boldsymbol{M}$.

1.5.3.20 `prior::TTwoMultivariateNormalsMixedModelInferred`

This class implements a special mixture model of $K = 2$ multivariate normal distributions:

$$\mathbb{P}(\boldsymbol{x}_i) = (1 - z_i)\Phi(\boldsymbol{\mu}, \boldsymbol{\Sigma}^{(0)}) + z_i\Phi(\boldsymbol{\mu}, \boldsymbol{\Sigma}^{(1)}).$$

Here, z_i is an binary indicator variable representing the two models, and $\Phi(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ denotes the multivariate normal density with a D-dimensional mean vector $\boldsymbol{\mu}$ and DxD variance-covariance matrix $\boldsymbol{\Sigma}$. $\boldsymbol{x}_i$ itself is a D-dimensional vector. In this parametrization, the two mixture components share a common mean vector $\boldsymbol{\mu}$ and the variance-covariance matrix is larger in case $z_i = 1$ than in the case $z_i = 0$.
We achieve this with the following parametrization: The variance-covariance matrix of the null-model is factorized as for `prior::TMultivariateNormalInferred` by applying Cholesky factorization to $\boldsymbol{\Sigma}^{(0)}$, resulting in $\boldsymbol{M}^{(0)}$, which is split into the elements $m^{(0)}$, $\boldsymbol{m}_{rr}^{(0)}$ and $\boldsymbol{m}_{rs}^{(0)}$. The density of the null-model is thus given by equation 35. The variance-covariance matrix of the one-model, $\boldsymbol{\Sigma}^{(1)}$, is calculated in the following way:

1. Calculate
$$\boldsymbol{\Sigma}^{(0)} = \left(\boldsymbol{M}^{(0)}\boldsymbol{M}^{(0)^T}\right)^{-1}. \tag{39}$$

2. Do Eigendecomposition of $\boldsymbol{\Sigma}^{(0)}$:
$$\boldsymbol{\Sigma}^{(0)} = \boldsymbol{Q}\boldsymbol{\Lambda}^{(0)}\boldsymbol{Q}^{-1}. \tag{40}$$
Here, $\boldsymbol{\Lambda}^{(0)}$ is a diagonal matrix whose elements are the eigenvalues of $\boldsymbol{\Sigma}^{(0)}$.

3. Multiply with $\boldsymbol{\rho}$ to get $\boldsymbol{\Lambda}^{(1)}$, which contains the eigenvalues of $\boldsymbol{\Sigma}^{(1)}$:
$$\boldsymbol{\Lambda}^{(1)} = \boldsymbol{\Lambda}^{(0)}\left(\mathbf{1} + \boldsymbol{\rho}\right), \tag{41}$$
in matrix notation:
$$\begin{pmatrix} \lambda_1^{(1)} & 0 & \dots & 0 \\ 0 & \lambda_2^{(1)} & \dots & 0 \\ \vdots & & & \\ 0 & 0 & \dots & \lambda_D^{(1)} \end{pmatrix} = \begin{pmatrix} \lambda_1^{(0)} & 0 & \dots & 0 \\ 0 & \lambda_2^{(0)} & \dots & 0 \\ \vdots & & & \\ 0 & 0 & \dots & \lambda_D^{(0)} \end{pmatrix} \begin{pmatrix} 1+\rho_1 \\ 1+\rho_2 \\ \vdots \\ 1+\rho_D \end{pmatrix}.$$

4. Now construct $\boldsymbol{\Sigma}^{(1)}$:
$$\boldsymbol{\Sigma}^{(1)} = \boldsymbol{Q}\boldsymbol{\Lambda}^{(1)}\boldsymbol{Q}^{-1}. \tag{42}$$

5. From this, we get $\boldsymbol{M}^{(1)}$ with (34).

6. We split $\boldsymbol{M}^{(1)}$ into the elements $\boldsymbol{m}_{rr}^{(1)}$ and $\boldsymbol{m}_{rs}^{(1)}$, and set $m^{(1)} = 1$.

For $\boldsymbol{\rho} > 0$, the formulation ensures that the variances and covariances are larger in case $z_i = 1$ than in the case $z_i = 0$. We can think of $\boldsymbol{\Sigma}^{(1)}$ as a stretched version of $\boldsymbol{\Sigma}^{(0)}$, as we stretch $\boldsymbol{\Sigma}^{(0)}$ in every dimension with ρ_d.
A multivariate normal mixed model with two components is parametrized by

- A mean vector (for both components), $\boldsymbol{\mu}$ of size D (`_mus`).
- Null-model: A scale factor $m^{(0)}$ of size 1 (`_m`).

- Null-model: Diagonal elements $\boldsymbol{m}_{rr}^{(0)}$ of size D (`_Mrr`).
- Null-model: Off-diagonal elements $\boldsymbol{m}_{rs}^{(0)}$ of size T (`_Mrs`), see (36).
- Indicator variables $\boldsymbol{z}$ of size N (`_z`).
- Stretching parameter $\boldsymbol{\rho}$ of size D (`_rhos`).

The parameter on which this prior is defined, $\boldsymbol{X}$, must have dimensions $N \times D$, i.e. the number of dimensions of the multivariate normal prior should be the columns of $\boldsymbol{X}$. This is beneficial for cache handling: When calculating the prior densities, we always need one row $\boldsymbol{x}_i$ from $\boldsymbol{X}$, which corresponds to one consecutive chunk of memory because we store matrices in a row-major order.
The multivariate normal distribution is a non-iid prior, as the prior density of one x_{id} depends on all $\boldsymbol{x}_i$. The log prior density of a value x_{id} with parameters $\boldsymbol{\mu}$, $\boldsymbol{M}^{(0)} = \{m, \boldsymbol{m}_{rr}^{(0)}, \boldsymbol{m}_{rs}^{(0)}\}$, $\boldsymbol{M}^{(1)}$ is given by (57) and the log prior ratio is given by (58) (with $\boldsymbol{\mu}^{(k)} = \boldsymbol{\mu}$).
The log likelihood ratio l_{μ_d} for an update of μ_d is:

$$
\begin{aligned}
l_{\mu_d} &= \log\left(\frac{\mathbb{P}(\boldsymbol{X}|\mu_d', \boldsymbol{M}^{(0)}, \boldsymbol{M}^{(1)}, \boldsymbol{z})}{\mathbb{P}(\boldsymbol{X}|\mu_d, \boldsymbol{M}^{(0)}, \boldsymbol{M}^{(1)}, \boldsymbol{z})}\right) \\
&= \frac{1}{2m^{(0)2}} \sum_{i=1}^{N} (1 - z_i) \sum_{s=1}^{D} \left(\left(\sum_{r=s}^{D} m_{rs}^{(0)} (x_{ir} - \mu_r) \right)^2 - \left(\sum_{r=s}^{D} m_{rs}^{(0)} (x_{ir} - \mu_r') \right)^2 \right) + \\
&\quad \frac{1}{2m^{(1)2}} \sum_{i=1}^{N} z_i \sum_{s=1}^{D} \left(\left(\sum_{r=s}^{D} m_{rs}^{(1)} (x_{ir} - \mu_r) \right)^2 - \left(\sum_{r=s}^{D} m_{rs}^{(1)} (x_{ir} - \mu_r') \right)^2 \right) \\
&= \sum_{k=0}^{1} \left(\frac{1}{2m^{(k)2}} \left((\mu_d^2 - \mu_d'^2) \left(\sum_{r=1}^{d} m_{dr}^{(k)2} \right) \left(\sum_{i=1}^{N} \mathcal{I}(z_i = k) \right) + \right. \right. \\
&\quad 2(\mu_d' - \mu_d) \sum_{r=1}^{d} m_{dr}^{(k)2} \sum_{i=1}^{N} \mathcal{I}(z_i = k) x_{id} + \\
&\quad \left. \left. 2(\mu_d' - \mu_d) \sum_{i=1}^{N} \mathcal{I}(z_i = k) \sum_{s=1}^{S} m_{ds}^{(k)} \sum_{r=s, r \neq d}^{D} m_{rs}^{(k)} (x_{ir} - \mu_r) \right) \right).
\end{aligned}
$$

Whenever we update any $m^{(0)}$, $m_{rr}^{(0)}$, $m_{rs}^{(0)}$ and ρ_d, we need to re-calculate $\boldsymbol{M}_1$. It is important to realize that the entire matrix $\boldsymbol{M}_1$ changes after any such update. Hence, for each of these updates, we need to calculate

$$
\begin{aligned}
&\log\left(\frac{\mathbb{P}(\boldsymbol{X}|\boldsymbol{\mu}, \boldsymbol{M}^{(1)'}, \boldsymbol{z})}{\mathbb{P}(\boldsymbol{X}|\boldsymbol{\mu}, \boldsymbol{M}^{(1)}, \boldsymbol{z})}\right) = \\
&\quad \sum_{i=1}^{N} z_i \left(\sum_{r=1}^{D} \log\left(\frac{m_{rr}^{(1)'}}{m_{rr}^{(1)}}\right) + \frac{1}{2m^{(1)2}} \sum_{s=1}^{D} \left(\left(\sum_{r=s}^{D} m_{rs}^{(1)} (x_{ir} - \mu_r) \right)^2 - \left(\sum_{r=s}^{D} m_{rs}^{(1)'} (x_{ir} - \mu_r) \right)^2 \right) \right).
\end{aligned}
$$

The log likelihood ratio $l_{m^{(0)}}$ for an update of $m^{(0)}$ is:

$$
\begin{aligned}
l_{m^{(0)}} &= \log\left(\frac{\mathbb{P}(\boldsymbol{X}|\boldsymbol{\mu},\boldsymbol{M}^{(0)'},\boldsymbol{M}^{(1)'},\boldsymbol{z})}{\mathbb{P}(\boldsymbol{X}|\boldsymbol{\mu},\boldsymbol{M}^{(0)},\boldsymbol{M}^{(1)},\boldsymbol{z})}\right) \\
&= D(\log m^{(0)} - \log m^{(0)'})\sum_{i=i}^{N}(1-z_i) + \frac{1}{2}\left(\frac{1}{m^{(0)2}} - \frac{1}{m^{(0)'2}}\right)\sum_{i=1}^{N}(1-z_i)\sum_{s=1}^{D}\left(\sum_{r=s}^{D} m_{rs}^{(0)}(x_{ir}-\mu_r)\right)^2 + \\
&\quad \log\left(\frac{\mathbb{P}(\boldsymbol{X}|\boldsymbol{\mu},\boldsymbol{M}^{(1)'},\boldsymbol{z})}{\mathbb{P}(\boldsymbol{X}|\boldsymbol{\mu},\boldsymbol{M}^{(1)},\boldsymbol{z})}\right).
\end{aligned}
$$

The log likelihood ratio $l_{m_{rr}^{(0)}}$ for an update of $m_{rr}^{(0)}$ is:

$$
\begin{aligned}
l_{m_{rr}^{(0)}} &= \log\left(\frac{\mathbb{P}(\boldsymbol{X}|\boldsymbol{\mu},\boldsymbol{M}^{(0)'},\boldsymbol{M}^{(1)'},\boldsymbol{z})}{\mathbb{P}(\boldsymbol{X}|\boldsymbol{\mu},\boldsymbol{M}^{(0)'},\boldsymbol{M}^{(1)'},\boldsymbol{z})}\right) \\
&= \sum_{i=1}^{N}(1-z_i)\left(\log\left(\frac{m_{rr}^{(0)'}}{m_{rr}^{(0)}}\right) + \frac{1}{2m^{(0)2}}\sum_{s=1}^{D}\left(\left(\sum_{d=s}^{D} m_{ds}^{(0)}(x_{id}-\mu_d)\right)^2 - \left(\sum_{d=s}^{D} m_{ds}^{(0')}(x_{id}-\mu_d)\right)^2\right)\right) + \\
&\quad \log\left(\frac{\mathbb{P}(\boldsymbol{X}|\boldsymbol{\mu},\boldsymbol{M}^{(1)'},\boldsymbol{z})}{\mathbb{P}(\boldsymbol{X}|\boldsymbol{\mu},\boldsymbol{M}^{(1)},\boldsymbol{z})}\right) \\
&= (\log m_{rr}^{(0)'} - \log m_{rr}^{(0)})\sum_{i=1}^{N}(1-z_i) + \frac{1}{2m^{(0)2}}\Big((m_{rr}^{(0)2} - m_{rr}^{(0)'2})\sum_{i=1}^{N}(1-z_i)(x_{ir}-\mu_r)^2 + \\
&\quad 2(m_{rr}^{(0)} - m_{rr}^{(0)'})\sum_{i=1}^{N}(1-z_i)(x_{ir}-\mu_r)\sum_{d=r+1}^{D} m_{dr}^{(0)}(x_{id}-\mu_d)\Big) + \\
&\quad \log\left(\frac{\mathbb{P}(\boldsymbol{X}|\boldsymbol{\mu},\boldsymbol{M}^{(1)'},\boldsymbol{z})}{\mathbb{P}(\boldsymbol{X}|\boldsymbol{\mu},\boldsymbol{M}^{(1)},\boldsymbol{z})}\right).
\end{aligned}
$$

The log likelihood ratio $l_{m_{rs}^{(0)}}$ for an update of $m_{rs}^{(0)}$ is:

$$
\begin{aligned}
l_{m_{rs}^{(0)}} &= \log\left(\frac{\mathbb{P}(\boldsymbol{X}|\boldsymbol{\mu},\boldsymbol{M}^{(0)'},\boldsymbol{M}^{(1)'},\boldsymbol{z})}{\mathbb{P}(\boldsymbol{X}|\boldsymbol{\mu},\boldsymbol{M}^{(0)'},\boldsymbol{M}^{(1)'},\boldsymbol{z})}\right) \\
&= \frac{1}{2m^{(0)2}}\sum_{i=1}^{N}(1-z_i)\sum_{s=1}^{D}\left(\left(\sum_{r=s}^{D} m_{rs}^{(0)}(x_{ir}-\mu_r)\right)^2 - \left(\sum_{r=s}^{D} m_{rs}^{(0)'}(x_{ir}-\mu_r)\right)^2\right) \\
&\quad + \log\left(\frac{\mathbb{P}(\boldsymbol{X}|\boldsymbol{\mu},\boldsymbol{M}^{(1)'},\boldsymbol{z})}{\mathbb{P}(\boldsymbol{X}|\boldsymbol{\mu},\boldsymbol{M}^{(1)},\boldsymbol{z})}\right) \\
&= \frac{1}{2m^{(0)2}}\Bigg((m_{rs}^{(0)2} - m_{rs}^{(0)'2})\sum_{i=1}^{N}(1-z_i)(x_{ir}-\mu_r)^2 + \\
&\quad 2(m_{rs}^{(0)} - m_{rs}^{(0)'})\sum_{i=1}^{N}(1-z_i)(x_{ir}-\mu_r)\sum_{d=s,d\neq r}^{D} m_{ds}^{(0)}(x_{id}-\mu_d)\Bigg) + \\
&\quad \log\left(\frac{\mathbb{P}(\boldsymbol{X}|\boldsymbol{\mu},\boldsymbol{M}^{(1)'},\boldsymbol{z})}{\mathbb{P}(\boldsymbol{X}|\boldsymbol{\mu},\boldsymbol{M}^{(1)},\boldsymbol{z})}\right).
\end{aligned}
$$

Note that if $D = 1$, $\boldsymbol{m}_{rs}^{(0)}$ is non-existant and $\boldsymbol{m}_{rr}^{(0)}$ is superficial and fixed to 1, such that only m is estimated.
The log likelihood ratio l_{ρ_d} for an update of ρ_d is:

$$\begin{aligned} l_{\rho_d} &= \log\left(\frac{\mathbb{P}(\boldsymbol{X}|\boldsymbol{\mu}, \boldsymbol{M}^{(0)}, \boldsymbol{M}^{(1)'}, \boldsymbol{z})}{\mathbb{P}(\boldsymbol{X}|\boldsymbol{\mu}, \boldsymbol{M}^{(0)'}, \boldsymbol{M}^{(1)'}, \boldsymbol{z})}\right) \\ &= \log\left(\frac{\mathbb{P}(\boldsymbol{X}|\boldsymbol{\mu}, \boldsymbol{M}^{(1)'}, \boldsymbol{z})}{\mathbb{P}(\boldsymbol{X}|\boldsymbol{\mu}, \boldsymbol{M}^{(1)}, \boldsymbol{z})}\right). \end{aligned}$$

The posterior distribution of z_i can be calculated analytically, because its values are limited to few discrete states. We therefore directly sample new values of z_i from the posterior distribution as specified in (1). The likelihood for sampling a specific z_i is then given by $\mathbb{P}(\boldsymbol{x}_i|\boldsymbol{\mu}, \boldsymbol{M}^{(z_i)})$, and the prior by $\mathbb{P}(z_i|\pi)$. Note that this hold regardless of the prior distribution, e.g. also for HMMs.
We use an EM algorithm for the initialization of the prior parameters. To simplify the EM, we assume that the covariance matrix of the null-model ($\boldsymbol{\Sigma}^{(0)}$) is independent of the covariance matrix of the one-model ($\boldsymbol{\Sigma}^{(1)}$). The EM algorithm then runs exactly as described on page 86, except for the update step of $\boldsymbol{\mu}$. Since $\boldsymbol{\mu}$ is now independent of component k, we can directly estimate it with its MLE and don't need to update it during the EM. The initialization of the EM therefore also proceeds differently:

1. Calculate the overall MLE mean $\boldsymbol{\mu}$ and variance-covariance matrix $\boldsymbol{\Sigma}$ of all $\boldsymbol{X}$ using (37) and (38), respectively.

2. Calculate the log density $\log \mathbb{P}(\boldsymbol{x}_{pi}|\boldsymbol{\mu}, \boldsymbol{\Sigma})$ for each data vector $\boldsymbol{x}_{pi}$ for a given parameter p with (35). Then, calculate the average log density over all parameters as $\bar{\boldsymbol{d}} = \frac{1}{P}\sum_{p=1}^{P} \log \mathbb{P}(\boldsymbol{x}_{pi}|\boldsymbol{\mu}, \boldsymbol{\Sigma})$.

3. Data points with a low density do not fit the overall variance well. Therefore, sort the densities $\bar{\boldsymbol{d}}$ in ascending order. Assign the first 10% of the data points $\bar{\boldsymbol{X}}$ with the lowest density to $z = 1$ and the rest to $z = 0$.

4. Calculate the MLE variance-covariance matrices $\boldsymbol{\Sigma}^{(0)}$ and $\boldsymbol{\Sigma}^{(1)}$ based on this classification.

5. Initialize the latent variable z_i to the state that maximizes the emission probabilities.

6. Initialize $\boldsymbol{\Sigma}^{(0)}$ and $\boldsymbol{\Sigma}^{(1)}$ by re-calculating the MLE variance-covariance matrices based on this classification.

We have then estimated mean and variance of null- and one-model independently, ignoring the condition that the variance of the one-model must be larger than the variance of the null-model. We therefore do eigendecomposition of both $\boldsymbol{\Sigma}^{(0)}$ and $\boldsymbol{\Sigma}^{(1)}$ with (40) to obtain two matrices $\boldsymbol{\Lambda}^{(0)}$ and $\boldsymbol{\Lambda}^{(1)}$. We compute the norm of the diagonal $||\boldsymbol{\Lambda}||_2$ for both and set the $\boldsymbol{\Lambda}$ that minimizes the norm to $\boldsymbol{\Lambda}_0$ (and assign $\boldsymbol{z}$ and π accordingly). Using (41), we now estimate $\boldsymbol{\rho}$ as:

$$\boldsymbol{\rho} = \frac{\boldsymbol{\Lambda}^{(1)}}{\boldsymbol{\Lambda}^{(0)}} - 1,$$

where we set all $\rho_d < 0$ to 0.0001. We then re-calculate $\boldsymbol{\Lambda}^{(1)}$ with (41), using these $\boldsymbol{\rho}$, and $\boldsymbol{\Sigma}^{(1)}$ with (42).
This class has 2 objects of class `TVarCovarMatrix` (one per component) that each store four matrices: `_oldM`, `_M`, `_oldSigma` and `_Sigma`. `TVarCovarMatrix` then has a couple of functions implemented that allow the conversion from $\boldsymbol{M}$ to $\boldsymbol{\Sigma}$ and back.

1.5.4 Deterministic functions

`stattools` allows for deterministic relationships, specifically for link functions and linear models.

1.5.4.1 `TLinkFunction`

This class allows for an arbitary invertible link function. The class is parametrized by

- A parameter (`_param`).
- A function that transforms `_param` into the value of the node below (`below = _funDown(param)`).
- A function that transforms the value of the node below to `_param` (`param = _funUp(below)`). This function must be the inverse of `_funDown()`.

The parameter `_param` is stochastic and is updated. To evaluate its likelihood ratio, however, it must ask the parameter(s) below for the likelihood ratio. The parameter(s) below are deterministic. When calculating the likelihood ratio, they first sets their values deterministically according to function `valueForBelow()`, and then ask the box around them to calculate the likelihood ratio.
To initialize `_param`, `_funUp()` is called to transform the values of the parameter below to `_param`.

1.5.4.2 `TExp`

This is a common use-case of a link function:

- `_funDown()` corresponds to the exponential function such that `below = exp(param)`.
- `_funUp()` corresponds to the natural logarithm such that `param = log(below)`.

This link function is very useful if a parameter should be updated in the log-space. Updating $y = \log(x)$ instead of x is commonly done for strictly positive parameters x to achieve better mixing. This has two reasons:

1. When updating $y = \log(x)$, this naturally imposes an exponential prior with rate 1 on x, because $y \sim \mathcal{U}(-\infty, \infty)$ results in $x \sim e^{\mathcal{U}(-\infty,\infty)} = \text{Exp}(1)$.
2. The proposal range scales with the actual value of x. For example, a jump of 1 log-unit on y corresponds to a little change in x if x is small and to a large change in x if x is large.

In this example, y corresponds to `param` and x corresponds to `below`. The class is called `TExp` since $x = e^y$ (`below = exp(param)`), and we usually define relationships in this order (below given above).

1.5.4.3 `TLog`

This is another common use-case of a link function:

- `_funDown()` corresponds to the natural logarithm such that `below = log(param)`.
- `_funUp()` corresponds to the exponential function such that `param = exp(below)`.

Note that this class should not be used when updating a parameter in the log-space, instead use `TExp` which does the inverse!

1.5.4.4 `TLogistic`

This is another common use-case of a link function:

- `_funDown()` corresponds to the logistic function such that `below = logistic(param)`.
- `_funUp()` corresponds to the logit function such that `param = logit(below)`.

The logistic function (also called expit) is defined as $x = \frac{1}{1+e^{-y}}$, where y corresponds to `param` and x corresponds to `below`. The logit function is the inverse of the logistic function and defined as $y = \log\left(\frac{x}{1-x}\right)$.
This link function is useful when an unconstrained parameter $y \in [-\infty, \infty]$ should be transformed into a probability $x \in [0, 1]$.

1.5.4.5 `TLogit`

This is another common use-case of a link function:

- `_funDown()` corresponds to the logit function such that `below = logit(param)`.
- `_funUp()` corresponds to the logistic function such that `param = logistic(below)`.

The logit function is defined as $x = \log\left(\frac{y}{1-y}\right)$, where y corresponds to `param` and x corresponds to `below`. The logistic function (also called expit) is the inverse of the logit function and defined as $y = \frac{1}{1+e^{-x}}$.

1.5.4.6 `TLinearModel`

This class implements a general linear model of the form

$$\boldsymbol{Y} = \boldsymbol{X}\boldsymbol{B},$$

where $\boldsymbol{Y}$ is a $I \times J$ matrix, $\boldsymbol{X}$ is a $I \times K$ matrix of covariates and $\boldsymbol{B}$ is a $K \times J$ matrix of regression coefficients to be estimated. The columns of $\boldsymbol{Y}$ and $\boldsymbol{B}$ represent dependent variables that share the same set of explanatory variables. This is the most general implementation. A simple linear regression represents the special case $I = 1$, $J = 1$ and $K = 1$. A multiple linear regression represents the case $J = 1$ where there is a single dependent variable.
The general model can also be written as:

$$Y_{ij} = \beta_{0j} + \beta_{1j}X_{i1} + \beta_{2j}X_{i2} + \ldots + \beta_{Kj}X_{iK} = \beta_{0j} + \sum_{k=1}^{K} \beta_{kj}X_{ik},$$

where β_{0j} represents an optional intercept term that can be turned on and off through the boolean template argument `Intercept`.
The parameter to be estimated in this model is the matrix $\boldsymbol{B}$ with elements $[\boldsymbol{B}]_{kj} = \beta_{kj}$. When one β_{kj} is updated, one column $\boldsymbol{Y}_j = Y_{1j}, \ldots, Y_{Ij}$ changes deterministically. The likelihood ratio of this update must then be evaluated on this range of values.
To estimate initial values of $\boldsymbol{B}$, we use an ordinary least squares (OLS) scheme to solve the equation $\boldsymbol{X}\boldsymbol{B}_j = \boldsymbol{Y}_j$ for each $j = 1, \ldots, J$ using the Armadillo function `solve()`. To account for an intercept β_{0j}, we add an extra first row with 1's to $\boldsymbol{X}$ for the OLS.

1.6 Parameter definitions

The class `TObservationDefinition` is the main interface between user, developer and the library `stattools`. Inside this class, we can set all variables that "define" a parameter and observation. These are:

- `_priorParameters`: The prior parameters, in case these are fixed numbers.
- `_simulationFile`: The name of the file where simulated values should be written to.

There exists a derived class, `TParameterDefinition`, that adds extra members specific to MCMC parameters:

- `_isUpdated`: Whether the parameter should be updated or not.
- `_meanVarFile`: File to write posterior mean and value to.
- `_statePosteriorsFile`: File to write state posteriors for categorical variables to.
- `_traceFile`: File to write traces to.
- `_initPropKernel`: Initial jump size for proposals.
- `_propKernelDistr`: Name of the proposal kernel distribution (e.g. normal, uniform, integer).
- `_oneJumpSizeForAll`: Whether all parameters in an array should share the same jump size for proposing, or alternatively have all a unique jump size.
- `_initVal`: The initial values (given as string, to enable proper templating).

These two classes offer a variety of set- and get-functions to access these variables. A couple of other member variables are used to keep track of changed made to the definition.
All of the above variables have default values.

1.7 Providing initial values

If needed, parameters can be initialized based on fixed initial values that are provided as a string by the user. In this case, the default initialization (based on MLE and other algorithms) is deactivated.
The string of initial values can be multiple things:

- A comma-separated vector of numbers. The size of this vector must match the total size of the parameter. This option is chosen if the string contains a comma.
- A single number. All elements of the parameter will be set to this number. This option is chosen if the string is probably a single number (i.e. contains only numerical characters).
- A filename, containing the initial values. This file can have five formats:
    1. A trace file, as written by this framework. This option is chosen if the filename contains "trace". The header of the file is then parsed and all columns that start with the name of the parameter are extracted (note that the number of columns that match must be equal to the total size of the parameter). The last line (corresponding to the last iteration) of the file is used to initialize the parameter.

2. A file containing simulated values, as written by this framework. This option is chosen if the filename contains "simulated". The initialization then runs exactly as for the trace file above, as simulation files have the same format as trace files, except that they only consist of one line.
3. A file containing the posterior means and variances, as written by this framework. This option is chosen if the filename contains "meanVar". The header is then parsed and all relevant columns are extracted (note that the number of columns that match must be equal to the total size of the parameter). The first line (corresponding to the posterior mean) of the file is used to initialize the parameter.
4. A file containing the state posteriors as written by this framework. This option is chosen if the filename contains "statePosteriors". The header is then parsed and all relevant columns are extracted. For each column, the row index with the maximum value is used to initialize the parameter, since this corresponds to the state with the highest posterior probability.
5. A file containing only the values for the parameter (i.e. without header), where all values are in one column and the number of rows matches the total size of the parameter.
6. A file containing only the values for the parameter (i.e. without header), where all values are in one row and the number of columns matches the total size of the parameter.

1.8 Building a DAG

When initializing our framework, we can think of this as building a DAG. Therefore, the class that assembles all observations and parameters is called `TDAGBuilder`. This class is available as a singleton over `instances::dagBuilder()`. Let's go through its functionalities in the order we use them in a program:

- The constructors of `TParameter` and `TObservation` automatically add a base class pointer to `instances::dagBuilder()`. There is a check for unique parameter/observation names.
- When the MCMC starts, the function `buildDAG()` is called. This function executes the following steps:
  - First, we create an instance of class `TMCMCUserInterface`. The idea is that after the developer has specified this definitions, the user can come in and modify and potentially override these definitions. Specifically, we:
    1. Check if a config file has been defined on the command-line. The config file has the following format: A header with an mandatory column "name" and optional columns "observation", "priorParameters", "traceFile", "meanVarFile", "statePosteriorsFile", "simulationFile", "update", "propKernel", and "sharedJumpSize". Then, one row corresponds to one definition. We parse all modifications of one parameter/observation and re-set the definitions accordingly. Some columns only make sense for parameters, but not for observations (i.e. "traceFile", "meanVarFile", "statePosteriorsFile", "update", "propKernel" and "sharedJumpSize"). If a user tries to modify these for an observation, an error is thrown. Also note that the column "observation" is just for information. If a user attempts to change an observation to a parameter or vice versa, an error is thrown.

2. Check if a file with initial values and initial jump sizes has been defined on the command-line. This file must have the following format: 3 columns that are name of parameter, initial value and initial jump size. One row corresponds to one parameter definition. We parse all rows and re-set the definitions accordingly. Columns can also be empty (e.g. only specify initial value, but not initial jump size). Note that in here, only parameter definitions are currently allowed. If the initial values for an observation definition are modified, an error is thrown. The reason is that the initial values file is usually a file generated by a former run of the framework, i.e. if you want to restart the MCMC at a given position. In this file, we don't write observation values, as they never change during the MCMC. This is a design choice that can easily be changed, if this is necessary in the future.
3. Check if configs have been defined on the command-line directly. Specifically, we loop over all config-names (as above: "priorParameters", "traceFile", etc.), and check if a command-line argument matching the pattern `paramName.config` exists (e.g. `myParam.priorParameters=...`). If yes, we re-set the definition of this parameter/observation accordingly (note that this overrides specifications from the config file).
4. Check if initial values and/or initial jump sizes have been defined on the command-line directly. Specifically, we check if a command-line argument matching the pattern `paramName` and `paramName.jumpSize` exists. If yes, we re-set the definition accordingly (note that this overrides specifications from the init-file).

– We check for a valid DAG. A valid DAG must have 1) at least 1 observation and 2) no parameter at bottom of DAG (this must be an observation). We then initialize an instance of class `TDAG`. To illustrate this, consider the following complicated DAG, represented in Figure 5. Currently, we only have pointers to each parameter/observation in the DAG (note: we don't have pointers to the priors!). Parameters that belong to the same prior (e.g. b and c) must always be simulated/initialized/updated together. Imagine we would now treat each node of the DAG (i.e. each parameter and observation) independently. The problem then is that we will simulate/initialize/update shared prior parameters multiple times, which would be a waste of computational time (or even worse). For example, if we would tell first x1 to initialize its values, and then x2 to initialize its values, we would initialize b and c twice. We therefore need an unique path through a DAG, which we can follow while simulating/initializing, and which guarantees us to visit each node exactly once. To find such a path, we use the following recursive algorithm:

1. Create two vectors that store pointers to `TNodeTyped`: `final` and `tmp`.
2. Start at first observation at the bottom of the DAG.
3. Call function `constructDAG()` on current observation/parameter. This function first add the this-pointer (a pointer to `TNodeTyped`) to `tmp`.
4. The function then goes to the prior. The prior checks:
   (a) If all children (i.e. all parameters/observations that share this prior) are present in `tmp`, the prior adds the pointer to the first parameter/observation to `final`. Then, the prior will call `constructDAG()` of all its prior parameters, i.e. repeat steps 3) and 4).
   (b) Else (not all children are present in `tmp`): Do nothing. The recursion stops.
5. Re-start recursion at next observation at bottom of DAG and repeat steps 3) to 5) until all observations have been visited.

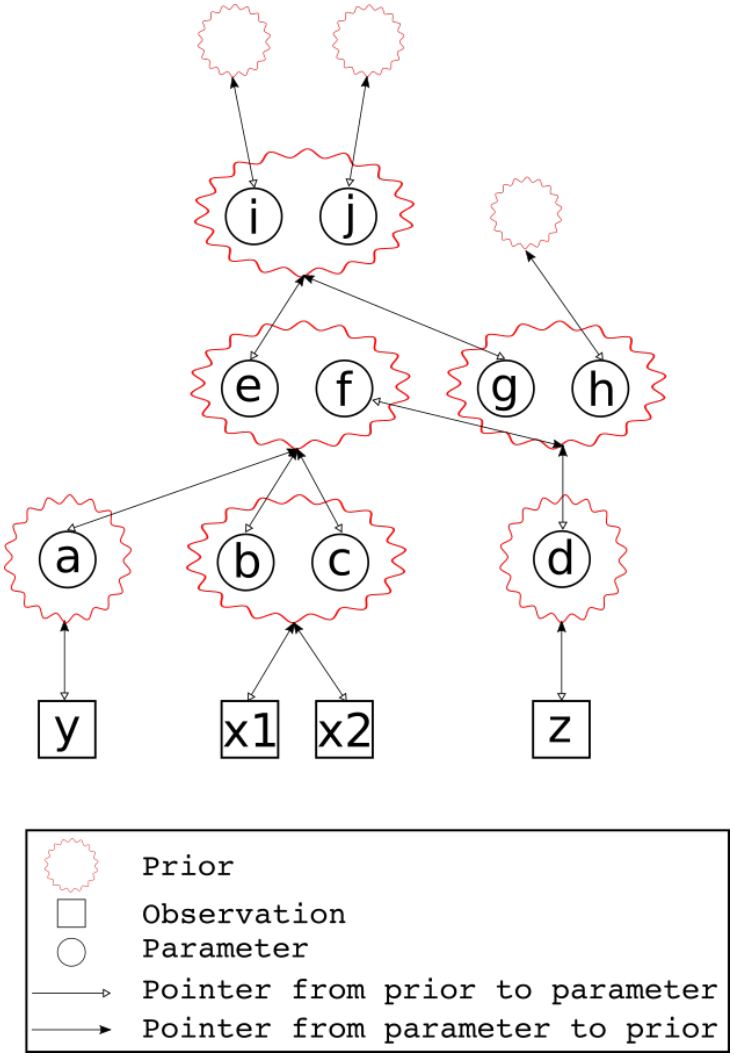

Figure 5: *DAG of a complicated model. We have multiple observations at the bottom of the DAG. A complicated hierarchical model is defined on these observations. We have some parameters/observations that share a common prior (e.g. observations x1/x2 share a prior, and parameters a,b,c also share a prior).*

The resulting vector (`final`) is stored inside class `TDAG`. For Figure 5, we obtain a vector $\{y, x_1, a, z, d, e, i, j, h\}$. Whenever we want to...

* Initialize values (with MLE/EM etc.): Go from left $\rightarrow$ right through the vector (i.e. from bottom to top through DAG). Call function `guessInitialValues()` on each element. This will re-direct to the prior, which will initialize its prior parameters.
* Simulate values: Go from right $\rightarrow$ left throught the vector (i.e. from top to bottom throught DAG). Call function `simulateUnderPrior()` on each element. This will re-direct to the prior, which will fill each of child (i.e. each parameter/observation on which this prior is defined) with random values from the prior distribution.

To update parameters, we use another vector of base class pointers. This is because i) for updating, the order does not matter and ii) we do not want to update deterministic parameters. The function `update()` will be called for each parameter present in this

vector.

- In the last step, the function `initializeStorage()` is called, that will go through the DAG and initialize the dimensions, storage and dimension names of all parameters (by comparing the expectations based on the prior with the developer-defined dimensions). This function can only be executed after all observations have been filled with data, since the dimensions of some parameters depend on the dimensions of the observations that only known after reading the data.

1.9 MCMC files

We distinguish four types of MCMC files:

1. Trace files. During the MCMC, every `_thinning`'st iteration, we write the parameter values to a trace file. The header contains the parameter name(s) (if the parameter is an array, it creates one column per element of the array). The rows are the values of this parameter per iteration.

2. MeanVar files. These contain the posterior mean and variance of the parameter and are written at the end of the MCMC. The header contains the parameter name(s) and there are two rows: mean and variance.

3. State posterior files. For categorical variables, these contain the posterior probability for each state, $\mathbb{P}(z = Z|\mathcal{D})$. This corresponds to the fraction of times the parameter took this value during the MCMC. The names of the trace, meanVar and statePosteriors file are given by the parameter definition. If multiple parameters share the same file, they will be written as adjacent columns. Note that all parameters that share a state posterior file must have the same number of states (otherwise the number of rows in the file don't match). This is checked upon construction.

4. State file. This file has three columns: name, value and jump size. It contains the current values and jump sizes for all parameters of a certain iteration. The idea is that this file is written every i^{th} iteration (and is overwritten every time), as well as after the last iteration. It can then be used as a init-file for re-starting another MCMC exactly at the location where the old one has stopped, e.g. if the program crashed or if the MCMC has not reached convergence.

Finally, we write all parameter configurations to a config-file. This file has the same format as the config-file that can be read when initializing the parameters, and can hence be used to re-start an MCMC with the same configurations as a previous MCMC.

1.10 The MCMC

Class `TMCMC` is the class that a developer needs to instantiate. A typical procedure would be:

1. Read the data into a `TMultiDimensionalStorage` object.

2. Create all parameters and observations, as well as their corresponding priors.

3. Create an instance of class `TMCMC`. Call function `runMCMC()`. This will first initialize all parameters with MLE/EM etc., and then run the MCMC algorithm.

General MCMC parameters are (specified over the command line):

- `_iterations`: The number of iterations for the MCMC.
- `_burnin`: The number of iterations for the burnin.
- `_numBurnin`: The number of burnins to run.
- `_thinningStateFile`: Write a state file every i^{th} iteration. By default, no state file is written.
- `_writeBurnin`: Write the trace after the initialization adn during the burnin, too.
- `_thinning`: Write to the trace file every i^{th} iteration.

1.11 Simulations

The function `simulate()` of class `TSimulator` simulates values under the prior distribution (top-down in DAG), also filling observations with random values. The simulated values will be written to file specified in the definition. Note that if values should be read in specific file format for observations, the developer will again need to get the pointer to the observation and write the values himself, because the framework does not know how to handle specific file formats. Parameters will only be simulated if no initial values have been set.

2 The EM algorithm

The expectation-maximization (EM) algorithm is an iterative algorithm that provides maximum-likelihood estimates of parameters in the presence of latent variables. Let us denote by $\boldsymbol{X}$ the observed data, by $\boldsymbol{\theta}$ the model parameters and by $\boldsymbol{z}$ the hidden (latent) variables. We will assume the most general model with K independent observations $\boldsymbol{X} = \boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(K)}$, where each $\boldsymbol{x}^{(k)} = x_1^{(k)}, \ldots, x_N^{(k)}$ is of length N with $k = 1, \ldots, K$.
The EM algorithm seeks to obtain parameter estimates of $\boldsymbol{\theta}$ that maximize the marginal likelihood $\mathcal{L}(\boldsymbol{\theta})$:

$$\mathcal{L}(\boldsymbol{\theta}) = \mathbb{P}(\boldsymbol{X}|\boldsymbol{\theta}) = \sum_{\boldsymbol{z}} \mathbb{P}(\boldsymbol{X}, \boldsymbol{z}|\boldsymbol{\theta}) = \sum_{\boldsymbol{z}} \mathcal{L}_c(\boldsymbol{\theta}).$$

Here, $\mathcal{L}_c(\boldsymbol{\theta})$ denotes the complete data likelihood that treats $\boldsymbol{z}$ as observed. The EM algorithm proceeds by iterating between two steps. In the expectation step (E-step), the Q-function

$$Q(\boldsymbol{\theta}|\boldsymbol{\theta}') = \mathbb{E}\left[\log \mathcal{L}_c(\boldsymbol{\theta})|\boldsymbol{X}, \boldsymbol{\theta}'\right] = \sum_{\boldsymbol{z}} \log \mathbb{P}(\boldsymbol{X}, \boldsymbol{z}|\boldsymbol{\theta}) \mathbb{P}(\boldsymbol{z}|\boldsymbol{X}, \boldsymbol{\theta}')$$

is calculated using the current parameter estimates $\boldsymbol{\theta}'$. This can be factorized to

$$Q(\boldsymbol{\theta}|\boldsymbol{\theta}') = \sum_{k=1}^{K} \sum_{i=1}^{N} \sum_{z=1}^{Z} \log \mathbb{P}(x_i^{(k)}, z|\boldsymbol{\theta}) \mathbb{P}(z|x_i^{(k)}, \boldsymbol{\theta}'), \tag{43}$$

where Z denotes the number of hidden states. The weights $\mathbb{P}(z|x_i^{(k)}, \boldsymbol{\theta}')$ can be calculated with the aid of Bayes' Theorem as

$$\mathbb{P}(z|x_i^{(k)}, \boldsymbol{\theta}') = \frac{\mathbb{P}(x_i^{(k)}, z|\boldsymbol{\theta}')}{\sum_{h=0}^{Z} \mathbb{P}(x_i^{(k)}, h|\boldsymbol{\theta}')}. \tag{44}$$

In the maximization step (M-step), the values $\boldsymbol{\theta}$ that maximize the Q-function are determined:

$$\tilde{\boldsymbol{\theta}} = \arg\max_{\boldsymbol{\theta}} Q(\boldsymbol{\theta}|\boldsymbol{\theta}'),$$

and these estimates $\tilde{\boldsymbol{\theta}}$ are then used as $\boldsymbol{\theta}'$ in the next EM iteration.
In many models, hierarchical parameters $\boldsymbol{\pi} \subseteq \boldsymbol{\theta}$ are used to model the distribution of the hidden states $\boldsymbol{z}$. The complete data likelihood $\mathcal{L}_c(\boldsymbol{\theta})$ can then be factorized as

$$\mathcal{L}_c(\boldsymbol{\theta}) = \mathbb{P}(\boldsymbol{X}, \boldsymbol{z}|\boldsymbol{\theta}) = \mathbb{P}(\boldsymbol{X}|\boldsymbol{z}, \boldsymbol{\phi})\mathbb{P}(\boldsymbol{z}|\boldsymbol{\pi}),$$

where $\boldsymbol{\phi} = \boldsymbol{\theta}_{-\boldsymbol{\pi}}$ denotes the parameter vector $\boldsymbol{\theta}$ without $\boldsymbol{\pi}$. In such cases, the Q-function from equation (43) can be written as

$$Q(\boldsymbol{\phi}, \boldsymbol{\pi}|\boldsymbol{\phi}', \boldsymbol{\pi}') = \sum_{k=1}^{K}\sum_{i=1}^{N}\sum_{z=1}^{Z}\Big(\log \mathbb{P}(x_i^{(k)}|z, \boldsymbol{\phi}) + \log \mathbb{P}(z|\boldsymbol{\pi})\Big)\, \mathbb{P}(z|x_i^{(k)}, \boldsymbol{\phi}', \boldsymbol{\pi}'),$$

and the corresponding EM weights $\mathbb{P}(z|x_i^{(k)}, \boldsymbol{\phi}', \boldsymbol{\pi}')$ can be calculated as

$$\mathbb{P}(z|x_i^{(k)}, \boldsymbol{\phi}', \boldsymbol{\pi}') = \frac{\mathbb{P}(x_i^{(k)}|z, \boldsymbol{\phi}')\mathbb{P}(z|\boldsymbol{\pi}')}{\sum_{h=0}^{Z}\mathbb{P}(x_i^{(k)}|h, \boldsymbol{\phi}')\mathbb{P}(h|\boldsymbol{\pi}')}. \tag{45}$$

`stattools` partitions the EM algorithm in three main classes. Class `TLatentVariableWithRep` represents the observation model $\mathbb{P}(x_i^{(k)}|z, \boldsymbol{\phi})$. Class `TEMPriorIndependent_base` represents the distribution of the hidden states $\mathbb{P}(z|\boldsymbol{\pi})$. And finally, class `TEM` runs the EM algorithm and coordinates the interaction between the latent variable and EM prior. In the following, these classes are described in more detail.

2.1 `TLatentVariableWithRep` and `TLatentVariable`

The class `TLatentVariableWithRep` represents the observational model and is in charge of calculating $\mathbb{P}(\boldsymbol{X}, \boldsymbol{z}|\boldsymbol{\phi})$. It is the most general representation of a latent variable model that also allows for multiple independent observations (replicates). In many applications, however, there is a single observation and the additional index k is superfluous. The class `TLatentVariable` inherits from `TLatentVariableWithRep` and provides a simpler interface without the index k. Apart from this, both classes follow the same logic and structure. In the following, we will describe the interface of `TLatentVariableWithRep`, as this is the more general case, but we note that the interface of `TLatentVariable` is the same except that all replicate indices k are omitted from the function definition.
Both classes are abstract base classes and define a single pure virtual function:

1. `calculateEmissionProbabilities(size_t i, size_t k, TDataVector v)`: This function calculates the probabilities $\mathbb{P}(x_i^{(k)}, z|\boldsymbol{\phi})$ at index `i` and replicate `k` for all possible hidden states $z = 1, \ldots, Z$ and stores these probabilities in the vector `v`.

A developer who wishes to implement the EM algorithm must therefore derive from `TLatentVariableWithRep` (if there are replicates) or `TLatentVariable` otherwise, and override the function `calculateEmissionProbabilities()`. Both classes provides more virtual functions that may be overridden if needed. There are three main use-cases that can be combined in any way: i) run EM, ii) calculate likelihood and iii) estimate state posteriors. The functions that need to be implemented for each use-case are described in the following three sections.

2.1.1 Run EM

If the goal is to obtain MLE estimates of $\boldsymbol{\phi}$, the following functions should be considered for overriding. Note that it might not be necessary to override all of them, e.g. if there is no memory to be allocated for storing EM results, some functions may be omitted. We will describe these functions in the order they are called.
Before the EM starts, these two functions are called:

1. `prepareEMParameterEstimationInitial()`: Initialize temporary values or allocate memory needed to store the EM weights (only if necessary).

2. `initializeEMParameters()`: Estimate an initial guess of $\boldsymbol{\phi}$.

Then, in each EM iteration, these functions are called:

1. `prepareEMParameterEstimationOneIteration()`: Called right in the beginning of the iteration. This function should reset temporary values for the new iteration (only if necessary).

2. `handleEMParameterEstimationOneIteration(size_t i, size_t k, TDataVector v)`: The vector `v` contains the EM weights $\mathbb{P}(z|x_i^{(k)}, \boldsymbol{\phi}', \boldsymbol{\pi}')$ that were calculated within `stattools` according to equation (45). Depending on the M-step, the developer will need to either store the weights or sum them up.

3. `finalizeEMParameterEstimationOneIteration()`: Called at the end of the iteration. This function should perform the M-step.

4. `reportEMParameters()`: Called at the end of the iteration. Provides the possibility to report the updated parameters $\boldsymbol{\phi}$ to the logfile.

At the very end of the EM, this function is called:

1. `finalizeEMParameterEstimationFinal()`: Clear temporary variables or memory (only if necessary).

2.1.2 Calculate likelihood

To calculate the likelihood, the following functions should be considered for overriding.

1. `prepareLLCalculation()`: Initialize temporary values or allocate memory needed to store the EM weights (only if necessary).

2. `finalizeLLCalculation()`: Clear temporary variables or memory (only if necessary).

2.1.3 Estimate state posterior probabilities

To estimate the state posterior probabilities $\mathbb{P}(z|\boldsymbol{X}, \boldsymbol{\phi}, \boldsymbol{\pi})$ for all $z = 1, \ldots, Z$, the following functions should be considered for overriding.

1. `prepareStatePosteriorEstimation()`: Initialize temporary values or allocate memory needed to store the EM weights (only if necessary).

2. `handleStatePosteriorEstimation(size_t i, size_t k, TDataVector v)`: The vector `v` contains the state posterior probabilities (i.e. the EM weights) $\mathbb{P}(z|x_i^{(k)}, \boldsymbol{\phi}', \boldsymbol{\pi}')$ that were calculated within `stattools` according to equation (45). Depending on the objective, these probabilities can be stored or processed.

3. `finalizeStatePosteriorEstimation()`: Clear temporary variables or memory (only if necessary).

A common use-case is to write the state posterior probabilities to a file. In such a case, the base class `TLatentVariableWithRep` already implements the following functions:

1. `prepareStatePosteriorEstimationAndWriting()`: Opens file and writes header.
2. `handleStatePosteriorEstimationAndWriting()`: Writes state posterior probabilities $\mathbb{P}(z|x_i^{(k)}, \boldsymbol{\phi}', \boldsymbol{\pi}')$ to file.
3. `finalizeStatePosteriorEstimationAndWriting()`: Clear temporary variables or memory (only if necessary).

If necessary, these functions can be overridden by the developer to write custom files.

2.2 `TEMPriorIndependent_base`

The class `TEMPriorIndependent_base` represents the hierarchical model on the hidden states $\boldsymbol{z}$ and is in charge of calculating $\mathbb{P}(\boldsymbol{z}|\boldsymbol{\pi})$. It is an abstract base class and defines a single pure virtual function:

1. `operator(size_t i, size_t z)`: This function calculates the probability $\mathbb{P}(z_i|\boldsymbol{\pi})$ at index `i`

A developer who whishes to implement the EM algorithm must therefore derive from `TEMPriorIndependent_base` and override the function `operator()`. The class provides more virtual functions that may be overridden if needed. If the goal is to obtain MLE estimates of $\boldsymbol{\pi}$, the following functions should be considered for overriding.
Before the EM starts, the following functions are called:

1. `prepareEMParameterEstimationInitial()`: Initialize temporary values or allocate memory needed to store the EM weights (only if necessary).
2. `initializeEMParameters()`: Estimate an initial guess of $\boldsymbol{\pi}$. By default, the initial guess is obtained by one full loop over all $\boldsymbol{X}$, calculating the maximum likelihood state $\hat{z}$ for each index i and calling `handleEMParameterInitialization()` with this state.
3. `handleEMParameterInitialization(size_t i, size_t z)`: Store or sum the guess of the hidden state `z`.
4. `finalizeEMParameterInitialization()`: Get an initial guess for $\boldsymbol{\pi}$ based on the guesses of the hidden states $\boldsymbol{z}$ obtained by the previous function.

Then, in each EM iteration, these functions are called:

1. `prepareEMParameterEstimationOneIteration()`: Called right in the beginning of the iteration. This function should reset temporary values for the new iteration (only if necessary).
2. `handleEMParameterEstimationOneIteration(size_t i, TDataVector v)`: The vector `v` contains the EM weights $\mathbb{P}(z|x_i^{(k)}, \boldsymbol{\phi}', \boldsymbol{\pi}')$ that were calculated within `stattools` according to equation (45). Depending on the M-step, the developer will need to either store the weights or sum them up.

3. `finalizeEMParameterEstimationOneIteration()`: Called at the end of the iteration. This function should perform the M-step.

4. `reportEMParameters()`: Called at the end of the iteration. Provides the possibility to report the updated parameters $\boldsymbol{\phi}$ to the logfile.

At the very end of the EM, this function is called:

1. `finalizeEMParameterEstimationFinal()`: Clear temporary variables or memory (only if necessary).

A typical EM prior is the categorical distribution, as described in section 1.5.3.10. The prior `prior::TCategoricalInferred` inherits from `TEMPriorIndependent_base` and and can be readily used for any latent variable model where $z|\boldsymbol{\pi} \sim \text{Categorical}(\boldsymbol{\pi})$.

2.3 `TEM`

Class `TEM` runs the EM algorithm and the coordinates the interaction between classes `TLatentVariableWithRep` and `TEMPriorIndependent_base`. In line with the use-cases described before, it provides three main functionalities:

1. `runEM()`: Run the EM algorithm to find the ML estimates of $\boldsymbol{\phi}$ and $\boldsymbol{\pi}$.

2. `calculateLL()`: Calculate the marginal likelihood $\mathcal{L}(\boldsymbol{\phi}, \boldsymbol{\pi})$ given fixed estimates $\boldsymbol{\phi}$ and $\boldsymbol{\pi}$.

3. `estimateStatePosteriors()`: Estimate the state posterior probabilities $\mathbb{P}(z|\boldsymbol{X}, \boldsymbol{\phi}, \boldsymbol{\pi})$ for all states $z = 1, \ldots, Z$ given fixed estimates $\boldsymbol{\phi}$ and $\boldsymbol{\pi}$. If called with a filename, the state posterior probabilities will automatically be written to a file.

The EM algorithm will iterate until i) the difference in log-likelihoods between two iterations is smaller than some threshold (default 10^{-4}) or until ii) the maximum number of iterations is reached.

2.3.1 SQUAREM

The EM algorithm is very stable and guarantees a monotonic increase of the likelihood. However, the EM algorithm is also infamous for its slow, linear convergence, which can result in very long run times, especially for high-dimensional problems. Multiple acceleration schemes have been proposed. `stattools` implements the SQUAREM method (Varadhan and Roland, 2008), which provides an off-the-shelf solution applicable to any model. The SQUAREM algorithm in `stattools` is adapted from (Varadhan and Roland, 2008; Du and Varadhan, 2020), and one SQUAREM cycle runs as described in algorithm 1. The SQUAREM cycles are repeated until convergence or when a maximum number of iterations is reached. The SQUAREM is implemented within `TEM` and is called through the function `runSQUAREM()`.
It is well possible that the SQUAREM proposes non-valid parameter values (e.g. negative values for parameters that need to be positive). It therefore rejects all updates that result in invalid parameters.
The SQUAREM algorithm requires direct access to get and set the EM parameters in both latent variable and EM prior. If a SQUAREM is used, both the latent variable and EM prior class must override the following functions:

- `getParameters()`: Return the current values of all EM parameters $\boldsymbol{\theta}$.

Algorithm 1 One SQUAREM cycle

$\boldsymbol{\theta}_1 \leftarrow \text{EMupdate}(\boldsymbol{\theta}_0)$
if converged **then**
  stop
end if
$\boldsymbol{\theta}_2 \leftarrow \text{EMupdate}(\boldsymbol{\theta}_1)$
if converged **then**
  stop
end if
$L_{\boldsymbol{\theta}_2} \leftarrow \text{CalculateLL}(\boldsymbol{\theta}_2)$
$\boldsymbol{r} = \boldsymbol{\theta}_1 - \boldsymbol{\theta}_0$
$\boldsymbol{v} = (\boldsymbol{\theta}_2 - \boldsymbol{\theta}_1) - \boldsymbol{r}$
$\alpha = -\frac{||\boldsymbol{r}||}{||\boldsymbol{v}||}$
$\boldsymbol{\theta}_{sq} = \boldsymbol{\theta}_0 - 2\alpha\boldsymbol{r} + \alpha^2\boldsymbol{v}$
if $\boldsymbol{\theta}_{sq}$ is not valid **then**
  $\boldsymbol{\theta}_0 \leftarrow \boldsymbol{\theta}_2$
  continue to next iteration
end if
$L_{\boldsymbol{\theta}_{sq}} \leftarrow \text{CalculateLL}(\boldsymbol{\theta}_{sq})$
if $L_{\boldsymbol{\theta}_{sq}} > L_{\boldsymbol{\theta}_2}$ **then**
  $\boldsymbol{\theta}_0 \leftarrow \text{EMupdate}(\boldsymbol{\theta}_{sq})$
else
  $\boldsymbol{\theta}_0 \leftarrow \boldsymbol{\theta}_2$
end if
continue to next iteration

- `setParameters(coretools::TConstView<PrecisionType> Params)`: Set all EM parameters $\boldsymbol{\theta}$ according to values given in `Params`. Must return `true` if these parameters are valid, and `false` otherwise.

If the SQUAREM is used, but these functions are not implemented, an error is thrown.

3 Hidden Markov Models

A Hidden Markov Model (HMM) assumes that a hidden, underlying Markov model emits noisy observations. The states $\boldsymbol{z} = (z_0, \dots, z_N)$ of the Markov model can thus not be observed directly, but only inferred through the data $\boldsymbol{x} = (x_0, \dots, x_N)$. We will assume that the hidden states are discrete and denote by Z the number of hidden states such that $z_i \in 1, \dots, Z$ for all $i = 1, \dots, N$. A HMM is characterized by three probabilities: the transition probabilities, the initial state probability and the emission probabilities. In the following sections, I will describe those probabilities and how they are modelled within `stattools`.

3.1 Transition probabilities

The transition probabilities $\mathbb{P}(z_i|z_{i-1})$ model the Markov assumption of the hidden states. Transition probabilities can be organized into a transition matrix $\boldsymbol{Q}$ whose element $[\boldsymbol{Q}]_{jk}$ represents the probability of moving from state j to state k. The transition probabilities may be either parametrized directly or through a transition rate matrix. A transition rate matrix $\boldsymbol{\Lambda}$ is an

infinitesimal generator of a transition matrix and is defined such that the rows sum to zero, $[\boldsymbol{\Lambda}]_{jj} = -\sum_{k \neq j}[\boldsymbol{\Lambda}]_{jk}$, while $[\boldsymbol{\Lambda}]_{jk} \geq 0$ for $j \neq k$. The transition matrix is then given by the matrix exponential of the transition rate matrix, $\boldsymbol{Q} = e^{\boldsymbol{\Lambda}}$ (Suhov and Kelbert, 2008). However, it generally makes sense to assume that the transition probabilities are affected by the distance δ_i between two neighbouring observations i and $i-1$. A classic example is the distance between two genetic markers (in space) or the distance between two recordings (in time). We therefore define $\boldsymbol{Q}(\delta_i)$ for a distance δ_i as

$$\boldsymbol{Q}(\delta_i) = e^{\delta_i \boldsymbol{\Lambda}}.$$

Since the number of possible distances is large and the computation of the transition probabilities for a specific distance expensive, we group the distances δ_i into $E+1$ ensembles such that

$$e_i = \lceil \log_2 \delta_i + 1 \rceil - 1, e_i = 0, \ldots, E, \tag{46}$$

for every data point i. By using the $\log_2$-transformation, the bin size scales with the distance: all points with distance 1 go in the first group, all points with distance 2-3 into the second, all points with distance 4-7 in the third etc. This is implemented in class `genome-tools::TDistancesBinned`.

We then use the same transition matrix $\boldsymbol{Q}(2^e)$ for all data points in ensemble e. We thus only need to calculate the transition matrix for the first ensemble $e = 0$ where $\boldsymbol{Q}(2^0) = \boldsymbol{Q}(1)$, which corresponds to the transition matrix for a distance of $\delta = 1$. The transition matrices for all other ensembles can then be obtained through the recursion $\boldsymbol{Q}(e) = \boldsymbol{Q}(e-1)^2$, which is relatively cheap to calculate.

In the following sections, we will describe the transition matrices that are implemented in `stat-tools`.

3.1.1 `TTransitionMatrixBool`

This transition matrix generalizes the concept of a Bernoulli prior to a Markov chain by defining two parameters:

- π: The stationary probability for $z = 1$. This is equivalent to the success probability of the Bernoulli prior.
- γ: Modulates the expected peak width, i.e. the amount of clustering.

Our transition matrix is:

$$\boldsymbol{Q} = \frac{1}{1+\gamma} \begin{pmatrix} 1-\pi+\gamma & \pi \\ 1-\pi & \pi+\gamma \end{pmatrix}.$$

The stationary probability then is exactly π:

$$\frac{\pi}{\pi + 1 - \pi} = \pi.$$

The parameter γ modulates the expected peak width. For $\gamma = 0$, the transition matrix is:

$$\boldsymbol{Q} = \begin{pmatrix} 1-\pi & \pi \\ 1-\pi & \pi \end{pmatrix}.$$

meaning that the probability to be in a specific state z_i is completely independent of the previous state z_{i-1}: this corresponds to a "memory-less" Bernoulli process. For any $\gamma > 0$, the expected peak width increases relative to the peak width under a Bernoulli process. In fact, the expected peak length is given by

$$\mathbb{E}(\text{peak width}) = \frac{1}{Q_{1\to 0}} = \frac{1+\gamma}{1-\pi},$$

where the peak width under the Bernoulli model is simply $\frac{1}{1-\pi}$.

3.1.2 `TTransitionMatrixCategorical`

This transition matrix generalizes the concept of a categorical prior to a Markov chain by defining three parameters:

- π: The stationary probability for $z \neq 0$.
- γ: Modulates the expected peak width, i.e. the amount of clustering.
- $\boldsymbol{\rho} = (\rho_1, \ldots, \rho_Z)$: Model the probability to jump into a certain state z.

Our goal is to define a transition matrix of form

$$\boldsymbol{Q} = \begin{pmatrix} 1 - \sum_{z=1}^{Z} \lambda_z & \lambda_1 & \lambda_2 & \ldots & \lambda_Z \\ \lambda_0 & 1 - \lambda_0 & 0 & \ldots & 0 \\ \lambda_0 & 0 & 1 - \lambda_0 & \ldots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \lambda_0 & 0 & 0 & \ldots & 1 - \lambda_0 \end{pmatrix}.$$

as a function of parameters π, γ and ρ. The stationary distribution of this transition matrix can be derived from the following system of linear equations:

$$\mathbb{P}_0 = \left(1 - \sum_{z=1}^{Z} \lambda_z\right) \mathbb{P}_0 + \lambda_0 \sum_{z=1}^{Z} \mathbb{P}_z$$

and

$$\mathbb{P}_0 + \sum_{z=1}^{Z} \mathbb{P}_z = 1,$$

which results in

$$\pi = \sum_{z=1}^{Z} \mathbb{P}_z = \frac{\sum_{z=1}^{Z} \lambda_z}{\lambda_0 + \sum_{z=1}^{Z} \lambda_z}. \tag{47}$$

We have thus defined π as the stationary distribution to be in a non-zero state.
We note that the expected peak width for this transition matrix is:

$$\mathbb{E}(\text{peak width}) = \frac{1}{Z} \sum_{z=1}^{Z} \frac{1}{Q_{z \to 0}} = \frac{1}{\lambda_0},$$

whereas the expected peak width for a Bernoulli process is $\frac{1}{1-\pi}$. We now parametrize our transition matrix such that the peak width corresponds to the Bernoulli process if $\gamma = 0$, but results in more clustering if $\gamma > 0$:

$$\mathbb{E}(\text{peak width}) = \frac{1 + \gamma}{1 - \pi}.$$

Solving for λ_0 results in

$$\lambda_0 = \frac{1 - \pi}{1 + \gamma}.$$

With these equations at hand, we readily define $\sum_{z=1}^{Z} \lambda_z$ as:

$$\sum_{z=1}^{Z} \lambda_z = \frac{\pi}{1 + \gamma}.$$

So far, our parametrization acts on all λ_z jointly, as we defined π and γ on $\sum_{z=1}^{Z} \lambda_z$. In addition, we now want the model the probability to jump into a certain state z by a parameter ρ_z such that

$$\lambda_z = \frac{\rho_z \pi}{1+\gamma},$$

where we have the constraint that $\sum_{z=1}^{Z} \rho_z = 1$.
Our transition matrix then is:

$$\boldsymbol{Q} = \frac{1}{1+\gamma} \begin{pmatrix} 1+\gamma-\pi & \rho_1\pi & \rho_2\pi & \dots & \rho_Z\pi \\ (1-\pi) & \gamma+\pi & 0 & \dots & 0 \\ (1-\pi) & 0 & \gamma+\pi & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ (1-\pi) & 0 & 0 & \dots & \gamma+\pi \end{pmatrix}.$$

3.1.3 `TTransitionMatrixBoolGeneratingMatrix`

This transition matrix is defined through a 2×2 transition rate matrix $\boldsymbol{\Lambda}$, which is parametrized by λ_1 and λ_2:

$$\boldsymbol{\Lambda} = \begin{pmatrix} -\lambda_1 & \lambda_1 \\ \lambda_2 & -\lambda_2 \end{pmatrix}. \tag{48}$$

The stationary probabilities are calculated with

$$\mathbb{P}(z_0 = 0|\lambda_1, \lambda_2) = \frac{\lambda_2}{\lambda_1 + \lambda_2}$$

and

$$\mathbb{P}(z_0 = 1|\lambda_1, \lambda_2) = 1 - \mathbb{P}(z_0 = 0|\lambda_1, \lambda_2).$$

This transition rate matrix is implemented in the class `TGeneratingMatrixBool`. As described above, the transition matrix for a distance $\delta = 1$ is given by the matrix exponential of the generating matrix.

3.1.4 `TTransitionMatrixLadder`

This transition matrix is defined through a $Z \times Z$ ladder-type transition rate matrix $\boldsymbol{\Lambda}$, which is parametrized by κ:

$$\boldsymbol{\Lambda} = \kappa \cdot \begin{pmatrix} -1 & 1 & 0 & \dots & 0 & 0 \\ 1 & -2 & 1 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & 1 & -1 \end{pmatrix}.$$

The stationary probabilities are simply $\frac{1}{Z}$ for each state.
This transition rate matrix is implemented in the class `TGeneratingMatrixLadder`. As described above, the transition matrix for a distance $\delta = 1$ is given by the matrix exponential of the generating matrix.

3.1.5 `TTransitionMatrixScaledLadder`

This transition matrix is defined through a $Z \times Z$ ladder-type transition rate matrix $\mathbf{\Lambda}$ with an attractor state, which is parametrized by κ, ν and μ (Galimberti et al., 2020):

$$\mathbf{\Lambda} = \kappa \cdot \begin{pmatrix} -1 & 1 & 0 & 0 & \dots & 0 & 0 & 0 & 0 \\ \mu & -1-\mu & 1 & 0 & \dots & 0 & 0 & 0 & 0 \\ 0 & \mu & -1-\mu & 1 & \dots & 0 & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \dots & 1 & -1-\mu & \mu & 0 \\ 0 & 0 & 0 & 0 & \dots & 0 & 1 & -1-\mu & \mu \\ 0 & 0 & 0 & 0 & \dots & 0 & 0 & 1 & -1 \end{pmatrix},$$

where the middle row of this matrix (the attractor) is given by

$$\begin{pmatrix} 0 & \dots & 0 & \nu\mu & -2\nu\mu & \nu\mu & 0 & \dots & 0 \end{pmatrix}.$$

The parameter μ models the rate of moving towards the peripheral states of the matrix. If $\mu \approx 1$, it is equally likely up or down, whereas if $\mu \approx 0$, it is very unlikely to move towards the periphery of the matrix. The parameter ν models the rate of leaving the attractor. If $\nu \approx 1$, it is likely to leave the attractor, whereas if $\nu \approx 0$, it is very unlikely to leave the attractor.
The stationary distribution for this Markov Chain can be calculated directly from the parameters of the generator matrix as:

$$\Pi = c \cdot \begin{pmatrix} 1 & \frac{1}{\mu} & \frac{1}{\mu^2} & \dots & \frac{1}{\mu^{T-1}} & \frac{1}{\mu^T \nu} & \frac{1}{\mu^{T-1}} & \dots & \frac{1}{\mu^2} & \frac{1}{\mu} & 1 \end{pmatrix},$$

where

$$c = \frac{(\mu - 1)\nu\mu^T}{2\nu\mu(\mu^T - 1) + \mu - 1}.$$

Here, $T = \frac{Z-1}{2}$ denotes the number of states on one side of the attractor state, where we enforce that Z is an odd number.
If $\mu = 1$, this equation results in a division by zero. For this case, we use the following equation to calculate the stationary:

$$\Pi = c \cdot \begin{pmatrix} \nu & \nu & \nu & \dots & \nu & 1 & \nu & \dots & \nu & \nu & \nu \end{pmatrix},$$

where

$$c = \frac{1}{Z\nu - \nu + 1}.$$

κ, μ and ν all must be strictly positive. However, only if μ and ν are in (0,1], the attractor state is actually attracting. We therefore constrain μ and ν to [0,1] by running optimization in the logistic space.
This transition rate matrix is implemented in the class `TGeneratingMatrixScaledLadder`. As described above, the transition matrix for a distance $\delta = 1$ is given by the matrix exponential of the generating matrix.

3.1.6 `TTransitionMatrixScaledLadderAttractorShift`

This transition matrix is defined through a $Z \times Z$ ladder-type transition rate matrix $\mathbf{\Lambda}$ with an attractor state, which is parametrized by κ, ν and μ. This parametrization is exactly equivalent to the previous one (`TGeneratingMatrixScaledLadder`) with the only difference that the attractor

row does not need to be in the middle row. There is thus an extra parameter ix that corresponds to the row index of the attractor (0 corresponding to the top row, 1 to the second row etc).
If there are only two states, the transition rate matrix $\mathbf{\Lambda}$ is given by:

$$\mathbf{\Lambda} = \begin{cases} \kappa \cdot \begin{pmatrix} -\nu & \nu \\ 1 & -1 \end{pmatrix} & \text{if } ix = 0 \\ \kappa \cdot \begin{pmatrix} -1 & 1 \\ \nu & -\nu \end{pmatrix} & \text{if } ix = 1. \end{cases}$$

Note that in this case, there is no μ, since the matrix would be over-parametrized.
If there are three states, the transition rate matrix $\mathbf{\Lambda}$ is given by:

$$\mathbf{\Lambda} = \begin{cases} \kappa \cdot \begin{pmatrix} -\nu\mu & \nu\mu & 0 \\ 1 & -1-\mu & \mu \\ 0 & 1 & -1 \end{pmatrix} & \text{if } ix = 0 \\ \kappa \cdot \begin{pmatrix} -1 & 1 & 0 \\ \nu & -2\nu & \nu \\ 0 & 1 & -1 \end{pmatrix} & \text{if } ix = 1 \\ \kappa \cdot \begin{pmatrix} -1 & 1 & 0 \\ \mu & -1-\mu & 1 \\ 0 & \nu\mu & -\nu\mu \end{pmatrix} & \text{if } ix = 2. \end{cases}$$

Note that in the case of three states and ix=1, there is no μ since also here, μ and ν are non-identifiable.
The stationary is complicated to calculate explicitly. We instead solve a system of normal equations, as explained in section 3.2.
This transition rate matrix is implemented in the class `TGeneratingMatrixScaledLadderAttractorShift`. As described above, the transition matrix for a distance $\delta = 1$ is given by the matrix exponential of the generating matrix.

3.1.7 `TTransitionMatrixScaledLadderAttractorShift2`

This transition matrix is defined through a $Z \times Z$ ladder-type transition rate matrix $\mathbf{\Lambda}$ with an attractor state, which is parametrized by κ, ν and μ as well as a index ix that corresponds to the row index of the attractor (0 corresponding to the top row, 1 to the second row etc.). This parametrization is similar to the previous one (`TGeneratingMatrixScaledLadderAttractorShift`), with the exception that the attractor row is parametrized only by ν (without μ) to improve convergence.
The transition rate matrix $\mathbf{\Lambda}$ is given by:

$$\mathbf{\Lambda} = \kappa \cdot \begin{pmatrix} -1 & 1 & 0 & 0 & \dots & 0 & 0 & 0 & 0 \\ 1-\mu & -2 & 1+\mu & 0 & \dots & 0 & 0 & 0 & 0 \\ 0 & 1-\mu & -2 & 1+\mu & \dots & 0 & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \dots & 1+\mu & -2 & 1-\mu & 0 \\ 0 & 0 & 0 & 0 & \dots & 0 & 1+\mu & -2 & 1-\mu \\ 0 & 0 & 0 & 0 & \dots & 0 & 0 & 1 & -1 \end{pmatrix},$$

where the attractor row is

$$(\; 0 \quad \ldots \quad 0 \quad \nu \quad -2\nu \quad \nu \quad 0 \quad \ldots \quad 0) \; .$$

The parameter μ models the rate of moving away from the periphery towards the attractor. If $\mu \approx 0$, it's equally likely to move up or down. If $\mu \approx 1$, it is much more likely to move towards the attractor. The parameter ν models the rate of leaving the attractor. If $\nu \approx 0$, it's very unlikely to leave the attractor. If $\nu \approx 1$, it becomes more and more likely to leave the attractor.
If there are only two states, the transition rate matrix $\mathbf{\Lambda}$ is given by:

$$\mathbf{\Lambda} = \begin{cases} \kappa \cdot \begin{pmatrix} -\nu & \nu \\ 1 & -1 \end{pmatrix} & \text{if } ix = 0 \\ \kappa \cdot \begin{pmatrix} -1 & 1 \\ \nu & -\nu \end{pmatrix} & \text{if } ix = 1. \end{cases}$$

Note that in this case, there is no μ, since the matrix would be over-parametrized.
If there are three states, the transition rate matrix $\mathbf{\Lambda}$ is given by:

$$\mathbf{\Lambda} = \begin{cases} \kappa \cdot \begin{pmatrix} -\nu & \nu & 0 \\ 1+\mu & -2 & 1-\mu \\ 0 & 1 & -1 \end{pmatrix} & \text{if } ix = 0 \\ \kappa \cdot \begin{pmatrix} -1 & 1 & 0 \\ \nu & -2\nu & \nu \\ 0 & 1 & -1 \end{pmatrix} & \text{if } ix = 1 \\ \kappa \cdot \begin{pmatrix} -1 & 1 & 0 \\ 1-\mu & -2 & 1+\mu \\ 0 & \nu & -\nu \end{pmatrix} & \text{if } ix = 2. \end{cases}$$

Note that in the case of three states and ix=1, there is no μ.
The stationary is complicated to calculate explicitly. We instead solve a system of normal equations, as explained in section 3.2.
This transition rate matrix is implemented in the class `TGeneratingMatrixScaledLadderAttractorShift2`. As described above, the transition matrix for a distance $\delta = 1$ is given by the matrix exponential of the generating matrix.

3.2 Initial state distribution

The initial state distribution $\mathbb{P}(z_0)$ models the distribution of the first hidden state z_0. It often makes sense to assume that the initial state distribution follows the stationary distribution of the Markov chain, which is given by the transition matrix $\boldsymbol{Q}$. Depending on the parametrization of $\boldsymbol{Q}$, the stationary distribution may be calculated explicitly, as it is the case for the transition matrices described in sections 3.1.1, 3.1.2, 3.1.3, 3.1.4 and 3.1.5.
If this is not possible, the stationary distribution can be calculated from the transition matrix $\boldsymbol{Q}$ using normal equations as follows. We are attempting to find a solution for

$$\boldsymbol{\pi} = \boldsymbol{\pi}\boldsymbol{Q},$$

such that

$$\sum_{z=1}^{Z} \pi_z = 1.$$

We can re-arrange the above expressions to

$$\begin{aligned} \boldsymbol{\pi} - \boldsymbol{\pi}\boldsymbol{Q} &= \mathbf{0} \\ \boldsymbol{\pi}(\boldsymbol{I} - \boldsymbol{Q}) &= \mathbf{0}. \end{aligned}$$

We can add the constraint that all $\boldsymbol{\pi}$ must sum to one by adding an extra column of 1's to $(\boldsymbol{I} - \boldsymbol{Q})$ and an extra 1 to the zero vector (call this new vector $\boldsymbol{b}$). To simplify the notation, we will write $\boldsymbol{\pi}$ as a column vector and hence transpose $\boldsymbol{I} - \boldsymbol{Q}$ (call this new matrix $\boldsymbol{A}$):

$$\begin{pmatrix} 1 - Q_{11} & -Q_{21} & \dots & -Q_{Z1} \\ -Q_{12} & 1 - Q_{22} & \dots & -Q_{Z2} \\ \vdots & \vdots & \ddots & \vdots \\ -Q_{1Z} & -Q_{2Z} & \dots & 1 - Q_{ZZ} \\ 1 & 1 & \dots & 1 \end{pmatrix} \begin{pmatrix} \pi_1 \\ \pi_2 \\ \vdots \\ \pi_Z \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 1 \end{pmatrix}$$

We now solve $\boldsymbol{A}\boldsymbol{\pi} = \boldsymbol{b}$, which is overdetermined, and can be solved with the normal equation

$$\boldsymbol{A}^T\boldsymbol{A}\boldsymbol{\pi} = \boldsymbol{A}^T\boldsymbol{b}.$$

This is achieved using the matrix library Armadillo (Sanderson and Curtin, 2016; Sanderson and Curtin, 2019).

3.3 Emission probabilities

The emission probabilities $\mathbb{P}(x_i^{(k)}|z_i)$ represent the observational model. In fact, the emission probabilities by themselves are "ignorant" of the Markov assumption of the hidden states $\boldsymbol{z}$ and as such not different from any other latent variable model. In `stattools`, they are simply latent variables deriving from the class `TLatentVariableWithRep`, as described in 2.1, regardless if the latent variable has a Markov property or not.

3.4 The forward-backward algorithm

Given the transition probabilities $\mathbb{P}(z_i|z_{i-1})$, the initial state probability $\mathbb{P}(z_0)$ and the emission probabilities $\mathbb{P}(x_i|z_i)$, we can define the joint distribution of the HMM as

$$\mathbb{P}(\boldsymbol{z}, \boldsymbol{x}) = \mathbb{P}(z_0) \prod_{i=1}^{N} \mathbb{P}(z_i|z_{i-1}) \prod_{i=0}^{N} \mathbb{P}(x_i|z_i).$$

Note that this is easily extendable to multiple K independent observations. For simpler notation, we will omit the index $x^{(k)}$ in the following equations.
A major goal is to estimate $\mathbb{P}(z_i|\boldsymbol{x})$ for all data points i, which is the state posterior probability of the hidden state z_i given all the data $\boldsymbol{x}$. This can be calculated efficiently with the forward-backward algorithm, which relies on the fact that by conditioning on the state z_i at index i, we can break the chain into two parts, the past and the future. This allows us to factorize the joint distribution to

$$\mathbb{P}(z_i, \boldsymbol{x}_{0:N}) = \mathbb{P}(z_i, \boldsymbol{x}_{0:i})\mathbb{P}(\boldsymbol{x}_{i+1:N}|z_i) := \alpha_i(z_i)\beta_i(z_i).$$

Here, $\alpha_i(z_i) := \mathbb{P}(z_i, \boldsymbol{x}_{0:i})$ is the probability to observe the past data up index i and to be in state z_i, and $\beta_i(z_i) := \mathbb{P}(\boldsymbol{x}_{i+1:N}|z_i)$ is the probability of all future data given that the hidden state at index i is z_i. The state posterior probability $\mathbb{P}(z_i|\boldsymbol{x}_{0:N})$ is then given by:

$$\mathbb{P}(z_i|\boldsymbol{x}_{0:N}) = \frac{\alpha_i(z_i)\beta_i(z_i)}{\sum_{z_i}\alpha_i(z_i)\beta_i(z_i)}. \tag{49}$$

This state posterior probability is commonly denoted by $\gamma_i(z_i) = \mathbb{P}(z_i|\boldsymbol{x}_{0:N})$. In `stattools`, this equation in implemented in class `THMMPosteriorGamma`.
Both $\alpha_i(z_i)$ and $\beta_i(z_i)$ can be calculated efficiently with recursions.

3.4.1 The forward recursion

The forward probabilities $\alpha_i(z_i)$ can be calculated efficiently by starting with $\alpha_0(z_0) = \mathbb{P}(x_0|z_0)\mathbb{P}(z_0)$ and then propagating to the index on the right where $\alpha_i(z_i)$ is given by

$$\alpha_i(z_i) = \mathbb{P}(x_i|z_i)\sum_{z_{i-1}=1}^{Z}\mathbb{P}(z_i|z_{i-1})\alpha_{i-1}(z_{i-1}).$$

Since these probabilities can get very small, we normalize them at each step:

$$\mathbb{P}(z_i|\boldsymbol{x}_{0:i}) = \frac{\alpha_i(z_i)}{\sum_{z_i}\alpha_i(z_i)}.$$

The forward recursion is implemented in the `stattools` class `THMMForwardAlpha`.

3.4.2 The backward recursion

The backward probabilities $\beta_i(z_i)$ can be calculated efficiently by starting at $\beta(z_N) = 1$ and then propagating to the index on the left where $\beta_i(z_i)$ is given by

$$\beta_i(z_i) = \sum_{z_{i+1}=1}^{Z}\mathbb{P}(z_{i+1}|z_i)\mathbb{P}(x_{i+1}|z_{i+1})\beta(z_{i+1}),$$

As above, we normalize the probabilities at each step.
The backward recursion is implemented in the `stattools` class `THMMBackwardBeta`.

3.5 The Baum-Welch algorithm

The Baum-Welch algorithm is a special case of the expectation-maximization (EM) algorithm used for maximum likelihood estimation in HMMs. The following description of the Baum-Welch algorithm is mostly based on Wegmann and Leuenberger (2019). Using the same notation as in section 2, we denote by $\boldsymbol{X}$ the observed data, by $\boldsymbol{\theta}$ the model parameters and by $\boldsymbol{z}$ the hidden (latent) variables. We will assume the most general model with K independent observations $\boldsymbol{X} = \boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(K)}$, where each $\boldsymbol{x}^{(k)} = x_0^{(k)}, \ldots, x_N^{(k)}$ is of length N with $k = 1, \ldots, K$.
The model parameters $\boldsymbol{\theta}$ govern all initial distributions $\mathbb{P}(z_0)$, the transition probabilities $\mathbb{P}(z_i|z_{i-1})$ and the emission probabilities $\mathbb{P}(x_i^{(k)}|z_i)$. The complete data log likelihood is given by

$$\ell_c(\boldsymbol{\theta}) = \sum_{k=1}^{K}\left[\log\mathbb{P}(z_0^{(k)}) + \sum_{i=1}^{N}\log\mathbb{P}(z_i^{(k)}|z_{i-1}^{(k)}) + \sum_{i=0}^{N}\log\mathbb{P}(x_i^{(k)}|z_i^{(k)})\right].$$

Maximum likelihood estimates of $\boldsymbol{\theta}$ can be obtained by iteratively maximizing the Q-function $Q(\boldsymbol{\theta}|\boldsymbol{\theta}') = \mathbb{E}[\ell_c(\boldsymbol{\theta})|\boldsymbol{\theta}', \boldsymbol{X}]$, which is defined as

$$
\begin{aligned}
Q(\boldsymbol{\theta}|\boldsymbol{\theta}') &= \sum_{k=1}^{K} \mathbb{E}\left[\log \mathbb{P}(z_0^{(k)})|\boldsymbol{x}_{0:N}^{(k)}, \boldsymbol{\theta}'\right] + \sum_{k=1}^{K}\sum_{i=1}^{N} \mathbb{E}\left[\log \mathbb{P}(z_i^{(k)}|z_{i-1}^{(k)})|\boldsymbol{x}_{0:N}^{(k)}, \boldsymbol{\theta}'\right] + \dots \\
&\quad + \sum_{k=1}^{K}\sum_{i=0}^{N} \mathbb{E}\left[\log \mathbb{P}(x_i^{(k)}|z_i^{(k)})|\boldsymbol{\theta}'\right] \\
&= \sum_{k=1}^{K}\sum_{z_0} \gamma_0^{(k)}(z_0) \log \mathbb{P}(z_0) + \sum_{k=1}^{K}\sum_{i=1}^{N}\sum_{z_i}\sum_{z_{i-1}} \xi_i^{(k)}(z_i, z_{i-1}) \log \mathbb{P}(z_i|z_{i-1}) + \dots \\
&\quad + \sum_{k=1}^{K}\sum_{i=0}^{N}\sum_{z_i} \gamma_i^{(k)}(z_i) \log \mathbb{P}(x_i^{(k)}|z_i),
\end{aligned}
$$

where the $\gamma_i^{(k)}(z_i)$ and $\xi_i^{(k)}(z_i, z_{i-1})$ denote the expectation weights $\mathbb{P}(z_i|\boldsymbol{x}_{0:N}^{(k)})$ and $\mathbb{P}(z_i, z_{i-1}|\boldsymbol{x}_{0:N}^{(k)})$, respectively. These weights can be calculated efficiently with the forward-backward algorithm, where the $\gamma_i^{(k)}(z_i)$ are given by

$$
\gamma_i^{(k)}(z_i) = \frac{\alpha_i^{(k)}(z_i)\beta_i^{(k)}(z_i)}{\sum_h \alpha_i^{(k)}(h)\beta_i^{(k)}(h)}.
$$

and all $\xi_i^{(k)}(z_i, z_{i-1})$ by

$$
\xi_i^{(k)}(z_i, z_{i-1}) := \mathbb{P}(z_i, z_{i-1}|\boldsymbol{x}_{0:N}^{(k)}) \propto \alpha_{i-1}^{(k)}(z_{i-1})\mathbb{P}(z_i|z_{i-1})\beta_i^{(k)}(z_i)\mathbb{P}(x_i^{(k)}|z_i), \tag{50}
$$

where one has to normalize such that the sum over z_i adds up to one. In `stattools`, this is implemented in class `THMMPosteriorXi`.

In line with the EM algorithm, `stattools` partitions the Baum-Welch algorithm in three main classes. Class `TLatentVariableWithRep` represents the observation model $\mathbb{P}(x_i^{(k)}|z_i)$ and is exactly the same as in the EM framework. Class `TTransitionMatrix_baseWithRep` represents the transition probabilities $\mathbb{P}(z_i|z_{i-1})$. And finally, class `THMM` runs the forward-backward and/or the Baum-Welch algorithm and coordinates the interaction between the latent variable and transition matrix. In the following, the latter two classes are described in more detail.

3.6 `TTransitionMatrix_base`

The class `TTransitionMatrix_base` represents the transition probabilities and is in charge of calculating $\mathbb{P}(z_i|z_{i-1})$. It is an abstract base classes and defines two pure virtual functions:

1. `operator(size_t i, size_t from, size_t to)`: This function calculates the transition probability $\mathbb{P}(z = \text{to}|z = \text{from})$ at data point `i`.
2. `stationary(size_t z)`: This function calculates the stationary probability $\mathbb{P}(z)$.

The class has a pre-defined implementation of initial probabilities, which are simply estimated as the expected number of the K independent runs that started in state z_0:

$$
\mathbb{P}(z_0) = \frac{1}{K}\sum_{k=1}^{K} \gamma_0^{(k)}(z_0).
$$

We will assume that this implementation will be used in all applications, and won't describe the functions needed to override this default below for simplicity. However, we note that every

function listed below that is applicable to `TransitionProbabilities` can also be defined for `InitialDistribution` (e.g. `handleEMParameterEstimationOneIterationTransitionProbabilities()` and `handleEMParameterEstimationOneIterationInitialDistribution()`), and can be easily overridden if needed.
A developer who whishes to implement an HMM must derive from `TTransitionMatrix_base` and override the functions `operator()` and `stationary()`. The class class provides more virtual functions that may be overridden if needed. If the goal is to run a Baum-Welch algorithm, the following functions should be considered for overriding.
Before the EM starts, the following functions are called:

1. `prepareEMParameterEstimationInitial()`: Initialize temporary values or allocate memory needed to store the EM weights (only if necessary).

2. `initializeEMParameters()`: Estimate an initial guess of the transition matrix parameters. By default, the initial guess is obtained by one full loop over all $\boldsymbol{X}$, calculating the maximum likelihood state $\hat{z}$ for each index i and calling `handleEMParameterInitializationTransitionProbabilities()` with this state.

3. `handleEMParameterInitializationTransitionProbabilities(size_t i, size_t from, size_t to)`: Store or sum the guess of the hidden state `from` to hidden state `to`.

4. `finalizeEMParameterInitializationTransitionProbabilities()`: Get an initial guess for the transition matrix parameters based on the guesses of the hidden states $\boldsymbol{z}$ obtained by the previous function.

Then, in each EM iteration, these functions are called:

1. `prepareEMParameterEstimationOneIterationTransitionProbabilities()`: Called right in the beginning of the iteration. This function should reset temporary values for the new iteration (only if necessary).

2. `handleEMParameterEstimationOneIterationTransitionProbabilities(size_t i, THMMPosteriorXi Xi)`: The object `Xi` contains the EM weights $\xi_i^{(k)}(z_i, z_{i-1})$ that were calculated within `stattools` according to equation (50). Depending on the M-step, the developer will need to either store the weights or sum them up.

3. `finalizeEMParameterEstimationOneIterationTransitionProbabilities()`: Called at the end of the iteration. This function should perform the M-step.

4. `reportEMParameters()`: Called at the end of the iteration. Provides the possibility to report the updated parameters to the logfile.

At the very end of the EM, this function is called:

1. `finalizeEMParameterEstimationFinal()`: Clear temporary variables or memory (only if necessary).

In addition, the class `TTransitionMatrix_base` provides functions to simulate hidden states based on the transition matrix:

1. `simulateStates(Container Data)`: Fills a (templated) container with hidden states according to the transition probabilities. Iteratively calls the function `sampleNextState()`.

2. `sampleNextState(size_t i, size_t from`: Samples the next state given the previous state `from` according to the transition probabilities.

3.6.1 `TTransitionMatrixDistances`

As described in section 3.1, `stattools` implements a collection of transition matrices that are scaled with the distance between two neighbouring data points. The class `TTransitionMatrixDistances` inherits from `TTransitionMatrix_base` and overrides all virtual functions described above such that the parameters of any arbitrarily parametrized transition matrix can be estimated.
The most important member variables of `TTransitionMatrixHMMDistancesBase` are:

- `_distances`: A pointer an instance of class `TDistancesBinnedBase` that handles the distances δ_i between adjacent data points. As described in section 3.1, these distances are binned to E ensembles for computational reasons.
- `_tau_allDist`: A vector of size $E+1$ that stores the current transition matrices $(\boldsymbol{Q}(0), \boldsymbol{Q}(1), \ldots, \boldsymbol{Q}(E))$ as `Armadillo` matrices. Here, $\boldsymbol{Q}(0)$ represents the stationary distribution $\mathbb{P}(z_0)$, and all subsequent $\boldsymbol{Q}(e)$ represent the transition matrices for ensemble e.
- `_tau_allDist_old`: A vector of size $E + 1$ of `Armadillo` matrices. If used within an MCMC, these correspond to the old transition matrices. If used within an Baum-Welch, these store the EM sums ξ.
- `_optimizer`: A pointer to a derived class of `TTransitionMatrixDistancesOptimizerBase`, which handles the optimization of the transition matrix parameters in the M-step.

The following virtual functions of class `TTransitionMatrix_base` are overridden:

1. `operator(size_t i, size_t from, size_t to)`: This function returns the transition probability $\mathbb{P}(z = \text{to}|z = \text{from})$ at data point `i`, which is stored in `_tau_allDist[e]` for ensemble `e` of index `i`.
2. `stationary(size_t z)`: This function returns the stationary probability $\mathbb{P}(z)$, which is stored in `_tau_allDist[0]`.
3. `initializeEMParameters()`: Three initialization types are supported (enum `TypeTransMatInitialization`). `none` implies no initialization and keeps the user-provided values. `ML` implies maximum-likelihood estimation through `handleEMParameterInitializationTransitionProbabilities()`. `defaultValues` implies that the default values of the transition matrix parameters are used, which are defined in the corresponding transition matrix class.
4. `handleEMParameterInitializationTransitionProbabilities(size_t i, size_t from, size_t to)`: This sums up all the transitions `from` $\rightarrow$ `to` for all data points that fall within the same ensemble e.
5. `finalizeEMParameterEstimationOneIterationTransitionProbabilities()`: If initialization type is `ML`, this will call the function `finalizeEMParameterEstimationOneIterationTransitionProbabilities()`, which is also used at the end of a regular EM iteration.
6. `prepareEMParameterEstimationOneIterationTransitionProbabilities()`: This fills `_tau_allDist_old` with zeros.
7. `handleEMParameterEstimationOneIterationTransitionProbabilities(size_t i, THMMPosteriorXi Xi)`: This sums up all `Xi` for all data points that fall within the same ensemble e. `finalizeEMParameterEstimationOneIterationTransitionProbabilities()`: This class asks the `_optimizer` to maximize the Q-function.

In addition, the class provides functions to simulate distances (`simulateDistances()`) and to simulate hidden states according to the transition matrix parameters and distances (`sampleNextState()`). Also, it is possible to activate the reporting of EM parameters to the logfile through the function `report()`.

3.6.2 `TTransitionMatrixDistancesOptimizerBase` and derived classes

The class `TTransitionMatrixDistancesOptimizerBase` is an abstract base class that defines the interface of the optimizer classes. It defines the pure virtual function `maximizeQFunction()` that is called at the M-step of the Baum-Welch algorithm. In addition, it defines the pure virtual functions `_fillTransitionProbabilities()` and `_fillStationaryDistribution()` that fill a vector of `Armadillo` matrices with the estimated transition matrices for all distance groups.
The class `TTransitionMatrixDistancesExponential` derives from the base class. It is used as a base class for all transition matrices that calculate all transition matrices of ensemble $e > 1$ by squaring the transition matrix of the previous ensemble e. The class is templated with `TypeTransMat` that defines the type of transition matrix from section 3.1. The class holds an instance of such a transition matrix, and the function `_fillTransitionProbabilities()` fills a vector of `Armadillo` matrices by repeatedly squaring this transition matrix.
The following classes derive from `TTransitionMatrixDistancesExponential`:

3.6.2.1 TOptimizeTransMatLineSearch

Uses a line search to find parameters maximizing the Q-function. This only works for transition matrices parametrized by a single parameter, i.e. for `TTransitionMatrixLadder`.

3.6.2.2 TOptimizeTransMatNelderMead

Uses the Nelder-Mead algorithm to find parameters maximizing the Q-function. This is used for transition matrices with multiple parameters, i.e. for `TTransitionMatrixBool`, `TTransitionMatrixCategorical`, `TTransitionMatrixBoolGeneratingMatrix` and `TTransitionMatrixScaledLadder`. Unfortunately, it is not possible to take the derivative of the Q-function with respect to the transition matrix parameters for these transition matrices due to the matrix exponential. We therefore resort to numerically optimize the relevant term of the Q-function in every EM-iteration using the Nelder-Mead algorithm (see section 4):

$$\begin{aligned} f &= \sum_{i=1}^{N} \sum_{z_i} \sum_{z_{i-1}} \xi_i(z_i, z_{i-1}) \log \mathbb{P}(z_i | z_{i-1}), \\ &= \sum_{e=0}^{E} \sum_{z_i} \sum_{z_{i-1}} \log \mathbb{P}(z_i | z_{i-1}, e) \sum_{i=1}^{N} I(e_i = e) \xi_i(z_i, z_{i-1}), \end{aligned}$$

Here, we sum up all ξ for one distance group and multiply with the transition probability for that distance group. For the first Nelder-Mead (in the first EM iteration), we choose a initial vertex based on the initial values of the transition matrix parameters, and define some displacement. In the subsequent iterations, we simply use the last simplex of the previous EM iteration. In addition, we dynamically adjust the termination threshold depending on the fractional difference between the best vertex of the current and previous EM-iteration. If the estimate of the previous run was already very good, we want to be very precise. If the estimate of the previous run was rough, we want to be preciser than before, but don't waste time with finding the exact root,

because the value is likely to change after the next EM iteration anyways. Therefore, the Nelder-Mead algorithm will have a rather high termination threshold in the first iterations, leading to little computational overhead.

3.6.2.3 TOptimizeTransMatNelderMeadAttractorShift

Uses the Nelder-Mead algorithm to find parameters maximizing the Q-function, but runs the Nelder-Mead algorithm for each attractor index separately. This is used for transition matrices that have an attractor, i.e. `TTransitionMatrixScaledLadderAttractorShift` and `TTransitionMatrixScaledLadderAttractorShift2`.

3.6.3 HMM with combined states

Certain models are defined by multiple parallel HMMs where the transition probabilities are calculated independently for each chain, but the emission probabilities are calculated on the combination of states of all chains jointly. For example, for chromosome painting in a diploid organism, two seperate HMMs run on the haplotypes. The transition probabilities are defined for each haplotype seperately. However, the emission probability is defined on the combination of the two haplotypes.
Class `TCombinedStatesTransitionMatrices` inherits from `TTransitionMatrix_base` to account for such cases. It is written as a wrapper around a vector of pointers to `TTransitionMatrixHMMDistancesBase` that represent the individual chains. Generally speaking, all functions of the class either decompose the combined states into the individual states per chain, or the other way around.
The transition probability of the combined states (implemented in function `operator()`) takes as an argument the combined states of the particular transition. The function decomposes these combined states into the individual states for each chain by determining the subscripts of the linear index based on the number of states per chain. The combined transition probability is then simply the product of the transition probabilities for these individual states of each chain. The same logic applies for calculating the stationary probability.
During the EM, the class received the EM weights ξ of the combined states in the function `handleEMParameterEstimationOneIterationTransitionProbabilities()`. This function "budles" these weights for each individual HMM by summing over the states of the other HMMs. Currently, this is done by a double loop over all combined states, and determining the relevant individual states. However, this algorithm could be improved, since the states are represented as "blocks", and certain terms could be re-used. One would need to think on how to do this in a clever way.

3.7 `THMM`

Class `THMM` runs the forward-backward and/or the Baum-Welch algorithm and the coordinates the interaction between classes `TLatentVariableWithRep` and `TTransitionMatrix_base`. In line with the EM algorithm and class `TEM` (section 2.3, it provides three main functionalities:

1. `runEM()`: Run the Baum-Welch algorithm to find the ML estimates of emission, initial state and transition matrix parameters.

2. `calculateLL()`: Run the forward recursion to calculate the marginal likelihood $\mathcal{L}(\boldsymbol{\theta})$ given fixed estimates of emission, initial state and transition matrix parameters.

3. `estimateStatePosteriors()`: Run the forward-backward algorithm to estimate the state posterior probabilities $\mathbb{P}(z|\boldsymbol{X}, \boldsymbol{\theta})$ for all states $z = 1, \ldots, Z$ given fixed estimates of emission,

initial state and transition matrix parameters. If called with a filename, the state posterior probabilities will automatically be written to a file.

The Baum-Welch algorithm will iterate until i) the difference in log-likelihoods between two iterations is smaller than some threshold (default 10^{-4}) or until ii) the maximum number of iterations is reached.

3.8 HMM in MCMC

The class `prior::THMMPrior` derives from `TTransitionMatrixHMMDistancesBase` and coordinates the Hastings ratios of transition matrix parameters in MCMCs. Prior distributions that represent an HMM (e.g. `THMMBoolInferred`, `THMMBoolGeneratingMatrixInferred` and `THMMLadderInferred`) are then straightforward to implement: They need to define their transition matrix, and manage its parameter(s). All the rest is either implemented in `TTransitionMatrixHMMDistancesBase` (i.e. initialization via Baum-Welch) and in `THMMPrior` (i.e. MCMC updates of z and transition matrix parameters).

3.9 Smart initialization of transition matrix parameters

Assume we have a model with a transition matrix and a latent variable that specifies an emission probability $\mathbb{P}(x_i^{(k)}|z_i)$, which is the probability to observe data $x_i^{(k)}$ at position i and replicate k given hidden state $z_i \in (0, \dots, Z-1)$ where Z is the number of states. To initialize the transition matrix parameters, a simple approach is to calculate the maximum likelihood state, which is the state z_i that maximizes the emission probabilities, and running a single optimization (e.g. Nelder-Mead) based on these states. However, the maximum likelihood state is very noisy and an initialization of the transition matrix based on this can move the transition matrix parameters to a very bad initial estimate.
We instead propose to run an initial Baum-Welch algorithm on a "simplified" version of the latent variable that is based on a smoothed version of the mean posterior state. In this case, the observed data is the mean posterior state $y_i^{(k)}$ at position i and replicate k, projected to the interval [0,1], which is given by

$$y_i^{(k)} = \frac{1}{Z-1} \sum_{z=0}^{Z-1} z \frac{\mathbb{P}(x_i^{(k)}|z)}{\sum_{h=0}^{Z-1} \mathbb{P}(x_i^{(k)}|h)}.$$

We model the emission probabilities $\mathbb{P}(y_i^{(k)}|z_i)$ using a Beta distribution parametrized in terms of mode m and concentration κ, which are given by

$$m = \frac{\alpha - 1}{\alpha + \beta - 2} \text{ and } \kappa = \alpha + \beta - 2, \tag{51}$$

with the constraints $0 \leq m \leq 1$ and $\kappa \geq 0$ and $\alpha, \beta \geq 1$. We parametrize the mode m of the Beta distribution as a function of ϕ and of the hidden state z_i as:

$$m(z_i) = \frac{1}{2}(1 - \phi) + \phi \frac{z_i}{Z-1},$$

where ϕ is a measure of information content with $\phi = 1$ meaning positive correlation and $\phi = 0$ meaning no correlation between $y_i^{(k)}$ and z_i.
Solving equation (51) for α and β, we get

$$\alpha(z_i) = 1 + \kappa m(z_i),$$

$$\beta(z_i) = 1 + \kappa(1 - m(z_i)).$$

The emission probability is then given by the density of the Beta distribution

$$y_i^{(k)}|z_i \sim \text{Beta}(\alpha(z_i), \beta(z_i)).$$

To update the EM parameters ϕ and κ, we use a Nelder-Mead algorithm that searches the values of ϕ and κ that maximize the following term of the Q-function:

$$Q = \sum_{k=1}^{K} \sum_{i=0}^{N} \sum_{z_i} \log \mathbb{P}(y_i^{(k)}|z_i)\gamma_i^{(k)}(z_i),$$

where we need to store all $\gamma_i^{(k)}(z_i)$ in memory to do this calculation. As the Nelder-Mead algorithm can not handle parameter constraints, it operates on $p = \text{logit}(\phi)$ and $k = \log(\kappa)$. In addition, to prevent the algorithm from being stuck at $\phi = 0$ (meaning no correlation at all), we set $\phi = \delta + \text{logistic}(p)(1 - \delta)$, where δ is some fixed small noise term. This simple latent variable is very useful for initialization of the transition matrices and is implemented in class `TLatentVariableInitialization`.

4 The Nelder-Mead algorithm

The Nelder-Mead algorithm (also called downhill simplex or amoeba algorithm) is a derivative-free multidimensional optimization algorithm that is used to find the minimum of a function with more than one independent variable (Nelder and Mead, 1965; Press et al., 2007). The method is a direct search method that only requires function evaluations, but no derivatives.
Consider a function $f(\boldsymbol{x})$ of N unknown variables $\boldsymbol{x} = x_1, \ldots, x_N$. The goal is to find the variables $\boldsymbol{x}_m$ that minimize this function. To explore the function space of f, the algorithm evaluates the function values at different points with a geometrical shape named simplex. A simplex is the simplest volume in the N-dimensional space and consists of $N + 1$ points. In one dimension, a simplex is a line, in two dimensions, it is a triangle, in three dimensions, it is a tetrahedron, etc. Each point of the simplex corresponds to a set of variables $\boldsymbol{x}_i$, and for each point, we can calculate the value of the function $f(\boldsymbol{x}_i)$ we aim to minimize. The algorithm then proceeds as follows. We first define a starting guess for all $N + 1$ points. One possibility is to determine one initial starting point $\boldsymbol{x}_1$ only, and to define all other N points as

$$\boldsymbol{x}_i = \boldsymbol{x}_1 + \Delta \boldsymbol{e}_i,$$

where $\boldsymbol{e}_i$ are N unit vectors and Δ is a constant that represents a guess of the length of the function, i.e. how large the initial simplex should be. The implemented algorithm in this framework provides an interface for 1) starting with an initial vertex $\boldsymbol{x}_1$ and Δ, 2) starting with an initial vertex $\boldsymbol{x}_1$ and all $\boldsymbol{e}_i$ displacement vectors, and 3) starting with a full initial simplex. Based on this initial guess, the algorithm proceeds with the following steps:

1. **Order**: Order the $N + 1$ points by their function value $f(\boldsymbol{x}_1) \leq \ldots \leq f(\boldsymbol{x}_N) \leq f(\boldsymbol{x}_{N+1})$. Here, $\boldsymbol{x}_1$ is the point with the lowest function value (i.e. the best point), $f(\boldsymbol{x}_N)$ is the point with the second-highest function value (i.e. the second-worst point), and $f(\boldsymbol{x}_{N+1})$ is the point with the highest function value (i.e. the worst point).

2. **Terminate**: Calculate the fraction

$$\epsilon = \frac{2|f(\boldsymbol{x}_{N+1}) - f(\boldsymbol{x}_1)|}{|f(\boldsymbol{x}_{N+1})| + |f(\boldsymbol{x}_1)| + \nu}, \tag{52}$$

where ν is an arbitrarily small number to avoid any division by zero. Given a threshold δ, terminate the algorithm if $\epsilon < \delta$. Otherwise, determine the centroid of all points except $\boldsymbol{x}_{N+1}$ as $\boldsymbol{x}_o = \frac{1}{N}\sum_{i=1}^{N} \boldsymbol{x}_i$ and go to Step 3.

3. **Reflection**: Reflect the worst point of the simplex through the opposite face. Calculate the reflected point as $\boldsymbol{x}_r = \boldsymbol{x}_o + \alpha(\boldsymbol{x}_o - \boldsymbol{x}_{N+1})$ with $\alpha > 0$, and evaluate $f(\boldsymbol{x}_r)$:
   - If the reflected point is the best point ($f(\boldsymbol{x}_r) < f(\boldsymbol{x}_1)$), go to 4.
   - Else, if the reflected point is better than the second worst point ($f(\boldsymbol{x}_r) < f(\boldsymbol{x}_N)$), replace the worst point $\boldsymbol{x}_{N+1}$ with $\boldsymbol{x}_r$, and go to step 1.
   - Else, the reflected point is worse than the second worst ($f(\boldsymbol{x}_r) \geq f(\boldsymbol{x}_N)$). Go to step 5.

4. **Expansion**: Reflection was successful, so the algorithm will try to push the new point even a bit further in this beneficial direction. Calculate the expanded point as $\boldsymbol{x}_e = \boldsymbol{x}_o + \gamma(\boldsymbol{x}_r - \boldsymbol{x}_o)$ with $\gamma > 1$, and evaluate $f(\boldsymbol{x}_e)$:
   - If the expanded point is better than the reflected point ($f(\boldsymbol{x}_e) < f(\boldsymbol{x}_r)$), replace the worst point $\boldsymbol{x}_{N+1}$ with $\boldsymbol{x}_e$, and go to step 1.
   - Else ($f(\boldsymbol{x}_e) \geq f(\boldsymbol{x}_r)$), replace the worst point $\boldsymbol{x}_{N+1}$ with $\boldsymbol{x}_r$, and go to step 1.

5. **Contraction**: Reflection was not successful, for instance because it overshot the peak, so the algorithm will contract the simplex by moving the worst point closer to the centroid. Calculate the contracted point as $\boldsymbol{x}_c = \boldsymbol{x}_o + \rho(\boldsymbol{x}_{N+1} - \boldsymbol{x}_o)$ with $0 < \rho \leq 0.5$, and evaluate $f(\boldsymbol{x}_c)$:
   - If the contracted point is better than the worst point ($f(\boldsymbol{x}_c) < f(\boldsymbol{x}_{N+1})$), replace the worst point $\boldsymbol{x}_{N+1}$ with $\boldsymbol{x}_c$, and go to step 1.
   - Else ($f(\boldsymbol{x}_c) \geq f(\boldsymbol{x}_{N+1})$), go to step 6.

6. **Shrinkage**: If contraction was also not successful, the simplex is shrank by moving all other points towards the best one. Replace all points of the simplex except the best point $\boldsymbol{x}_1$ with the shrunken values $\boldsymbol{x}_i = \boldsymbol{x}_1 + \sigma(\boldsymbol{x}_i - \boldsymbol{x}_1)$, and go to step 1.

Standard values for the reflection, expansion, contraction and shrink coefficients are $\alpha = 1, \gamma = 2, \rho = 0.5$ and $\sigma = 0.5$.
The default value for the termination threshold δ is 10^{-15} but can easily be changed depending on the precision required. Press et al., 2007 recommends to let δ be of order the machine precision, however, in my tests, the algorithm never reached this precision. To avoid infinite loops in such cases, the algorithm will stop after `_maxNumFuncEvaluations` function calls (default value: 20000). In addition, there is an option to dynamically adjust the termination tolerance based on some reference value. This was implemented for a scenario where the Nelder-Mead algorithm was used for maximizing the Q-function of an EM. In the first EM iterations, there is no need for Nelder-Mead to be precise, as the values will anyways be optimized further in the next EM iteration. However, towards the end of the EM, the Nelder-Mead should be increasingly precise. To achieve this, we pass a reference value (e.g. the function value of the best vertex of the previous Nelder-Mead run) and a precision factor to the algorithm. After a few function calls (by default 10), the algorithm will determine a new threshold δ' by computing the fractional range between the function value at the current best vertex $\boldsymbol{x}_1$ and the reference value $\boldsymbol{x}_{ref}$ analogous to (52), and scaling this with the precision factor τ:

$$\delta' = \tau \frac{2|f(\boldsymbol{x}_{ref}) - f(\boldsymbol{x}_1)|}{|f(\boldsymbol{x}_{ref})| + |f(\boldsymbol{x}_1)| + \nu}.$$

5 The Newton-Raphson algorithm

5.1 One-dimensional problems

The Newton-Raphson algorithm is a very popular algorithm for finding the root of a function, that is, the value where the function equals zero. It requires the evaluation of both the function $f(x)$ and the derivative $f'(x)$ at arbitrary points x. Based on an initial guess x_0, the algorithm iteratively improves the current value by linearly projecting the location at which $f(x) = 0$ using the derivative. In each iteration, the step size s is computed as

$$s = \frac{f(x_i)}{f'(x_i)},$$

a new value x_{i+1} is obtained with

$$x_{i+1} = x_i - s.$$

The algorithm terminates if either the function value $|f(x_i)|$ is smaller than some convergence criterion ϵ, or if the maximal number of iteration was reached.
Newton-Raphson is very powerful because it converges quadratically to the root, that is, the number of significant digits approximately doubles with each step when being close to the root. However, it is also very sensitive to bad initial guesses x_0 that are so far away from the root such that the iteration step overshoots, with little hope of recovery. One way to overcome this issue is to backtrack along the direction of s, until the step results in a value where the function is closer to zero than before: While $|f(x_{i+1})| > |f(x)|$, only propose a step of $\frac{s}{2}$, $\frac{s}{4}$, $\frac{s}{8}$ etc, until the step does not overshoot. This variant is implemented in the function `runNewtonRaphson_withBacktracking()`. Another possibility to deal with overshooting is a combination of bisection and Newton-Raphson (Press et al., 2007). This algorithm requires some initial guess of bounds, i.e. some values x_{min} and x_{max} that enclose the root. The algorithm will then take a bisection step whenever Newton-Raphson result in a value that is outside the bounds or whenever Newton-Raphson is not reducing the size of the bounds rapidly enough. For the first case, the step is out of bounds if $x_i > x_{max}$ or $x_i < x_{min}$. For the second case, we require the function value at the current iteration $f(x_i)$ to be at maximum twice as large than in the previous step $f(x_{i-1})$, by assuming that the derivative remains constant between two iterations and checking if

$$|2f(x_i)| > |s_{i-1}f'(x_i)|.$$

If one of these conditions is true, we calculate an adjusted step that spans half of the range, $s = \frac{1}{2}(x_{max} - x_{min})$, which corresponds to a standard Newton step. If not, the step is accepted. After each iteration, the bounds are adjusted to narrow the range. This algorithm is implemented in the function `runNewtonRaphson_withBisection()`. This Newton-Raphson variant can has the option to deal with functions that have a root, but then also converge towards zero. For certain initial values, the standard implementation would bisect into the flat part of the function and the algorithm would terminate since the value is very close to zero. However, there is a true root that is exactly zero. To overcome this issue, we define a template parameter that does not allow termination after a bisection step. The algorithm can thus only terminate after a Newton-Raphson step, and will therefore always find the true root (at the cost of potentially some extra iterations).
Both Newton-Raphson variants are templated to take 1) a class object, 2) a function f of that class whose root should be found, and 3) the derivative of that function, f'. In many applications, the peak of a function (e.g. the MLE of a likelihood) can be obtained by Newton-Raphson by providing the first and second derivative of that function to the algorithm, since the peak is the place where the first derivative is zero.

The default value for the termination threshold ϵ is 10^{-10} but can easily be changed depending on the precision required. To avoid infinite loops in such cases, the algorithm will stop after `_maxIterations` function calls (default value: 1000). In addition, there is an option to dynamically adjust the termination tolerance based on some reference value. This was implemented for a scenario where the Newton-Raphson algorithm was used for maximizing the Q-function of an EM. In the first EM iterations, there is no need for Newton-Raphson to be precise, as the values will anyways be optimized further in the next EM iteration. However, towards the end of the EM, the Newton-Raphson should be increasingly precise. To achieve this, we can call the function `dynamicallyAdjustTolerance()` and give some precision factor τ (e.g. 1000) to the algorithm. The algorithm will stop if

$$\frac{|f(x_i)|}{|f(x_{ref})|} < \tau,$$

where x_{ref} is the inital value, since this corresponds to the last value of the previous Newton-Raphson run in the EM setting described above.

5.2 Multi-dimensional problems

For multidimensional problems, i.e. if nonlinear systems of equations must be solved, the problem becomes slightly more complex, as we now have a vector of functions $\boldsymbol{F}$ to be zeroed, involving variables x_j:

$$F_j(x_0, x_1, \dots x_{N-1}) = 0; \quad j = 0, \dots, N-1.$$

In addition, we need the matrix of partial derivatives, the so-called Jacobian matrix $\boldsymbol{J}$ with entries

$$J_{jk} = \frac{\partial F_j}{\partial x_k}.$$

The Newton step here is

$$\boldsymbol{x}' = \boldsymbol{x} + \delta\boldsymbol{x},$$

where

$$\delta\boldsymbol{x} = -\boldsymbol{J}^{-1}\boldsymbol{F}.$$

We implemented a globally convergent Newton-Raphson algorithm with backtracking as described in Press et al., 2007, Chapter 9.7. This algorithm always tries a full Newton step at first, which guarantees quadratic convergence if close enough to the solution. However, in order to prevent overshooting, there is a check at each iteration whether the decreases the function value sufficiently. If not, the algorithm backtracks along the direction of the step until the step is accepted.
We further introduced (optional) interval checks. In some cases, there are constraints on the parameter values, e.g. they need to be (strictly) positive. However, the standard Newton-Raphson algorithm does not know about these constraints and may propose parameter values that violate the constraints. We implemented two solutions:

- The user can specify minimal and/or maximal values for each parameter using the functions `setMin()` and `setMax()`. If a Newton-Raphson step result in a value x'_j that violates its interval m_j, we calculate the back-tracking factor λ such that we instead jump exactly to the boundary of the interval:

  $$\lambda = \frac{m_j - x_j}{\delta x_j}$$

  If multiple values j violate the intervals, the take the minimal λ to guarantee that the step is valid. This is a very effective way to do backtracking in case of bounded values.

- If the conditions of a valid step are more complex than simple intervals on the parameter values, there is also an option to communicate to the algorithm that the step was invalid. Specifically, the user-defined function `Fun()`, which calculates the functions to be zeroed $\boldsymbol{F}$, must return a pair: i) the function $\boldsymbol{F}$ and ii) whether or not the proposed values $\boldsymbol{x}$ are valid or not. If the values are not valid, Newton-Raphson will do standard backtracking by halving the proposed step, instead of the more elaborate backtracking algorithm as described in Press et al. (2007).

This algorithm is implemented in class `TMultiDimensionalNewtonRaphson`. Note that the same dynamical termination threshold is implemented as described above for the one-dimensional Newton-Raphson. Specifically, the algorithm will stop if

$$\frac{\max_{j=0,\ldots,N-1} |F_j(\boldsymbol{x})|}{\max_{j=0,\ldots,N-1} |F_j(\boldsymbol{x}_{ref})|} < \tau.$$

5.2.1 Approximations to the Jacobian matrix

In some cases, calculating the Jacobian is not possible analytically. Class `TApproxJacobian` calculates the necessary partial derivatives of function $\boldsymbol{F}$ numerically by finite differences (see Press et al., 2007, Chapter 5.7 and 9.7 for a description of the algorithm). This class is readily used in combination with the Newton-Raphson algorithm described above.

6 Line search algorithm

The line search is a very simple algorithm used to find the minimum, maximum or root of a one-dimensional function f. Based on some initial guess x_0, it iteratively proposes steps s into one direction, and changes that direction when $f(x_{i+1}) > f(x_i)$ (for finding the minimum), $f(x_{i+1}) < f(x_i)$ (for finding the maximum), or when $f(x_{i+1}) \cdot f(x_i) > 0$ (for finding the root). The third case holds because of a sign change: if $f(x_{i+1})$ and $f(x_i)$ are both positive or both negative, the product will always be positive; however, if one is positive and the other isn't, it means we have crossed the root and need to go back. The root-finding line search also needs to make sure that the first step goes into the correct direction by checking if either $f(x_{i+1}) \cdot f(x_i) > 0$ (we crossed the root) or $|f(x_{i+1})| > |f(x_i)|$ (the function gets larger, so we are walking away from the root). For all cases, the step size is decreased by $s_{i+1} = -\frac{s_i}{\exp(1)}$ after a change of direction, in order to encroach the target. The algorithm terminates if either the step s is smaller than some convergence criterion ϵ, or if the maximal number of iteration was reached. All variants (`findMax()`, `findMin()` and `findZero()`) are implemented in class `TLineSearch` and are templated to take a class object as well as a function f of that class whose minimum/maximum/root should be found.

7 K-Means Clustering

The K-Means clustering algorithm assigns each of N data points to one of K possible clusters $\boldsymbol{S} = S_1, \ldots, S_K$ such that the within-cluster sum of squares is minimized:

$$\arg\min_{\boldsymbol{S}} \sum_{k=1}^{K} \sum_{\boldsymbol{x} \in S_k} ||\boldsymbol{x} - \boldsymbol{\mu}_k||,$$

where $\boldsymbol{x}$ is a d-dimensional data point and $\boldsymbol{\mu}_k$ is the mean of all data points in cluster S_k. The naive clustering algorithm proceeds iteratively using a simple EM algorithm as follows:

1. E-step: Assign each data point $\boldsymbol{x}_i$ to the cluster k with the nearest mean $\boldsymbol{\mu}_k$ in terms of squared Euclidean distance.
2. M-step: For each cluster k, re-calculate $\boldsymbol{\mu}_k$ as the mean of all data points $\boldsymbol{x}_i$ assigned to cluster k.

These two steps are repeated until the E-step does not change an assignment of any data point. Using random initial estimates of $\boldsymbol{\mu}_k$ can result in poor performance of the K-means algorithm. We therefore also implemented the `k-means++` algorithm from David Arthur and Sergei Vassilvitskii to have an educated guess of the initial cluster means $\boldsymbol{\mu}_k$. This algorithm proceeds as follows:

1. Pick one data point $\boldsymbol{x}$ at random and set the mean of the first cluster to this value, $\boldsymbol{\mu}_1 = \boldsymbol{x}$.
2. For each data point $\boldsymbol{x}$, find the nearest mean $\boldsymbol{\mu}_k$ that has already been assigned, and record the distance $||\boldsymbol{x} - \boldsymbol{\mu}_k||$.
3. Pick a data point at random as a new mean $\boldsymbol{\mu}_k$ by sampling from a weighted probability distribution given by the distances to the nearest mean.
4. Repeat steps 2 and 3 until all means have been assigned.

Intuitively, while the first cluster is chosen at random, all other clusters are chosen such that it is more likely to select a cluster distant from all previously assigned clusters, leading to far-spread initial means $\boldsymbol{\mu}_k$.

8 Tests

8.1 Unit tests

Most functions of this framework have one or multiple unit tests that check if the expected behaviour is fulfilled. All test files are stored inside the directory `tests/unittests/`, where the filename of the test file corresponds to the file name of the source file being tested. The unittests run with Google Tests (and Google Mock). When compiling with CMake, the executable to run the unit tests is automatically created and can be run with: `./unitTestMCMC`. Whenever code is changed, make sure that all tests pass.

8.2 Integration tests

Unit tests only test certain functions or rarely the interplay of multiple functions. To complement this, an integration test investigates the entire program - from defining parameters to building the correct DAG, running the MCMC, estimating prior parameters, writing to file etc. In order to have an automated and reliable test of various priors, we implemented a integration testing framework that makes it easy and fast to add tests for certain priors.
An integration test suceeds if the posterior distribution of the inferred parameters fulfills certain properties of a probability distribution. Specifically, if the posterior distribution is a probability distribution, we can take a random value from it and calculate the empirical cumulative distribution function (ECDF). We expect the position of the random value inside the ECDF to follow an uniform distribution - independent of the type of posterior distribution. The so-called probability integral transform (PIT) (Held and Sabanés Bové, 2014) is the value of the posterior distribution F at the simulated value x_o:

$$\text{PIT}(x_o) = F(x_o) = \mathbb{P}(X \leq x_o)$$

For many replicates of simulation and MCMC, the distribution of $\text{PIT}(x_o)$ is expected to follow the standard uniform distribution, $\text{PIT}(x_o) \sim \mathcal{U}(0,1)$, if the posterior distribution is well calibrated. If it is not (possibly due to some bug), the distribution of $\text{PIT}(x_o)$ will deviate from the uniform distribution.
To test for this, we then apply a Kolmogorov-Smirnov (KS) test. The KS test compares two cumulative distribution functions by calculating the maximum value of the absolute difference between these functions: $D = \max |S_N(x) - P(x)|$. In our case, $S_N(x)$ corresponds to cumulative distribution function of the ECDF, and $P(x)$ corresponds to the known cumulative distribution of a uniform distribution between 0 and 1. We can then calculate a p-value to check whether the distributions are the same (null hypothesis) or not. The integration test passes if the p-value is not significant, that is, that the ECDF of the true value in the posterior distribution is indeed uniform.
For parameters with few discrete states (typically latent factors), we resort to another calibration measure, since the KS-test is not applicable for parameters with discrete values (e.g. if we simulated $x_o = 1$, $\text{PIT}(1) = 1$. For these parameters, we resort to Sanders' calibration, which is a measure of how well the posterior probabilities represent the (binned) simulated values (Held and Sabanés Bové, 2014). It implies that on average $\pi_i \cdot 100\%$ of the events actually occur with posterior probability π_i. To do so, the posterior probabilites need to be grouped in J roughly equally-sized groups. For each group, π_j is the mean posterior probability, n_j is the number of samples and $\bar{x}_j$ is the mean of all simulated values. The total number of samples is denoted by $N = \sum_{j=1}^{J} n_j$. Sanders' calibration is then defined as

$$\text{SC} = \frac{1}{N} \sum_{j=1}^{J} n_j (\bar{x}_j - \pi_j)^2$$

Perfect calibration would result in SC=0. In order to have a statistical measure of significance, we randomly sample simulated values from the posterior probabilities π, and calculate Sanders' calibration SC_r for many such replicates. We obtain a p-value by simply counting the number of replicates where SC_r was larger than the observed SC_o:

$$p = \frac{1}{R} \sum_{r=1}^{R} \text{SC}_r \geq \text{SC}_o$$

If the observed SC is larger than what we would expect by chance, the p-value will be small and we will reject the null hypothesis that the parameter is well calibrated.
There is one integration test per implemented prior. All integration test files are stored inside `tests/integrationtests/`. This integration test might also serve as an example for developers that want to implement their own MCMC based on this library. An R-File called `plotTraces.R` is provided that has some functions pre-implemented in order to plot the trace files, in case a test does not pass.

9 Additional chapters

The following sections contain ideas of algorithms that were not or only partially implemented.

9.1 Predicting the acceptance of an update

In complex "Big Data" settings, calculating the likelihood for the Hastings-ratio may require the evaluation of a large number of terms for every update of a parameter. This represents a

major bottleneck of the MCMC. To tackle this problem, the algorithm of delayed acceptance has been proposed (Banterle et al., 2015). This algorithm partitions the acceptance step into several blocks, where each block is consecutively compared to a random uniform number. The first block that is rejected terminates the update. Consequently, an update can only be accepted after all blocks were evaluated. We have implemented this algorithm, but found it very hard to tune the parameters such that it would work for any possible model. In addition, the acceptance of an update still requires the evaluation of all terms, which is still a computational effort.
Instead, we propose a novel algorithm that predicts the acceptance or rejection of an update. The key idea is to use a subset of the terms composing the likelihood ratio to predict whether or not a update will be accepted or rejected. If the prediction is not certain enough, then more terms are added to the subset, until eventually the prediction is good enough or all terms have been evaluated. This prediction model thus evaluates as little Hasting ratios as possible while maintaining the same behaviour as if the overall Hastings ratio was evaluated.
In similar spirit to delayed acceptance (Banterle et al., 2015), we partition our data into D blocks such that the overall Hastings ratio is given by:

$$\frac{\mathbb{P}(x')\mathbb{P}(x'|\mathcal{D})}{\mathbb{P}(x)\mathbb{P}(x|\mathcal{D})} = \frac{\mathbb{P}(x')}{\mathbb{P}(x)} \prod_{i=1}^{N} \frac{\mathbb{P}(x'|\mathcal{D}_i)}{\mathbb{P}(x|\mathcal{D}_i)} = \prod_{d=1}^{D} \rho_d(x, x'),$$

where

$$\rho_d(x, x') = \frac{\mathbb{P}(x')}{\mathbb{P}(x)}^{J/N} \prod_{j=d_1}^{J} \frac{\mathbb{P}(x'|\mathcal{D}_j)}{\mathbb{P}(x|\mathcal{D}_j)} \tag{53}$$

is the Hastings ratio of all the elements of the dth block.
The block Hastings ratio are likely to be correlated with the overall Hastings ratio, altough certain blocks might be more informative about the overall Hastings ratio than others. Our goal is therefore to evaluate highly informative blocks first, since these yield a better prediction of the overall Hastings ratio. To do so, we must rank the blocks based on the correlation of the block Hastings ratio with the overall Hastings ratio. We track this correlation in an extra round of burnin (after the usual burnin rounds). Note that we still accept and reject based on the overall Hastings ratio. We then rank the blocks as follows: The highest-ranking block will be the block that maximizes the correlation of block Hastings ratio to overall Hastings ratio. We add this block to our final set B. After this, we find the block b that maximizes the joint correlation of $\{b, B\}$ to the overall Hastings ratio, and add it to B. We repeat this step until all blocks are contained in B.
Based on this ranking, we then fit a linear model to each cumulative block. That is, we fit a linear model to the block B_1, a linear model to the blocks $B_{1:2}$, a linear model to the blocks $B_{1:3}$ etc. In particular, we estimate the intercept β_0 as well as the slope β_1 and the standard deviation of the error σ for each cumulative block using linear regression:

$$h_{tot} = \beta_0 + \beta_1 h_{blocks} + \epsilon, \tag{54}$$

where h_{tot} is the overall Hastings ratio, h_{blocks} is the (cumulative) block Hastings ratio and $\epsilon \sim \mathcal{N}(0, \sigma)$.
For the subsequent chain, we use the following scheme when updating a parameter:

1. Propose a new value x' based on x.
2. Calculate the log prior ratio $\frac{\mathbb{P}(x')}{\mathbb{P}(x)}$.

3. Draw a random number from an uniform distribution, $r \sim \mathcal{U}(0,1)$.

4. Loop over all blocks according to the block ranking. For each block, calculate the log Hastings ratio for all elements within that block, as given by equation 53.

5. Predict the overall Hastings ratio based from the coefficients of the linear regression of that block, as given by equation 54.

6. Terminate update if we are significantly certain that the update will be accepted or rejected. For this, we use the normal error of the linear model. The probability that the predicted overall Hastings ratio is larger than the random value r is given by the cumulative density of the normal distribution: $F = \log(1-\Phi(r))$ where Φ is the cumulative density of the normal distribution with mean h_{tot} and standard deviation σ_d. If F is larger than a threshold t, we reject the update and terminate. If F is smaller than the threshold $1-t$, we accept the update and terminate.

7. Else add next block and repeat steps 4-7, until all blocks were evaluated.

This scheme is implemented in classes `TLogHCalculatorRegular`, `TLogHCalculatorBlocks`, `TLogH-Correlation`, , `TTotalLogH` and `TMultiLogHCalculatorBlocks`. Here is a short overview of these classes:

1. `TLogHCalculatorRegular`: implements the regular case where the log Hastings ratio is evaluated based on all terms. This is used for all parameters that do not evaluate their Hastings ratio in blocks.

2. `TLogHCalculatorBlocks`: implements the case where the log Hastings ratio is evaluated successively in blocks, as described above. Contains an instance of `TLogHCorrelation` to keep track of the correlation between block and overall log Hastings ratio.

3. `TLogHCorrelation`: stores the block and overall Hastings ratio, and calculates correlation as well as linear models based on these. Unfortunately, since we need to calculate the correlation based on merged blocks, we cannot sum the coefficients while the burnin, but must store every block Hastings ratio in memory. If this is problematic at some point, an alternative would be to write them to a file and reading it at the end of the tuning round, or to split the round in two, where in the first half we only keep track of the covariance (which can be done efficiently without the need to store all ratios), then determine the block order, and then directly learn the values for the relevant combination of blocks.

4. `TTotalLogH`: for the overall Hastings ratio, we do not need to store all terms, but only the summed up values (x and x^2), which is done in this class.

5. `TMultiLogHCalculatorBlocks`: If a parameter has a proposal kernel that not shared but is specific for each element, we also might want the prediction of the acceptance rate to be specific for each element. This class stores a vector of `TLogHCalculatorBlocks` and makes sure the correct logH-calculator is used for each update.

These classes are implemented but not in use, as the use-case for which this feature was developed was abolished. What is lacking is the effective use of these classes in `TParameter`.

9.2 `prior::TNormalMixedModelInferred`

A mixture model of K normal distributions is defined as:

$$x_i|z_i = k \sim \mathcal{N}(\mu_k, \sigma_k^2).$$

Here, z_i is a categorical variable denoting the component of observation x_i, and μ_k and σ_k^2 denote mean and variance of component k.
A normal mixture model is parametrized by

- A vector of means, one for each component, $\boldsymbol{\mu}$ (`_mus`).
- A vector of variances, one for each component, $\boldsymbol{\sigma}^2$ (`_vars`).
- Categorical variables $\boldsymbol{z}$ (`_z`) that indicate the component k for each observation x_i. These are linear with size N.

Here, `_mus` and `_vars` are vectors of pointers to `TParameter`, and `_z` is a pointer to `TParameter`. There is a check that the size of $\boldsymbol{z}$ matches the size of the parameter $\boldsymbol{x}$.
Note that a natural choice for a prior on $\boldsymbol{z}$ would be $z_i \sim \text{Categorical}(\boldsymbol{\pi})$, with one π_k per component. Then, we can impose a Dirichlet distribution on $\boldsymbol{\pi}$. This has the advantages that 1) all $\boldsymbol{\pi}$ then sum to one and 2) that we have a way to force the MCMC to pick one of the $K!$ solutions that are possible if there are no constraints. If we didn't impose any constraints or priors, the MCMC would switch between the components, such that the posterior would be a smear of all components.
The log prior density of a value x_i is:

$$\log \mathbb{P}(x_i|\mu_k, \sigma_k^2, z_i = k) = -\frac{1}{2}\log(\sigma_k^2) - \frac{1}{2}\log(2\pi) - \frac{1}{2\sigma_k^2}(x_i - \mu_k)^2. \tag{55}$$

The log prior ratio is:

$$\log p_x = \log\left(\frac{\mathbb{P}(x_i'|\mu_k, \sigma_k^2, z_i = k)}{\mathbb{P}(x_i|\mu_k, \sigma_k^2, z_i = k)}\right) = \frac{1}{2\sigma_k^2}\left((x_i - \mu_k)^2 - (x_i' - \mu_k)^2\right).$$

The log likelihood ratio l_{μ_k} for an update of μ_k is

$$l_{\mu_k} = \log\left(\frac{\mathbb{P}(\boldsymbol{x}|\mu_k', \sigma_k^2, \boldsymbol{z})}{\mathbb{P}(\boldsymbol{x}|\mu_k, \sigma_k^2, \boldsymbol{z})}\right) = \frac{1}{2\sigma_k^2}\sum_{i=1}^{N}\mathcal{I}(z_i = k)\left((x_i - \mu_k)^2 - (x_i - \mu_k')^2\right),$$

where $\mathcal{I}(z_i = k)$ is an indicator variable that takes value one if $z_i = k$ and zero otherwise.
The log likelihood ratio $l_{\sigma_k^2}$ for an update of parameter σ_k^2 is

$$\begin{aligned} l_{\sigma_k^2} &= \log\left(\frac{\mathbb{P}(\boldsymbol{x}|\mu_k, \sigma_k^{2'}, \boldsymbol{z})}{\mathbb{P}(\boldsymbol{x}|\mu_k, \sigma_k^2, \boldsymbol{z})}\right) \\ &= \frac{1}{2}\log\left(\frac{\sigma_k^2}{\sigma_k^{2'}}\right)\sum_{i=1}^{N}\mathcal{I}(z_i = k) + \frac{1}{2}\left(\frac{1}{\sigma_k^2} - \frac{1}{\sigma_k^{2'}}\right)\sum_{i=1}^{N}\mathcal{I}(z_i = k)(x_i - \mu_k)^2. \end{aligned}$$

The posterior distribution of z_i can be calculated analytically, because its values are limited to few discrete states. We therefore directly sample new values of z_i from the posterior distribution as specified in (1). The likelihood for sampling a specific z_i is then given by $\mathbb{P}(x_i|\mu_{z_i}, \sigma_{z_i}^2)$, and the prior by $\mathbb{P}(z_i|\pi)$. Note that this hold regardless of the prior distribution, e.g. HMMs.

We use an Expectation-Maximization (EM) algorithm for the initialization of the prior parameters . Let $\boldsymbol{\theta} = \{\boldsymbol{\mu}, \boldsymbol{\sigma}^2\}$ be the parameters of the mixed model, and $\boldsymbol{\pi}$ be the prior parameters on $\boldsymbol{z}$. The Q-function is:

$$Q(\boldsymbol{\theta}, \pi | \boldsymbol{\theta}', \boldsymbol{\pi}') := \mathbb{E}\left[l_c(\boldsymbol{\theta}, \boldsymbol{\pi}) \middle| \boldsymbol{x}, \boldsymbol{\theta}', \boldsymbol{\pi}' \right],$$

where $l_c(\boldsymbol{\theta}, \boldsymbol{\pi})$ is the complete data log-likelihood given by

$$l_c(\boldsymbol{\theta}, \boldsymbol{\pi}) = \log \mathbb{P}(\boldsymbol{x}, \boldsymbol{z} | \boldsymbol{\theta}, \boldsymbol{\pi}) = \sum_{p=1}^{P} \sum_{i=1}^{N} \left(\log \mathbb{P}(x_{p_i} | z_i = k, \boldsymbol{\theta}) + \log \mathbb{P}(z_i = k | \boldsymbol{\pi}) \right).$$

Here, $\log \mathbb{P}(x_{p_i} | z_i, \boldsymbol{\theta})$ is given by (55), and $\mathbb{P}(z_i | \boldsymbol{\pi})$ depends on the prior on z_i.
E-step. Given a current estimate $\boldsymbol{\theta}', \boldsymbol{\pi}'$, we now determine the Q-function:

$$Q(\boldsymbol{\theta}, \pi | \boldsymbol{\theta}', \pi') = \sum_{p=1}^{P} \sum_{i=1}^{N} \sum_{k=1}^{K} \left(\log \mathbb{P}(x_{p_i} | z = k, \boldsymbol{\theta}) + \log \mathbb{P}(z = k | \pi) \right) \mathbb{P}(z = k | x_{p_i}, \boldsymbol{\theta}', \pi'),$$

where with aid of Bayes' Theorem we have

$$\rho_{ki} := \mathbb{P}(z = k | x_{p_i}, \boldsymbol{\theta}', \pi') = \frac{\mathbb{P}(z = k | \boldsymbol{\pi}') \prod_{p=1}^{P} \mathbb{P}(x_{p_i} | z = k, \boldsymbol{\theta}')}{\sum_{j=1}^{K} \mathbb{P}(z = j | \boldsymbol{\pi}') \prod_{p=1}^{P} \mathbb{P}(x_{p_i} | z = j, \boldsymbol{\theta}')}. \tag{56}$$

M-step. The Q-function splits into separate terms for each parameter μ_k, σ_k^2 and π_k. In order to optimize the Q-function with respect to a parameter, we calculate the derivative and set it to zero. We obtain

$$\mu_k = \frac{\sum_{i=1}^{N} \rho_{ki} \sum_{p=1}^{P} x_{p_i}}{P \sum_{i=1}^{N} \rho_{ki}}$$

and

$$\sigma_k^2 = \frac{\sum_{i=1}^{N} \rho_{ki} \sum_{p=1}^{P} (x_{p_i} - \mu_k)^2}{P \sum_{i=1}^{N} \rho_{ki}}.$$

Although the next value for π_k depends on the prior distribution on z and is as such not part of this class, we note that under a Categorical distribution (which is the most natural prior choice), π_k is given by:

$$\pi_k = \frac{\sum_{i=1}^{N} \rho_{ki}}{N}.$$

The EM algorithm will then proceed as follows:

1. Initialization: Set $t = 0$ and choose $\boldsymbol{\theta}^{(t)}, \pi^{(t)}$ according to the following procedure:
   (a) First calculate the mean for each data point $\bar{x_i}$ across all parameters on which the prior is defined, $\bar{x_i} = \frac{1}{P} \sum_{i=1}^{P} x_{ip}$.
   (b) Assign the first data point $\bar{x_1}$ to the first component, $k = 1$.
   (c) For all remaining components k, apply the following steps:
      i. Go over all data points $\bar{\boldsymbol{x}}$. For each data point $\bar{x_i}$, store the minimal distance to all previous components (this corresponds to the distance to the closest component). To do so, calculate the Euclidean distance $D_M(\bar{x_i})$ to each previous component $j = 1, \ldots, k-1$. Then, take the minimal values among $D_M(\bar{x_i})$.
      ii. The center of component k is now given as the value $\bar{x_i}$ that maximizes $D_M(\bar{x_i})$ (i.e. we pick the component that is furthest away from all previous components).

(d) Finally, we go over all data again and assign each data point to its closest group, which serves as the initialization of the latent factors $\boldsymbol{z}$.

(e) We then calculate the mean and variance of each group and initialize μ_k and σ_k^2 accordingly.

2. For $t = 1, \ldots, T-1$:

(a) E-step: Calculate the EM-weight $\rho_{ki}^{(t)}$ according to (56).

(b) M-step: Update the parameters $\mu_k^{(t)}, \sigma_k^{2(t)}, \pi_k^{(t)}$ as described above.

We stop the EM after 100 iterations or as soon as the difference between the old and new log likelihood is smaller than 10^{-3}. The log likelihood is obtained by integrating out the latent variable z:

$$\log \mathbb{P}(\boldsymbol{x}|\boldsymbol{\theta}, \pi) = \sum_{k=1}^{K} \left(N \log \mathbb{P}(z = k|\pi_k) + \sum_{p=1}^{P} \sum_{i=1}^{N} \log \mathbb{P}(x_{pi}|z = k, \boldsymbol{\theta}) \right).$$

We finally initialize $\boldsymbol{z}$ to the value k that maximizes their posterior $\sum_{p=1}^{P} \mathbb{P}(z_i|x_{pi}, \boldsymbol{\theta}, \pi)$. For this, we simply re-compute the EM weights with (56) one more time (using the final estiates for $\boldsymbol{\theta}$ and $\boldsymbol{\pi}$) and pick the z_i that maximizes the posterior.
Note that there is an elegant way to estimate the parameters of a univariate normal distribution with 2 components via method of moments estimators (Cohen, 1967). If the mean of the two components is equal, the algorithm gets even simpler. This has not been implemented yet, but would be a powerful and fast alternative to the EM algorithm.
This should be solved using a general mixture model.

9.3 `prior::TMultivariateNormalMixedModelInferred`

This class implements a mixture model of K multivariate normal distributions:

$$\boldsymbol{x}_i|z_i = k \sim \mathcal{N}(\boldsymbol{\mu}^{(k)}, \boldsymbol{\Sigma}^{(k)}).$$

Here, z_i is a categorical variable denoting the component of the D-dimensional observation vector $\boldsymbol{x}_i$, and $\boldsymbol{\mu}^{(k)}$ and $\boldsymbol{\Sigma}^{(k)}$ denote the D-dimensional mean vector and the DxD-dimensional variance-covariance matrix of the multivariate normal distribution of component k.
We parametrize the multivariate normal distribution of each component as described above for `TPriorMultivariateNormalWithHyperPrior`: We apply Cholesky factorization to $\boldsymbol{\Sigma}^{(k)}$ and obtain $\boldsymbol{M}^{(k)}$, which is factorized into the elements $m^{(k)}$, $\boldsymbol{m}_{rr}^{(k)}$ and $\boldsymbol{m}_{rs}^{(k)}$. The density of the each component k is thus given by equation 35. Therefore, a multivariate normal mixed model is parametrized by

- A vector of means, one for each component. Each $\boldsymbol{\mu}^{(k)}$ has size D (`_mus`).
- A vector of scale factors $m^{(k)}$, one for each component. Each $m^{(k)}$ has size 1 (`_m`).
- A vector of diagonal elements $\boldsymbol{m}_{rr}^{(k)}$, one for each component. Each $\boldsymbol{m}_{rr}^{(k)}$ has size D (`_Mrr`).
- A vector of off-diagonal elements $\boldsymbol{m}_{rs}^{(k)}$, one for each component. Each $\boldsymbol{m}_{rs}^{(k)}$ has size T (`_Mrs`), see (36).
- Categorical variables $\boldsymbol{z}$ of size N (`_z`).

Here, `_mus`, `_m`, `_Mrr` `_Mrs` are vectors of pointers to `TParameter`, and `_z` is a pointer to `TParameter`.
The parameter on which this prior is defined, $\boldsymbol{X}$, must have dimensions NxD, i.e. the number of dimensions of the multivariate normal prior should be the columns of $\boldsymbol{X}$. This is beneficial for cache handling: When calculating the prior densities, we always need one row $\boldsymbol{x}_i$ from $\boldsymbol{X}$, which corresponds to one consecutive chunck of memory because we store matrices in a row-major order.
Note that a natural choice for a prior on $\boldsymbol{z}$ would be $z_i \sim \text{Categorical}(\boldsymbol{\pi})$, with one $\pi^{(k)}$ per component. Then, we can impose a Dirichlet distribution on $\boldsymbol{\pi}$. This has the advantages that 1) all $\boldsymbol{\pi}$ then sum to one and 2) that we have a way to force the MCMC to pick one of the $K!$ solutions that are possible if there are no constraints. If we didn't impose any constraints or priors, the MCMC would switch between the components, such that the posterior would be a smear of all components.
The multivariate normal distribution is a non-iid prior, as the prior density of one x_{id} depends on all $\boldsymbol{x}_i$. The log prior density of a value x_{id} with parameters $\boldsymbol{\mu}^{(k)}$, $\boldsymbol{M}^{(k)} = \{m^{(k)}, \boldsymbol{m}_{rr}^{(k)}, \boldsymbol{m}_{rs}^{(k)}\}$ is

$$\begin{aligned}
&\log \mathbb{P}(x_{id}|\boldsymbol{\mu}^{(k)}, \boldsymbol{M}^{(k)}, z_i = k) \\
&\quad = -D \log m^{(k)} + \sum_{r=1}^{D} m_{rr}^{(k)} - \frac{D}{2}\log(2\pi) - \frac{1}{2m^{(k)2}} \sum_{s=1}^{D} \left(\sum_{r=s}^{D} m_{rs}^{(k)} (x_{ir} - \mu_r^{(k)}) \right)^2 . \quad (57)
\end{aligned}$$

The log prior ratio for x_{id} is:

$$\begin{aligned}
\log p_{x_{id}} &= \log \left(\frac{\mathbb{P}(x'_{id}|\boldsymbol{\mu}^{(k)}, \boldsymbol{M}^{(k)}, z_i = k)}{\mathbb{P}(x_{id}|\boldsymbol{\mu}^{(k)}, \boldsymbol{M}^{(k)}, z_i = k)} \right) = \\
&= \frac{1}{2m^{(k)2}} \sum_{s=1}^{D} \left(\left(\sum_{r=s}^{D} m_{rs}^{(k)} (x_{ir} - \mu_r^{(k)}) \right)^2 - \left(\sum_{r=s}^{D} m_{rs}^{(k)} (x'_{ir} - \mu_r^{(k)}) \right)^2 \right) \\
&= \frac{1}{2m^{(k)2}} \left(\sum_{r=1}^{d} m_{dr}^{(k)2} \left(x_{id}^2 - x_{id}'^2 + 2\mu_d^{(k)} (x'_{id} - x_{id}) \right) + \right. \quad (58) \\
&\qquad \left. 2(x_{id} - x'_{id}) \sum_{s=1}^{S} m_{ds}^{(k)} \sum_{r=s, r\neq d}^{D} m_{rs}^{(k)} (x_{ir} - \mu_r^{(k)}) \right),
\end{aligned}$$

where $S = \min(d, D-1)$.
The log likelihood ratio $l_{\mu_d^{(k)}}$ for an update of $\mu_d^{(k)}$ is:

$$\begin{aligned}
l_{\mu_d^{(k)}} &= \log \left(\frac{\mathbb{P}(\boldsymbol{X}|\mu_d^{(k)'}, \boldsymbol{M}^{(k)}, \boldsymbol{z})}{\mathbb{P}(\boldsymbol{X}|\mu_d^{(k)}, \boldsymbol{M}^{(k)}, \boldsymbol{z})} \right) \\
&= \frac{1}{2m^{(k)2}} \sum_{i=1}^{N} \mathcal{I}(z_i = k) \sum_{s=1}^{D} \left(\left(\sum_{r=s}^{D} m_{rs}^{(k)} (x_{ir} - \mu_r^{(k)}) \right)^2 - \left(\sum_{r=s}^{D} m_{rs}^{(k)} (x_{ir} - \mu_r^{(k)'}) \right)^2 \right) \\
&= \frac{1}{2m^{(k)2}} \left((\mu_d^{(k)2} - \mu_d^{(k)'2}) \left(\sum_{r=1}^{d} m_{dr}^{(k)2} \right) \left(\sum_{i=1}^{N} \mathcal{I}(z_i = k) \right) + 2(\mu_d^{(k)'} - \mu_d^{(k)}) \sum_{r=1}^{d} m_{dr}^{(k)2} \sum_{i=1}^{N} \mathcal{I}(z_i = k) x_{id} + \right. \\
&\quad \left. 2(\mu_d^{(k)'} - \mu_d^{(k)}) \sum_{i=1}^{N} \mathcal{I}(z_i = k) \sum_{s=1}^{S} m_{ds}^{(k)} \sum_{r=s, r\neq d}^{D} m_{rs}^{(k)} (x_{ir} - \mu_r^{(k)}) \right),
\end{aligned}$$

where $\mathcal{I}(z_i = k)$ is an indicator variable that takes value one if $z_i = k$ and zero otherwise. The log likelihood ratio $l_{m^{(k)}}$ for an update of $m^{(k)}$ is:

$$\begin{aligned} l_{m^{(k)}} &= \log\left(\frac{\mathbb{P}(\boldsymbol{X}|\boldsymbol{\mu}^{(k)}, \boldsymbol{M}^{(k)'}, \boldsymbol{z})}{\mathbb{P}(\boldsymbol{X}|\boldsymbol{\mu}^{(k)}, \boldsymbol{M}^{(k)}, \boldsymbol{z})}\right) \\ &= D\left(\log m^{(k)} - \log m^{(k)'}\right) \sum_{i=i}^{N} \mathcal{I}(z_i = k) + \\ &\quad \frac{1}{2}\left(\frac{1}{m^{(k)2}} - \frac{1}{m^{(k)'2}}\right) \sum_{i=1}^{N} \mathcal{I}(z_i = k) \sum_{s=1}^{D} \left(\sum_{r=s}^{D} m_{rs}^{(k)}(x_{ir} - \mu_r^{(k)})\right)^2. \end{aligned}$$

The log likelihood ratio $l_{m_{rr}^{(k)}}$ for an update of $m_{rr}^{(k)}$ is:

$$\begin{aligned} l_{m_{rr}^{(k)}} &= \log\left(\frac{\mathbb{P}(\boldsymbol{X}|\boldsymbol{\mu}^{(k)}, \boldsymbol{M}^{(k)'}, \boldsymbol{z})}{\mathbb{P}(\boldsymbol{X}|\boldsymbol{\mu}^{(k)}, \boldsymbol{M}^{(k)'}, \boldsymbol{z})}\right) \\ &= \sum_{i=1}^{N} \mathcal{I}(z_i = k) \left(\log\left(\frac{m_{rr}^{(k)'}}{m_{rr}^{(k)}}\right) + \frac{1}{2m^{(k)2}} \sum_{s=1}^{D} \left(\left(\sum_{d=s}^{D} m_{ds}^{(k)}(x_{id} - \mu_d^{(k)})\right)^2 - \left(\sum_{d=s}^{D} m_{ds}^{(k)'}(x_{id} - \mu_d^{(k)})\right)^2\right)\right) \\ &= (\log m_{rr}^{(k)'} - \log m_{rr}^{(k)}) \sum_{i=1}^{N} \mathcal{I}(z_i = k) + \frac{1}{2m^{(k)2}}\Big((m_{rr}^{(k)2} - m_{rr}^{(k)'2}) \sum_{i=1}^{N} \mathcal{I}(z_i = k)(x_{ir} - \mu_r^{(k)})^2 + \\ &\quad 2(m_{rr}^{(k)} - m_{rr}^{(k)'}) \sum_{i=1}^{N} \mathcal{I}(z_i = k)(x_{ir} - \mu_r^{(k)}) \sum_{d=r+1}^{D} m_{dr}^{(k)}(x_{id} - \mu_d^{(k)})\Big). \end{aligned}$$

The log likelihood ratio $l_{m_{rs}^{(k)}}$ for an update of $m_{rs}^{(k)}$ is:

$$\begin{aligned} l_{m_{rs}^{(k)}} &= \log\left(\frac{\mathbb{P}(\boldsymbol{X}|\boldsymbol{\mu}^{(k)}, \boldsymbol{M}^{(k)'}, \boldsymbol{z})}{\mathbb{P}(\boldsymbol{X}|\boldsymbol{\mu}^{(k)}, \boldsymbol{M}^{(k)'}, \boldsymbol{z})}\right) \\ &= \frac{1}{2m^{(k)2}} \sum_{i=1}^{N} \mathcal{I}(z_i = k) \sum_{s=1}^{D} \left(\left(\sum_{r=s}^{D} m_{rs}^{(k)}(x_{ir} - \mu_r^{(k)})\right)^2 - \left(\sum_{r=s}^{D} m_{rs}^{(k)'}(x_{ir} - \mu_r^{(k)})\right)^2\right) \\ &= \frac{1}{2m^{(k)2}}\Bigg((m_{rs}^{(k)2} - m_{rs}^{(k)'2}) \sum_{i=1}^{N} \mathcal{I}(z_i = k)(x_{ir} - \mu_r^{(k)})^2 + \\ &\quad 2(m_{rs}^{(k)} - m_{rs}^{(k)'}) \sum_{i=1}^{N} \mathcal{I}(z_i = k)(x_{ir} - \mu_r^{(k)}) \sum_{d=s, d\neq r}^{D} m_{ds}^{(k)}(x_{id} - \mu_d^{(k)})\Bigg). \end{aligned}$$

Note that if $D = 1$, $\boldsymbol{m}_{rs}^{(k)}$ is non-existant and $\boldsymbol{m}_{rr}^{(k)}$ is superficial and fixed to 1, such that only $m^{(k)}$ is estimated.

The posterior distribution of z_i can be calculated analytically, because its values are limited to few discrete states. We therefore directly sample new values of z_i from the posterior distribution as specified in (1). The likelihood for sampling a specific z_i is then given by $\mathbb{P}(x_i|\boldsymbol{\mu}^{(z_i)}, \boldsymbol{M}^{(z_i)})$, and the prior by $\mathbb{P}(z_i|\pi)$. Note that this hold regardless of the prior distribution, e.g. also for HMMs.

This prior requires the library Armadillo to be installed. If Armadillo is not found while compilation, the `stattools` library still compiles, but this class can not be used (an error is throwed

when its constructor is called).
We use an EM algorithm for the initialization of the prior parameters . Let $\boldsymbol{\theta} = \{\boldsymbol{\mu}^{(k)}, \boldsymbol{\Sigma}^{(k)}\}$ be the parameters of the mixed model, and $\boldsymbol{\pi}$ be the prior parameters on $\boldsymbol{z}$. The Q-function is:

$$Q(\boldsymbol{\theta}, \boldsymbol{\pi}|\boldsymbol{\theta}', \boldsymbol{\pi}') := \mathbb{E}\left[l_c(\boldsymbol{\theta}, \boldsymbol{\pi})\middle|\boldsymbol{X}, \boldsymbol{\theta}', \boldsymbol{\pi}'\right],$$

where $l_c(\boldsymbol{\theta}, \boldsymbol{\pi})$ is the complete data log-likelihood given by

$$l_c(\boldsymbol{\theta}, \boldsymbol{\pi}) = \log \mathbb{P}(\boldsymbol{X}, \boldsymbol{z}|\boldsymbol{\theta}, \boldsymbol{\pi}) = \sum_{p=1}^{P}\sum_{i=1}^{N}\left(\log \mathbb{P}(\boldsymbol{x}_{p_i}|z_i = k, \boldsymbol{\theta}) + \log \mathbb{P}(z_i = k|\pi^{(k)})\right).$$

Here, $\log \mathbb{P}(\boldsymbol{x}_{p_i}|z_i = k, \boldsymbol{\theta})$ is given by

$$\mathbb{P}(\boldsymbol{x}_{p_i}|z_i = k, \boldsymbol{\theta}) = -\frac{1}{2}\log|\boldsymbol{\Sigma}^{(k)}| - \frac{1}{2}(\boldsymbol{x}_{p_i} - \boldsymbol{\mu}^{(k)})^T{\boldsymbol{\Sigma}^{(k)}}^{-1}(\boldsymbol{x}_{p_i} - \boldsymbol{\mu}^{(k)}) - \frac{D}{2}\log(2\pi).$$

and $\mathbb{P}(z_i = k|\pi^{(k)})$ depends on the prior on z_i.
E-step. Given a current estimate $\boldsymbol{\theta}', \boldsymbol{\pi}'$, we now determine the Q-function:

$$Q(\boldsymbol{\theta}, \boldsymbol{\pi}|\boldsymbol{\theta}', \boldsymbol{\pi}') = \sum_{p=1}^{P}\sum_{i=1}^{N}\sum_{k=1}^{K}\left(\log \mathbb{P}(\boldsymbol{x}_{p_i}|z = k, \boldsymbol{\theta}) + \log \mathbb{P}(z = k|\boldsymbol{\pi})\right)\mathbb{P}(z = k|\boldsymbol{x}_{p_i}, \boldsymbol{\theta}', \boldsymbol{\pi}')$$

where with aid of Bayes' Theorem we have

$$\rho_{ki} := \mathbb{P}(z = k|\boldsymbol{X}_i, \boldsymbol{\theta}', \boldsymbol{\pi}') = \frac{\mathbb{P}(z = k|\boldsymbol{\pi}')\sum_{p=1}^{P}\mathbb{P}(\boldsymbol{x}_{p_i}|z = k, \boldsymbol{\theta}')}{\sum_{j=1}^{K}\mathbb{P}(z = j|\boldsymbol{\pi}')\sum_{p=1}^{P}\mathbb{P}(\boldsymbol{x}_{p_i}|z = j, \boldsymbol{\theta}')}. \tag{59}$$

M-step. The Q-function splits into separate terms for each parameter $\boldsymbol{\mu}^{(k)}, \boldsymbol{\Sigma}^{(k)}$ and $\pi^{(k)}$. In order to optimize the Q-function with respect to a parameter, we calculate the derivative and set it to zero. We obtain

$$\boldsymbol{\mu}^{(k)} = \frac{\sum_{i=1}^{N}\rho_{ki}\sum_{p=1}^{P}\boldsymbol{x}_{p_i}}{P\sum_{i=1}^{N}\rho_{ki}}$$

and

$$\boldsymbol{\Sigma}^{(k)} = \frac{\sum_{i=1}^{N}\rho_{ki}\sum_{p=1}^{P}\left(\boldsymbol{x}_{p_i} - \boldsymbol{\mu}^{(k)}\right)\left(\boldsymbol{x}_{p_i} - \boldsymbol{\mu}^{(k)}\right)^T}{P\sum_{i=1}^{N}\rho_{ki}}.$$

Although the next value for $\pi^{(k)}$ depends on the prior distribution on z and is as such not part of this class, we note that under a Categorical distribution (which is the most natural prior choice), $\pi^{(k)}$ is given by:

$$\pi^{(k)} = \frac{\sum_{i=1}^{N}\rho_{ki}}{N}.$$

The EM algorithm will then proceed as follows:

1. Initialization: Set $t = 0$ and choose $\boldsymbol{\theta}^{(t)}, \pi^{(t)}$ by the following algorithm:
   (a) First calculate the mean vector $\bar{\boldsymbol{x}}_i$ for each data vector across all parameters on which the prior is defined, $\bar{\boldsymbol{x}}_i = \frac{1}{P}\sum_{p=1}^{P}\boldsymbol{x}_{ip}$.
   (b) Assign the first data vector $\bar{\boldsymbol{x}}_1$ to the first component, $k = 1$.

(c) For all remaining components k, apply the following steps:

i. Go over all data vectors $\bar{\boldsymbol{x}}_i$. For each data vector $\bar{\boldsymbol{x}}_i$, store the minimal distance to all previous components (this corresponds to the distance to the closest component). To do so, calculate the Euclidean distance $D_M(\bar{\boldsymbol{x}}_i)$ to each previous component $j = 1, \dots, k-1$ with value m_{jd} by

$$D_M(\bar{\boldsymbol{x}}_i) = \sqrt{\sum_{d=1}^{D} (\bar{x}_{id} - m_{jd})^2}.$$

Then, take the minimal values among $D_M(\bar{\boldsymbol{x}}_i)$.

ii. The center of component k is now given as the value $\bar{\boldsymbol{x}}_i$ that maximizes $D_M(\bar{\boldsymbol{x}}_i)$ (i.e. we pick the component that is furthest away from all previous components).

(d) Finally, we go over all data again and assign each data point to its closest group, which serves as the initialization of the latent factors $\boldsymbol{z}$.

(e) We then calculate the mean and variance of each group and initialize $\boldsymbol{\mu}^{(k)}$ and $\boldsymbol{\Sigma}^{(k)}$ accordingly.

2. For $t = 1, \dots, T-1$:

(a) E-step: Calculate the EM-weight $\rho_{ki}^{(t)}$ according to (59).

(b) M-step: Update the parameters $\pi_{(t)}^{(k)}, \boldsymbol{\mu}_{(t)}^{(k)}$ and $\boldsymbol{\Sigma}_{(t)}^{(k)}$ as described above.

We stop the EM after 100 iterations or as soon as the difference between the old and new log likelihood is smaller than 10^{-3}. The log likelihood is obtained by integrating out the latent variable z:

$$\log \mathbb{P}(\boldsymbol{X}|\boldsymbol{\theta}, \boldsymbol{\pi}) = \sum_{k=1}^{K} \left(N \log \mathbb{P}(z = k|\pi^{(k)}) + \sum_{p=1}^{P} \sum_{i=1}^{N} \log \mathbb{P}(\boldsymbol{x}_{pi}|z = k, \boldsymbol{\theta}) \right).$$

We finally initialize $\boldsymbol{z}$ to the value that maximizes their posterior $\sum_{p=1}^{P} \mathbb{P}(z_i|x_{pi}, \boldsymbol{\theta}, \boldsymbol{\pi})$. For this, we simply re-compute the EM weights with (59) one more time (using the final estimates for $\boldsymbol{\theta}$ and $\boldsymbol{\pi}$) and pick the z_i that maximizes the posterior.
This should be solved using a general mixture model.

9.4 Adaptive burnin

A burnin has two major goals. First, it lets the Markov chain move towards its stationary distribution before actually sampling from the chain. Second, it optimizes the mixing of the chain by scaling the proposal kernels until an acceptance rate of approx. 0.44 is obtained. The first goal is difficult to evaluate. However, we can benefit from a significant speedup if we tune the burnins to reach the second goal as fast as possible, because burnins are "wasted" computational time, and we do not want to invest more iterations than necessary.
Estimating the acceptance rate is a binomial problem: In each MCMC iteration i, a parameter update is either accepted ($x = 1$) or rejected ($x = 0$), therefore following a Bernoulli distribution. The total number of accepted updates $T = \sum_{i=1}^{n} x_i$ then follows a Binomial distribution, $T \sim \text{Binom}(n, a)$. The acceptance rate a is calculated as $a = \frac{1}{n}T$
The idea of the following algorithm is to run multiple burnins and tune their length such that we get a reliable estimate of the acceptance rate. In general, the longer a single burnin, the smaller the binomial variance and the more precise the estimate of the acceptance rate. However, running

many long burnins is a bad strategy, because a precise estimate of the acceptance rate only makes sense if it is already close to the ideal acceptance rate of 0.44. If the acceptance rate is far off, it is better to stop the burnin, adjust the proposal kernels, and re-start it again.
We propose to run the burnin until some termination criteria is met. However, while running the burnin, we might realize at some point that there is no chance to reach the termination criteria after a fixed number of iteration. For example, when the acceptance rate after a couple of iterations is far away from 0.44, there is really no point to continue until the end, since we're anyways going to reject the critera. We therefore propose the following algorithm:

1. Run n_{adj} (typically few) burnin iterations.
2. Evaluate: continue or restart?
   (a) Calculate the values of k_l and k_u corresponding to the 90% quantile of the binomial distribution for the current iteration i and an acceptance rate of 0.44.
   (b) For each parameter: Check if the observed number of accepted updates k is within the interval $[k_l, k_u]$.
   (c) Continue burnin if a certain fraction q_{adj} of updates lies within the 90% quantile of the binomial distribution. Else re-start burnin.
3. Terminate? After a fixed number of iterations, n_{crit}:
   (a) Calculate the values of k_l and k_u corresponding to the 90% quantile of the binomial distribution for iteration n_{crit} and an acceptance rate of 0.44.
   (b) For each parameter: Check if the observed number of accepted updates k is within the interval $[k_l, k_u]$.
   (c) Terminate burnin if a certain fraction q_{crit} of updates lies within the 90% quantile of the binomial distribution. Else re-start burnin.

This is fully parametrized by n_{crit}, q_{crit}, n_{adj} and q_{adj}. Here, q_{crit} and q_{adj} can be the same if we want to be strict; or $q_{adj} < q_{crit}$ if we want the burnins to run longer before re-starting.

References

Anderson, T. W. (2003). *An Introduction to Multivariate Statistical Analysis*. Vol. 3. 1, p. 752.

Banterle, M. et al. (2015). "Accelerating Metropolis-Hastings Algorithms by Delayed Acceptance". In: *Foundations of Data Science* 1.2, pp. 103–128.

Barber, D. (June 2012). *Bayesian Reasoning and Machine Learning*. 1st ed. Cambridge University Press.

Brooks, S. et al. (2011). *Handbook of Markov Chain Monte Carlo*. 1st ed. New York: Chapman and Hall/CRC.

Cohen, A. C. (1967). "Estimation in Mixtures of Two Normal Distributions". In: *Technometrics* 9.1, pp. 15–28.

Du, Y. and R. Varadhan (Feb. 2020). "SQUAREM: An R Package for Off-the-Shelf Acceleration of EM, MM and Other EM-Like Monotone Algorithms". In: *Journal of Statistical Software* 92, pp. 1–41.

Galimberti, M. et al. (2020). "Detecting selection from linked sites using an F-model". In: *Genetics*.

Green, P. J. (1995). "Reversible Jump Markov Chain Monte Carlo Computation and Bayesian Model Determination". In: *Biometrika* 82.4, pp. 711–732. JSTOR: 2337340.

Hastings, W. K. (1970). "Monte carlo sampling methods using Markov chains and their applications". In: *Biometrika* 57.1, pp. 97–109.

Held, L. and D. Sabanés Bové (2014). *Applied statistical inference: Likelihood and bayes*. Vol. 9783642378, pp. 1–376.

Metropolis, N. et al. (1953). "Equation of state calculations by fast computing machines". In: *The Journal of Chemical Physics* 21.6, pp. 1087–1092.

Murphy, K. P. (2012). *Machine Learning: A Probabilistic Perspective*. Adaptive Computation and Machine Learning Series. Cambridge, MA: MIT Press.

Nelder, J. A. and R. Mead (1965). "A Simplex Method for Function Minimization". In: *The Computer Journal* 7.4, pp. 308–313.

Press, W. H. et al. (2007). *Numerical recipes 3rd edition: The art of scientific computing*. 3rd ed. New York: Cambridge University Press.

Sanderson, C. and R. Curtin (June 2016). "Armadillo: A Template-Based C++ Library for Linear Algebra". In: *The Journal of Open Source Software* 1.2, p. 26.

– (July 2019). "Practical Sparse Matrices in C++ with Hybrid Storage and Template-Based Expression Optimisation". In: *Mathematical and Computational Applications* 24.3, p. 70.

Suhov, Y. and M. Kelbert (2008). *Probability and Statistics by Example: Volume 2, Markov Chains: A Primer in Random Processes and Their Applications*.

Varadhan, R. and C. Roland (2008). "Simple and Globally Convergent Methods for Accelerating the Convergence of Any EM Algorithm". In: *Scandinavian Journal of Statistics* 35.2, pp. 335–353.

Wegmann, D. and C. Leuenberger (2019). "The Art of Statistical Modeling and Inference with Examples from Genetics". In: *Handbook ofStatistical Genomics*, pp. 1–52.