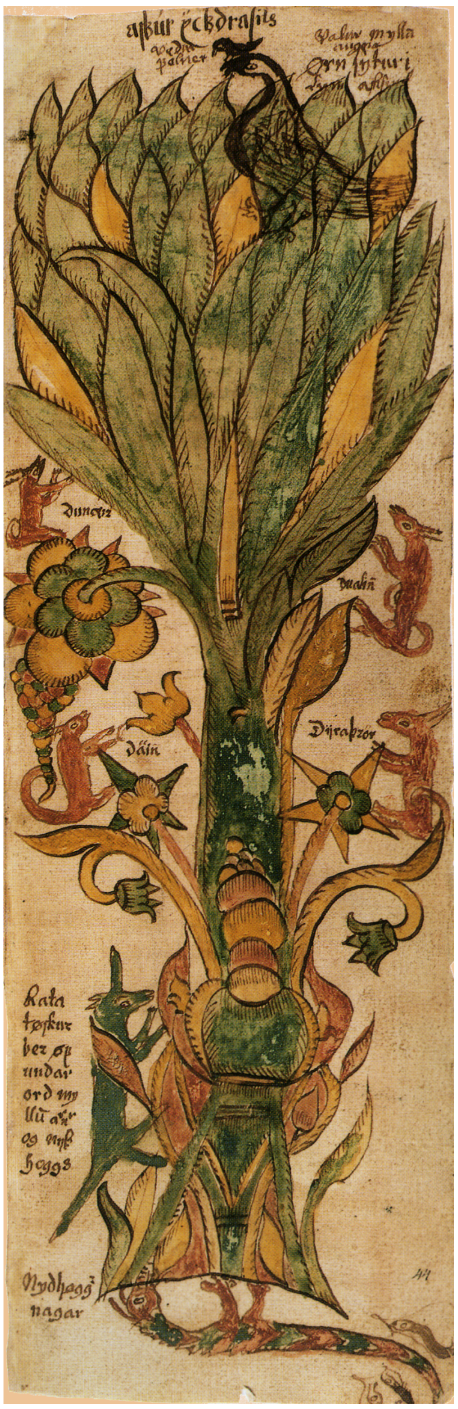
Yggdrasil Specifications

Philip Müller, 13-928-304

April 19, 2019



Front Image Shows YGGDRASIL the source is `https://de.wikipedia.org/wiki/Yggdrasil#/media/File:AM_738_4to_Yggdrasill.png`.

1 Scope of This Document

The purpose of this document is to describes how the program will be implemented. This is done such that the specifications are clear and an outline is available.

2 General Problem Analysis

Here we want to give an analysis of the problem.

2.1 Main Idea of the Method

The goal of the method is to estimate the probability density ~~distribution~~ (PDF) form a set of samples[1]. In the following we assume that we have ~~N~~ many samples. The samples are distribution in a bounded domain $\Omega \subset \mathbb{R}^d$. It is important to note that we assume that the domain Ω is a hypercube. In mathematical terms this is given as.

$$\Omega = \coprod_{i=1}^{d} \left[x_i^{(l)}, x_i^{(u)} \right] \tag{1}$$

In equation (1) $\coprod$ is the cross product[2]. $x_i^{(l)}$ is the lowest value in the i component[3] of all samples.

We then split the Ω into smaller hypercubes[4] C_k, that do not overlap.

$$\Omega = \bigsqcup_{k=1}^{m} C_k \tag{2}$$

The main advantages of this method is now, that inside such a small hypercube, C_k, we can use *simple* method to approximate the density function~~s~~. Thus the samples that fall~~s~~ into one hypercube C_k and the one in another hypercube $C_{k'}$, have nothing to do with each other.

We strengthen this kind of reasoning even more. We have d dimensional samples. So if we have n_k[5] many samples in cube C_k we have ~~$n \cdot k$~~ many numbers.
Now we assume that we have *pairwise independence* of the dimensions. This is an approximation to mutual independence. The reason for just considering pairwise instead of the much stronger mutual independence is, that it is cheaper to test for.
This approximation allows us to consider each dimension separately from each other.

In conclusion this means, instead of considering ~~N~~ $\cdot\, d$ many samples at once~~. We~~ have to consider only n_k many samples at once, but we have to do this $m \cdot d$ times.

2.1.1 Building of the Estimator

Start with some samples. Further we have a certain (simple) model in mind, how the distribution in a distribution element (hypercube) should look like.

We then appl~~ies~~ ~~or~~ model to all data in the current cube under consideration. We can do this for each dimension separately.
Then we estimate how well the model explains or fit~~,~~ the data. This can be tested with a goodness-of-fit (gof) test, such as the χ^2 test.

Especially at the beginning it is rather unlikely that the test will pass. But regarding what the current cube is, a failing of the gof test, indicates that the model can not explain the data well enough. In this case we thus split the current hypercube into ~~a certain number of~~ smaller cubes, that completely fill the old cube. For a description of the procedure see section 2.1.2 on page 4.

This is applied recursively to the newly created cubes until the gof tests do~~es~~ not reject our model. To verify that our second assumption, of the pairwise independence is justified, we apply independence tests. If they also fail, we again split the cube in ~~half and start anew~~. For a discussion of the steps of the test and splitting see section 2.1.3 on page 4.

It is relatively clear that the result of this method is a tree that is called "The Distribution Element Tree". The internal nodes of this trees represents cuts along one dimension. The leaves or final nodes represent~~s~~ the hypercubes that composes Ω.

[1] In the following we refer to them just as samples.
[2] LaTeX does not have a big cross product.
[3] Dimension
[4] An other name for hypercubes in this contexts is "distribution element".
[5] We define $n(k)$ as the function that returns the numbers of samples that are located in hypercube k. when we write n_k, then we refer to "the number of samples in a hypercube in a general sense", the contest should clarify its meaning.

2.1.2 Splitting a Hypercube

Here we discuss the procedure if some of the gof tests w~~h~~ere rejected. Since we assume independence it is very suggestive to split the hypercube along the dimension, where the gof tests w~~h~~ere rejected. Since this will only affect the dimensions along which the test failed and act as a refinement step.

To prevent exponential grow of the elements, we will not consider all failed tests, but only ~~two.~~ This implies that each cube will only be split in at most 4 smaller cubes.

Since we perform a statistical test, we will get p-values from them. Let be p_j the p-value that resulted from the test of dimension j. We will reject the null hypotheses[6] for dimension j if p_j is below a ~~certain value~~. ~~The~~ smaller p_j ~~is the~~ stronger ~~was the~~ rejection of the null hypothesis. This means that it makes sense to split along these axis first which has had a small p-value. Let be j_0 be the dimension which has the smallest p-value and j_1 the dimension with the second smallest value.
Since we are in a computer we must be prepared of p-values that are very close, so close that they are from a numerical point of view have to be considered equal. To handle such cases we need a tie breaker. We have chosen a scheme which also consider the geometrical properties of an element. Figuring out exactly what is done.

Not As I see the thing now, function `dimtosplit` only splits along one dimension, see line 222, which contains a break.

The splitting is now done the following way.

1. We split the current cube along dimension j_0 into two pieces. This results in two new cubes $C_k^{(L)}$ for the left[7] and $C_k^{(R)}$, for the samples on the right.

2. In the case that we have to split along a second dimension, j_1. In that case we split both cubes $C_k^{(R)}$ and $C_k^{(L)}$ again, but this time along dimension j_1. Thus we have created 4 cubes.

2.1.3 Testing for Independence

When the gof tests accept~~s~~ our prameteric model[8] then we have to verify, that the dimensions are pairwise independent with each other. This means we have to perform $d \cdot (d-1)/2$ many independence tests.

In the case that ~~they are not~~ rejected we are done with the element and consider the element as final. However in the case when some tests fails, we have to split again.

In a first step we determine which of the pairwise tests was rejected the most, meaning has the smallest p-value. As tie breaker we again use the geometry of the elements.

The interesting observation now is, that we have to do two splits, since two dimensions are involved Figuring out the order in which we split. .

We then start to process the 4 new elements.

2.1.4 Finding the Split Position

We now want also discuss how we determine the position where we split. In this discussion we assume that we split along dimension j.

Size Based This is the simplest way of spiting. Assume that the hypercube occupies the interval $\left[x_j^{(l)}, x_j^{(u)}\right]$, in dimension j. So now we simply cut it in halve. Lets define $x_j^{(m)} := \frac{x_j^{(u)} - x_j^{(l)}}{2}$, then the left hypercube spans $\left[x_j^{(l)}, x_j^{(m)}\right[$ and the right one $\left[x_j^{(m)}, x_j^{(u)}\right]$.

Score Based The scheme that we outlined above, is simple but it does not considers the distribution of the samples. So the other, still simple approach is, to determine the split location $x_j^{(m)}$ such, that the numbers of samples in both hypercubes is similar. Now lets denote x_j^i the jth component of of the ith samples in the hypercube under consideration. Then we define $x_j^{(m)}$ simply by the median~~.~~

$$x_j^{(m)} := \operatorname{median}\left(\{\, x_j^i \,\}_{i=1}^{n(k)}\right) \tag{3}$$

[6]Our simple parametric model of the cube.
[7]All samples that are smaller than the splitting point.
[8]In all dimensions.

2.1.5 Inserting

The method presented in the original paper did not discuss inserting of new samples, there the data is static. We will now discuss how the insertions of samples works in principle.

First of all a sample is just a point, so it is determined from the very beginning in which hypercube the samples belongs to. So a first idea would be to just recheck the cube where the sample is finally inserted and perform further splits as needed.

However there is a problem to this approach, to be blunt it is wrong. The tree is buil~~d,~~ by testing if the samples fit into a certain parametric model and that they are independent. Thus a sample could have a very strong impact on the cuts. It could for examples shift the p values in such a way that now different cuts have to be performed as before.

To find the hypercube where we insert the sample we traverse the tree. At each internal node we check if we have to go to the right or to the left. Instead of just following the path further down the tree, we have to perform the gof test on all samples below that node and on the additional new sample. If the test still produces the same result, we can go to the next node. Until we come to a final node where we can insert the sample~~s~~. In order to do the test, we have also to recompute the parametric model of that node[9].
We have to do this until we reach the leaf to which this samples belongs to. ~~We then have to perform two tests! First of all we must redo the gof test to see if the samples can still be sufficiently explained by the parametric model. If the model is accepted we have also to redo the independence test.~~

However there is also the possibility that the test that formed the split did not fail or is not the one that is rejected the most. In that case we have to reject the sub tree that is below this cut in its entirely. From a conceptional point of view, we merge all hypercubes below that split[10] and start ~~anew. This means we test if the parametric model is sufficient for explaining the data and split if needed. Then we finally perform the independence tests.~~ ~~Is there anything special when only one test fails or the second dimension that is selected for splitting changes?~~

What happens if a new sample lies outside the box, then we will need to change a lot.

2.2 Main Observations

In section 2.1 we have outlined the method. Now we want to summarize some important points that are important constraints for choosing the appropriate implementation strategy. They will be collected here such that they can be addressed later.

1. The main observation is that we have a tree structure. Further we note that subtrees which are truly distinct[11] do~~es~~ not influence each other.
   This means refining one part of the tree[12] does not influences other parts, which are not subtrees.

2. During normal operations, we do not need all the samples. Instead we just need the samples that are located in the current cube.

3. Except from splitting or merging cubes we do not work with the a complete sample, meaning the d numbers that forms the sample vector. Instead we work with the components of the samples, but with all samples at one.
   Putting it differently, instead of thinking of n_k vectors of dimension d, it is more natural to think of d vectors of dimension n_k.

4. Inserting a new sample could trigger a complete rebuild of the tree. But all parts that ~~roost~~ at splits which are passed[13] are unaffected by this.

5. The operations that forms a split, meaning the determining of the location are only determined by a single dimension per split.

3 Implementation; A High Level Overview

In this section we want to give a high level description about the implementation. It is important to note that this section, does not discusses the actual implementation, rather it discusses the strategies of the implementation. More important it outlies the strength and the weaknesses of ~~the~~ approaches.

[9]As before this involves all samples that are below the node and the new sample.

[10]In a sense we undo all the splits.

[11]I came up with this term myself, is there a correct one? It means that neither of the trees is a subtree off one of the others. Meaning that the disjunction of the two vertex sets is always empty.

[12]Meaning performing a split.

[13]The test result is still the same.

3.1 General Restrictions

First of all we will here present the general restrictions~~. T~~hat arise~~s~~ purely form technical reasons.

Fundamental Data Types We store and process everything in `double`, ~~why~~ this we mean the type that is used when we write `double` in `C` or `C++`. At the majority of the systems this corresponds to double precision which usually occupies 64 bits.

However to allow changing we will use a `typedef` declaration, to change this data type.

Counting For counting we will use the `size_t` type from `C`. This is usually a unsigned integer that is 8 bytes long. The standard guarantees that this value suffices to index any array.

We do not guarantee that the implementation will be able to handle that many samples, even in the presence of sufficient memory.

Address Space We require that the address space of the user space is less than 54 bits. The consequences of this is, that for *user space code* the highest 8 bits of an address are always zero.

Note that is usually fulfilled.

Numbers of Dimensions We work with vectors that have a certain length. We restrict the number of dimensions a sample can have to $256 = 2^8$. This choice allows us to store the length of the vector inside the pointer.

Language The implementation will be, for many reasons be `C++`. The version is at least `C++11`. This is done to use the bindings to python and R.

External Dependencies For establishing the binding to python `PyBind11` is used. The bindings to R are implemented with `Rcpp11`. These are dependencies that are required.

The consequences of not using them, would be that an imense amount of time would be needed to using the old[14] native `C` interfaces. This time will be *lost* and could have been used for other features, as the paged array.

For several reasons I would like to have `Boost`. Since we targeting scientist, so assuming they do not have `Boost` is like we assume that they do not have `fftw`.

3.2 Specific Considerations

In this section we will discuss the implementation in much more detail and we will focusses on specific aspects.

3.2.1 Storage of the Samples

Where we will only discuss the memory order we use. How and more importantly where we store the samples is discussed in section 3.2.2 on page 7. The samples could be seen as a matrix, but there are two ways to lay a matrix into memory, row-major or column-major.

For us this means, store we the samples consecutive or store we the dimensions consecutive in memory.

As we have noticed in point 3 of the summary in section 2.2 on page 5, the only point where we have to access the whole ~~sample at~~ ~~one~~[15] is when we split. At all other occasions, ~~we do not need one samples,~~ we only need one component of a sample.
So the idea of storing the dimension consecutive suggests itself.

The advantages are the following

- When we compute the statistics we have access to the data in a form as we need it for processing. This means that each memory load contains only data we need.
  Assume we have an 8 dimensional samples. Then we would need to load all 8 `double` but we need only one.

- We enable a form of potential parallelism. We assume that the ~~the~~ dimensions of the samples are independent of each other. This means that we can process each dimension in parallel. This would eliminate the need for synchronization.

However we also have a~~n~~ disadvantage[16] that arises from this choice.

[14]Ugly
[15]Actually we can also lift that requirement.
[16]I thought about it, but I did not find another one. If somebody sees one, please inform me.

- When we split[17] we have to move data around. When we keep the sample in one piece, then we can process a sample at once and we are done. If we split it, then we have to move something around for each iteration.
- Accessing a whole sample is relatively expensive, since it involved d memory accesses.

Discussion of Disadvantages It is true that we can process a sample at once. Data movement is expensive and slow. In order to move one sample at its final location we need to load d `double` from memory and write d `double` back. Assuming that memory will be our bottleneck we have to wait until we finished the transaction. If we split the samples, the moving only involves 2 `double`. This means that the CPU is not blocked long.

The location where we put a sample is deterministic and can be computed for all samples in advance. It is also so, that this decision is the same for all dimensions.
So we can precomputed it and optimize it. Then this plan can be executed for all dimensions, at once.

It is true that accessing a single sample, in its entirely becomes relatively expensive. But we do not need that capability. So it is like complaining that a toaster can not make coffee.

3.2.2 Organization of the Samples

How do we organizing the samples? The R code uses one array, that is reordered upon spiting. This is nice since one has only to store two numbers for each leaf, the positions where the samples start and how many[18].

This simple scheme starts to fail miserably as soon as we want to insert a sample. To see this consider the situation where you want to insert the sample. You can not swap it in, since this would change~~s~~ *all* offset of leafs that are stored after the position. You have to update it, every single reference. You also have to put there a heavy lo~~o~~k and nobody can do anything, since the data is inconsistent, and it is in a way that hinders you to do any thing else.
Also inserting empty spots into the array will not save it. Because what you do when the free space is used? You have to update it, you have to lock it you have to pay for it.

The next idea would be, that each leaf stores more information. Instead of just the start and the end of the range of samples that belongs to it. It stores the indexes of *all* samples that belong~~s~~ to that leaf.
The problem is that, then we do not have consecutive memory of the sample inside the array. Which can severely hinder performance. But it allows to insert samples without costly synchronization.

What is also a downside of having some single point of reference is, that when we have to grow this structure. Then we have to make sure that nothing accesses the structure such that we can update it.

The discussion above shows that a very important requirement that is imposed by the ability to add samples, is, that one can manipulate the sample storage without much cost and at the same time, does not hinder other entities.

Going with the idea of the tree, we split the sample array into many small arrays. We then establish a 1 to 1 correspondence between an array and a leaf node. Choosing this implementation, results in the following advantages.

- The samples of different subtrees are completely decoupled, since they are in different arrays. So one can reorganized different subtrees in parallel, without synchronization.
- When we have to expand an array, we do not need to allocate a very big array coping all the samples around and then deleting the old array and lock it.
- As long as we are alone in a subtree, we know that we do not have race conditions with threads that works in another subtree. We can basically do anything we want there without synchronizing.
- We can merge all leafs beneath an inner node, without synchronization. If you have one array then everything is blocked.
- ~~a~~fter the first step, when we work with smaller sizes, then we need less additional memory

We want to stress that the single and biggest advantage~~s~~ of this method is that subtrees are completely *decoupled*.
As before we now comes to the disadvantages of this implementation.

- In the initial step, we will need at twice as much memory as we would need.
- Merging involves coping of many data.
- The data is not localized anymore and could involve~~s~~ a lot of loads.

[17]Merging is easy regardless how we do it.
[18]One could also store where it ends.

Discussion of the Disadvantages Now we want to discuss the disadvantages from above, as we have d~~id~~ it before.

First of all merging involves always the movement of data. But if we do not decouple the data, then we will have to make sure that nobody else is working on the array. This is needed ~~that~~ when we update~~s~~ all references, we have no inconsistency of merges that run~~s~~ parallel. I agree we have to move more data, but we can do it parallel.

It is true that when we perform a split or a merge, there is a certain time span, where we need twice as much memory, as when we just would work with one array. However when we would grow the array, we also would need that much memory, to be honest even more.

It is true that something distributed is costlier. However there is also a trade of that one has to consider, since it allows to remove complexity form the implementation. To address this issue one could use preallocation for the nodes.
Another approach is that one does not store everything in the node itself. The node just caries enough information, that it can be used, like the split location, together with the dimension. This way only the necessary information is loaded. If more is needed the payload can be loaded.

3.2.3 Paged Array

This is an approach that will ~~smaller~~ the impact of the problems that are discussed above. I consider this as an important step in the implementation, but not the first one, and I would prefer that we consider this as a secondary goal.

The main idea is the same as the virtual memory system. For a program the memory looks like a continuous array, but ~~this is not true~~. In order to allow multitasking the memory is split into so called pages, a page has usually a size of 4 KB. When the user allocates memory, for example by using the `new` operator or `malloc`, for example 1 Byte. The operation system does not return one byte. Instead it returns a full page, the C library then manages the memory inside a page.

The idea of the paged array is now to mimic this behaviour. Instead of having a plain array for a set of samples, we would have many pages. Arrays would be just convenient interfaces to them. This would make merging very cheap, in a first approximation, since it contains only of merging the references.

As I said I consider this as a secondary goal. The first goal should be a working prototype, that uses classical arrays. But if this goal is reached, the paged array should be something that comes very soon afterwards.

However for domains where we have not so much samples, then we will have some waist.

Consequences There is a consequences, a very important one too. The memory is not consecutive, it looks only as if, with some help. To enable this, in a reasonable way, we need `C++`, since it allows to implement iterators, that are not mere pointers. The compiler will optimize a lot of the over head away, if done right[19].

3.2.4 Inserting

Inserting is something that we want to do. As we have seen inserting can be rather costly, since it can trigger~~s~~ the rebuilding of the whole tree. So checking the whole tree after each insertion might not be a good idea. Assume a for loop that inserts samples one by one and each time the entire tree is rebuild. It would be much more intelligent to just insert them, or store them in some temporary structure. And after the insertion is completed one can simply trigger one bigger update.

This would mean that the estimator can now be in some dirty or inconsistent state, but this is only a technicality. When one has to insert that single sample and then to make an estimate, then we have no choice, we have to do it. But if one knows that we have to add a million samples, and during that, we does not have to make a query to the tree, there is no read operation that depends on these new samples, there is just no~~w~~ reason to reprocess the tree after every single insertion.

Another statement we have to ma~~d~~e is that inserting is serial in some sense. Building the tree should be used by employing the parallelism of the system. By that we mean that we ~~not~~ support threads that insert~~ing~~ data in a concurrent way. Many threads could potentially insert data, but they will be completely serialized[20] by a lock.

It is important that we just have to test the nodes that w~~h~~ere touched by the search for the final location. The inserting of new samples basically works in the following fashion.
When we insert we walk down the tree, as we wanted to search for the leaf to which this samples belongs to. We will mark each internal node as “dirty” as we walk down. When we have found the leaf we will insert the new sample there.
When we want to rebuild or update the tree then we just have to redo the tests that are associated to nodes,

[19]This means using the `Boost` iterator library.
[20]I have a very simple idea that could help here, but this is something for later.

which where touched during the insertion. We can do this by recursively go down. We also see that here we have another form of parallelism there. As soon as we found a split which does not result in the same result as before we stop. We can then merge all leafs below that split, turn it into a single big leaf, that was one there and then start the splitting process that was discussed in section 2.1.1 on page 3.

We also see that there is a problem to this idea. It involves the recomputation of every test statistics in the tree. Since we have not direct access to them[21], it is not a simple `for` loop any more[22]. However mathematics has some iterative nature. When we look at some test statistics then we see that they process each sample at once and that some does not have to know the exact number~~s~~ of samples in advance[23]. The trick is that we have to store the statistic in a form that allows to update the statistics when new samples become available.

The advantages of these scheme are the following.

- It is cheap and conservative. We obviously do not rebuild the tree if we do not need it.
- The checking results in a natural parallelism, that is task based.
- We have to use tests that are updatable, such that we have to store them. This does not simply involve~~s~~ the statistics, but also has to hold for the parametric models.

As before ~~we~~ the approaches also result~~s~~ in some disadvantages.

- The estimator can be in an inconstant state. This means that the samples not necessarily corresponds to the estimator.
- Since we cache the model and the statistic at each node, we will increase the storage.

Discussion of the Disadvantages I have to agree that this is true, but as it was explained above, if we do not care if the tree is consistent or no~~r~~ this does not matter and is completely irrelevant. To address concerns, one could set the default to "update always" and the user has specifically to request that no update is performed.

It is true that we increase the storage, requirements of the node. But at the same time we also potentially lower the computational need. So we have to trade.

3.2.5 The Degree of Parallelism

Here we want to discuss the parallelism that we are exploit~~ed~~. As we have seen in the construction of the estimator, section 2.1.1 on page 3, that it is a recursive way. We have a form of parallelism that is called "task based" parallelism.

By that we mean we have a potential leaf, so the work flow is that we create a task, job or assignment for "checking leaf *xy*", we can then simply insert~~ing~~ this into a queue. A thread will then grab this task from the queue and execute this. Meaning performing the gof tests and if they are passed the independence test. If the tests are completed, we are done.
In the case of a failing, meaning that the null hypothesis is rejected. The thread will perform the splitting[24], and generate the four new tentative leafs. Instead of performing the tests on them itself, the thread will generate jobs for the four tests and inserting it into the queue.

Performing the update of the tree, after some insertion can also be decomposed into tasks.

As was mentioned before since the dimensions are independent they could also be processed in parallel. However they are not really task based, they can be seen as task based, but there arises dependency. Since one can not continue until *all* are completed. So it would be more appropriate to use a simple `#pargma omp parallel for`.

So the best way to implement this is a pool of threads that pull tasks from a queue. Each of these POSIX threads[25], has some `omp` threads.
As the reader might have noticed I explicitly wrote POSIX thread for the thread pool. `openMP` offers task but for me they are an enigma. `openMP` is good when you have a small loop that you want to parallelize, but for anything that is more complex it is just the wrong way. I will try to use them in a first step, since they are platform independent in theory, but I think that they are not the right choice.
By the way `Boost` offers thread pools.

Now we again come~~s~~ to the advantages of choosing the approach outlined above.

- We exploit the inherent parallelism of the method.

[21]They are still in a format that allows access them easily.
[22]But it is only slightly more complicated, see below.
[23]An example is the mean, where one only has to store the sum of all samples and the number of all samples that we have processed yet. We can simply update this statistic.
[24]This could also be done a job, but I does not like the idea.
[25]`C++11` introduced the `std::thread` class, which offers an operating system independent interface to threads.

- At least the parts that uses `openMP` are very portable at least as `openMP` is.

Also it has some disadvantages.

- As I have stated above, I consider `openMP` good for simple loop but wrong for anything more complex. So when we have to roll out our own, which is not completely impossible, but will take its time. We should use something that is offered by `Boost`, which is also portable.
- Inserting[26] is serial. This is discussed further down.

Discussion of the Disadvantages As I have said, I will try to use `openMP` first, but as I have written several time, I do not like it.

What is not parallel Here we want to give a small discussion about what is parallel in the first place. We have stated that building and update as some natural form of parallelism, that is also exploited here.

Density queries can also be done in parallel. Since they are only read access and thus we will not have a data race. So many threads can potentially access the data.

What is not parallel is the process of *adding* new data. This will be protected by a lock. The reason for doing that is that it lowers the requirements on the update function of both the test statistic and the parametric models.
One could improve that, by turning the insertion into a task and then just issue the task. Behind the scene the process will still be serialized, but for the user it looks as if the process would work concurrently. One also has to synchronize that with the updating, meaning that the update is postponed until all insertions has taken place.

3.2.6 Test statistics

As we have seen to speed up the update process, we should be able to update the test statistics. As far as I can tell this is possible to some degree for the χ^2 test. What is also important is, that we *must* save the result such that we can compare it with the new result.

So the interface of the test statistics will have a function which only *adds* the new samples to the underling data, that composes the test statistic. This is similar to the postponed updating, this means that the test statistic also can be in an inconsistent state.
Then after all insertions have taken place and we updating the tree, the interface also has to provide, a function to do it. This is as performing the last step of computing the p-value out of the test statistics. In order to potentially allow test statistics which are not updatable, we will also pass all the data, to this function such that the statistics could be computed.

It is important to note that the splits only have to cache the gof test statistics. However the leafs have to cache[27] the independent test.

It ia also very important that the test statistic should not operate on simple arrays, but on subtrees. Subtrees allows an interface to all the leaf. The iterator concept of `C++` will allow to write it in a nice way[28].

Please note that we also have to interact with the parametric model.

The advantages.

- The updatable concept allows us, to cache the statistic and avoids a costly recomputing of the whole statistics.
- The updatable and commit concept, which we apply joins our tentative concept of the interface.

Now the disadvantages.

- Since we have to cache the test statistics, we have to store more data in the inner nodes.
- The test statistic has the bo slightly complicated than necessary, since it needs to operate on subtrees instead of arrays. There must also be logic for the update.

Discussion of the Disadvantages Here we again discuss the disadvantages of the choices.

The decision is we cache or not, has some balance aspect in it. It is only worth if the space that is needed is not to large and if recomputing is to expensive. I think that at some point one has to disable it, for example a leaf does not really need it. But this is a point where only tests can show us the result.

The important word here is *slightly*. Since we have to operate on subtrees instead of arrays. When done right, with iterator (provided by `boost`), the overhead is there but it is not that large. As I have explained above, it turns a single loop into a two folded loop.

[26] Only the act of adding data, not the updating.
[27] If they do.
[28] It is a two folded loop, looping over all the leafs and in the inner loop over all the samples.

Used Statistics In this work we focuses on the χ^2, it is also the only test that is implemented. However when we designing the test, we will make sure, that potentially other statistics could be used. This is done by implementing a general interface and a factory like approach to create the instances.

Note on Thread Safety It is required that the object is thread safe when considering the dimensions. This means that it ~~should~~ should be safe to call multiple modifying functions at the same time, *iff* they act on separate dimensions. So calling some modifying function concurrently on the same dimension will result in a data race and undefined behaviour.

3.2.7 Implementation Hint

The best way to ensure that is that object does not have a state, and the state is passed as an argument. And the state is owned by a node. This could be done by employing the strategy pattern, which is complete state less.

Note on Parallelism As we have explained above, we have some sort of parallelism in the testing. Since we can potentially test in parallel. We thus allow that the function which performs the tests, can spawn threads with openMP. However the implementation of the object is not allowed to set the limit by itself. It should be a parameter.

3.2.8 Parametric model

The model is coupled with the implementation of the test statistic, but we will try to keep this coupling small. But much of the design choices that governs the test statistics should be also applied here.

The basic functionality that we need is that we are able to evaluate the model at some point. We also allow that the model does not have to ensure that the argument lies in the hypercube that is governed by the model. This means if $\vec{x} \notin C_k$ then we have unified behaviour. However implementations are encouraged to check at least if the result is plausible.

As we have seen with the test statistics we have to enable tentative update scheme. Since we have to redo the gof test each time an update is performed at any node that was touched by the insertion, each node, regardless if it is a leaf or not has to store the parametric model, that represents the results of all the samples beneath it. We also see that the model also need to be updatable.
As before this update should not take effect immediately, but only after the update of the tree was triggered.

We also see that the implementation should, at least in a first state, not rel~~ies~~ on arrays but on subtrees instead. As it was explained above this increases the complexity only by a very small factor.

The advantages and disadvantages are essentially the same as for the tests, so we redirect the reader to these sections.

Note on Thread Safety As before we require that calling non modifying functions should be thread safe in any case. Another requirement is modifing function should not result in a data race when the dimensions that are modified are different.

Note on Parallelism As we have explained above, we have some sort of parallelism in computing the model parameter. Since they only depend~~s~~ on one dimension. We thus allow that the function which computes the parameter, can spawn threads with openMP. However the implementation of the object is not allowed to set the limit by itself. It should be a parameter.

3.2.9 Summary

In this section we want to summarize the discussion from above.

- The data structure which is employed is a tree. The samples are stored inside the leafs.
  There is no `FORTRAN` style array politics.
- Parallelism is only in the building process. The usage case is that a single entity uses the object.
  However by design the functions which does not involves modification[29] should be technically able to be used concurrently.
- Inserting is implemented in a transaction based fashion. Meaning that we add new data and the tree will be in an inconsistent state. When the adding is finished, the changes are committed and the tree is rebuild.

[29]By this we mark all function which are declared `const`.

- During rebuilding the tree can not be used. Before the rebuilding is initiated, all accessed has to be finished. Violation of this rule will not be checked for, and results in undefined behaviour.
- The tests and models must be implemented in an updatable way.
- Until the implementation of the paged array, we consider subtrees as the highest collection of data. And objects that interacts with data must be able to operate on them.
- The parallelism that is used is task based. For that `openMP` should be used[30].
- This is only a side note, but the program will make use of the pimpl idiom.

4 Project Outline

In this section we will discuss~~es~~ the next steps that ~~are~~ need~~ed~~ to ~~do.~~ Also we will outline some functionality that needs to be implemented. This is only the main functionality. The majority of code, will then be used to glue the parts together.

4.1 Main Classes

In this section we will describe the main classes that are needed to be implemented.

4.1.1 Node

The node is one of the most important classes. As we have explained above, we should design that it is as small as possible, such that loading it can happens fast.

The data that is also stored in the node is then accessed in a second step, by means of a pointer. This avoids the loading of unnecessary data.

4.1.2 Tree

This class is responsible as an interface, of the functionality of the library. It methods to access ~~that~~ samples and handle queries.

It is also responsible for the data that is global for the whole tree.

4.1.3 Subtree

This class is similar to a tree, but it is different in the way that it does not ~~won~~ the data It is an interface that allows to virtual merges the leaf into one. This is done by providing iterators.

4.1.4 Parametric Models and Tests

We have described the tests and the parametric models in detail above. Here we just want to say that they need~~ed~~ to be updatable.

4.2 Bindings

In this section we will shortly discusses the bindings. However we have discussed them before, but just shortly.

I strongly recommend to use some tested, well known methods to create the bindings. Both languages offers a `C` interface, but especially in the case of R, they are old, outdate, hard to use and error prone. Using them will waist a lot of time.

4.2.1 Python

For the python bindings one should use `pybind11`, `https://github.com/pybind/pybind11`. It is very easy to use and the integration is good, I have already tested it.

And it is written by a guy who understand python why more better than I do. Probably also `C++`.

[30] I would not, but we will see.

4.2.2 R

The R `C`-interface is very ugly and I strogly advi~~vi~~se against it. Instead I propose to use `Rcpp11`. It seams *the* interface between R and `C++`. There is a great deal of documentation, even a book!.

It is designed similarly to the python interface, but does not offer~~s thi~~s much functionality. But it works, the interface will not look that clean as in python, but it will work. It is written by someone who understand R far more better than I do.

4.3 Roadmap for the Implementation

In this section we want propose a roadmap that shows the steps of the implementation.

1. First a simple serial implementation. This involves implementing a framework that is designed such, that it can be further improved. It is important that not jet all of the strategies to tackle~~s~~ the disadvantages are implemented.
   Also inserting is not yet implemented.

2. Then we set up some testing framework, to see if the implementation behaves as expected. I think that we do not have to redo the full tests that are shown in the paper, but some of them should be fine.

3. Until this point we have basically ported the code from R to `C++`.

4. We then implement the inserting routines. This is still the dump version, that does not use the paged array, but operates on sub trees. It depends on ow it is going if we already implement caching or not. This should be decided if one has some more experience, with the tests.

5. We then implement the bindings to R and python.

6. In this step we implement parallelism into the building process.

7. In this step we implement the paged array. We think that from all of the explained tricks this will be the most valuable.

8. In a last step we implement some other tweaks to improve performance.

4.4 Recommendations

My recommendations are we implement it as I have proposed in this document. Further I propose that we make `boost` to our dependency. As I have said, `boost` is not small, but ....

I also recommend to lift the requirement from `C++11` to `C++14`. The reason is quite simple, since `C++14` is basically all the stuff they forget to add in `C++11` and made wrong the first time.