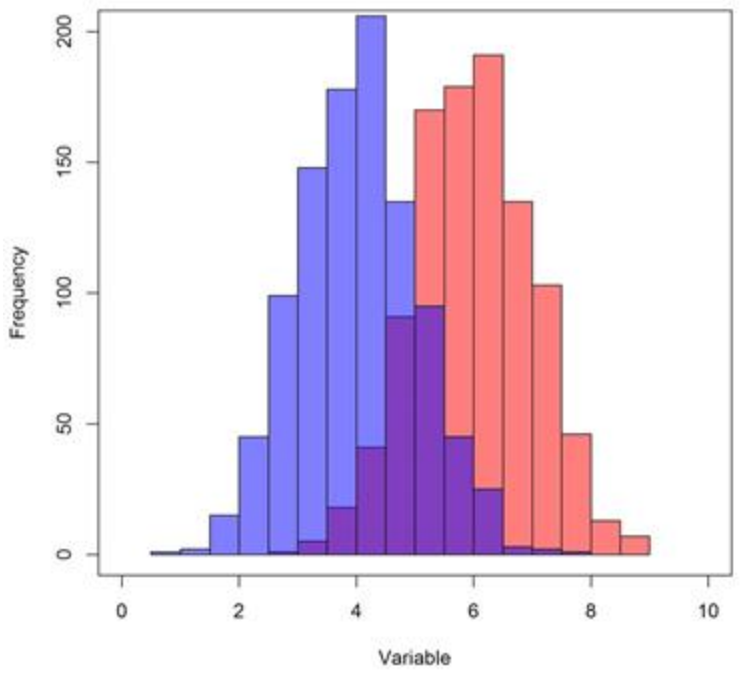
# The Bayesian Block Algorithm

discover local structures in background data

Raffaele Gaudio
2057974

## The problem: data visualization

- Sometimes it’s **hard**!
- What is the best data segmentation for **our** data?
- Am I using all the information I have?



## The Bayesian Block representation

- The Bayesian Blocks representation is a **non parametric** representation of data derived with a **bayesian statistical procedure**.

- Invented by Jeffrey D. Scargle and applied in the context of astronomical time series analysis [1].

## Features in our wish list

- **Non-parametric**: generic representation of data
- Discover local structures in background data **exploiting the full information** brought by the data
- Impose **few preconditions as possible** on signal and background shapes
- Handle arbitrary sampling and dynamic ranges of data
- Operate in a **bayesian framework** and work with posterior probabilities:

$$P(M|D) \propto P(D|M)P(M)$$

# Idea of the BB algorithm

Segmentation of the data interval into **variable-sized blocks**, each block containing consecutive data elements satisfying some well-defined criterion.

The optimal segmentation is the one that maximize some quantification of this criterion.

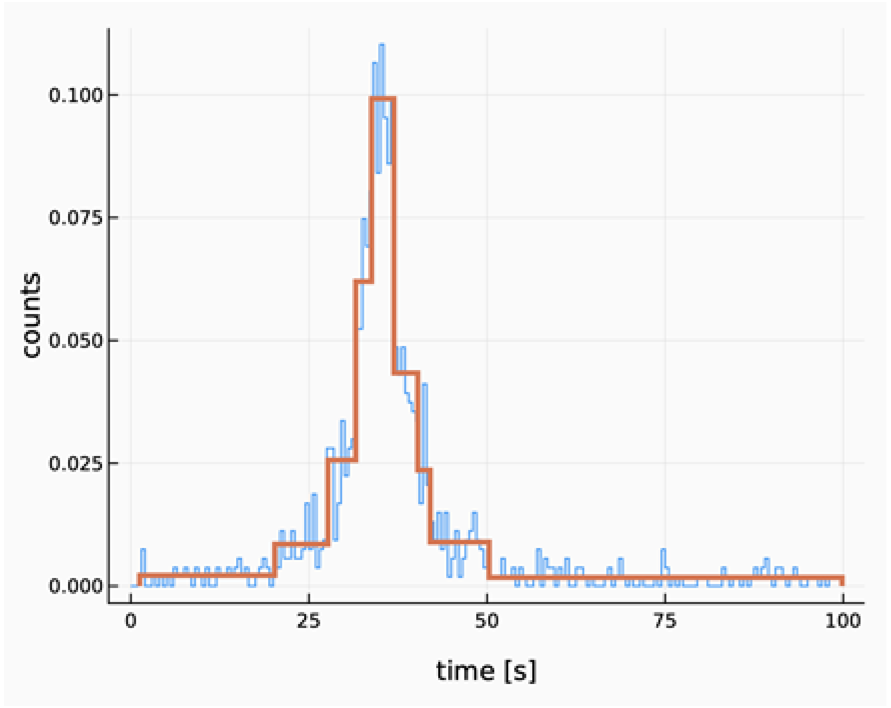


Image by Luigi Pertoldi [pertoldi@pd.infn.it]

# The Piecewise Constant Model

The range of the independent variable (e.g. time) is divided into subintervals generally unequal in size, in which the dependent variable (e.g. intensity) is modeled as constant within errors.

- the blocks can be treated **independently** (block's fitness depends on its data only)

- Our model for each block has two parameters:
  - the **signal amplitude**, and is treated as a nuisance parameter
  - the **length** of the interval spanned by the block

**Recipe**: build the fitness function and than marginalize (maximize/minimize) w.r.t. the nuisance parameters

## The fitness function

The algorithm relies on the additivity of the fitness function:

$$F_{total} = \sum_{k=1}^{N_{blocks}} f(B_k)$$

our best model is the one that maximize $F_{total}$ over all possible partitions.

From a Bayesian point of view the fitness function corresponds to the **likelihood**

$$P(D|M)$$

## The fitness function - Cash Statistics

With a model $M(t,\theta)$, the unbinned log-likelihood reads:

$$\log L(\theta) = \sum_n \log M(t_n,\theta) - \int M(t,\theta)dt$$

our model is constant: $M(t,\lambda) = \lambda$ so for block *k*

$$\log L^{(k)}(\lambda) = N^{(k)} \log \lambda - \lambda T^{(k)}$$

now we maximize wrt the nuisance parameter $\lambda$ (height of the block):

$$\log L^{(k)}_{max} = N^{(k)}(\log N^{(k)} - \log T^{(k)}) + N^{(k)}$$

## The fitness function - Cash Statistics

With a model $M(t,\theta)$, the unbinned log-likelihood reads:

$$\log L(\theta) = \sum_n \log M(t_n, \theta) - \int M(t, \theta) dt$$

our model is constant: $M(t,\lambda) = \lambda$ so for block *k*

$$\log L^{(k)}(\lambda) = N^{(k)} \log \lambda - \lambda T^{(k)}$$

now we maximize wrt the nuisance parameter $\lambda$ (height of the block):

independent of the partition

$$\begin{aligned} \log L^{(k)}_{max} &= N^{(k)}(\log N^{(k)} - \log T^{(k)}) + \cancel{N^{(k)}} \\ &= N^{(k)}(\log N^{(k)} - \log T^{(k)}) \end{aligned}$$

# The fitness function - Cash Statistics

$$\log L^{(k)}_{max} = N^{(k)}(\log N^{(k)} - \log T^{(k)})$$

where $N^{(k)}$ number of events and $T^{(k)}$ is the length of the Kth block.

This fitness function has two nice features:

- is simple
- is scale invariant ($T \rightarrow \alpha T$)

The fitness of the entire partition will be then:

$$\log L = \sum_k \log L^{(k)}_{max}$$

## The Prior on the number of blocks

We are interested in a representation where $N_{blocks} \ll N$ rather than $N_{blocks} \approx N$

- Uniform prior looks unreasonable

We selected two candidate priors for our algorithm that assign smaller probability on a large number of blocks:

- Geometric Prior
- Calibrated Prior

## Geometric Prior

$$P(N_{blocks}) = \begin{cases} P_0 \gamma^{N_{blocks}} & 0 \le N_{blocks} \le N \\ 0 & \text{otherwise} \end{cases}$$

$$P_0 = \frac{1-\gamma}{1-\gamma^{N+1}}$$

- $\gamma$ is a free parameter
- we need to tune this parameter…

## Geometric Prior

$$P(N_{blocks}) = \begin{cases} P_0 \gamma^{N_{blocks}} & 0 \leq N_{blocks} \leq N \\ 0 & \text{otherwise} \end{cases}$$

$$P_0 = \frac{1-\gamma}{1-\gamma^{N+1}}$$

- $\gamma$ is a free parameter
- we need to tune this parameter…

The prior can be interpreted as a control on the false-positive rate for detecting change-points.

$\downarrow$

**fixing the rate of Type-I errors**

## Geometric Prior

$$P(N_{blocks}) = \begin{cases} P_0 \gamma^{N_{blocks}} & 0 \leq N_{blocks} \leq N \\ 0 & \text{otherwise} \end{cases}$$

$$P_0 = \frac{1-\gamma}{1-\gamma^{N+1}}$$

- $\gamma$ is a free parameter
- we need to tune this parameter…

The prior can be interpreted as a control on the false-positive rate for detecting change-points.

$\downarrow$

**fixing the rate of Type-I errors**

**Recipe**: running the algorithm with a few different values. In general, the number of change-points is insensitive to a large range of reasonable values of the “steepness” parameter.

## Calibrated Prior

Another approach is to calibrate the prior as a function of the number of data points $N$ and the false-positive rate $p_0$ on toy pure-noise experiments.

$$\log P(N, p_0) = \log(73.53 p_0 N^{-0.478}) - 4$$

$$p_0 = 1 - (\text{false positive probability})$$

$p_0$ represents the frequency with which the algorithm correctly rejects the presence of a change-point in pure noise.

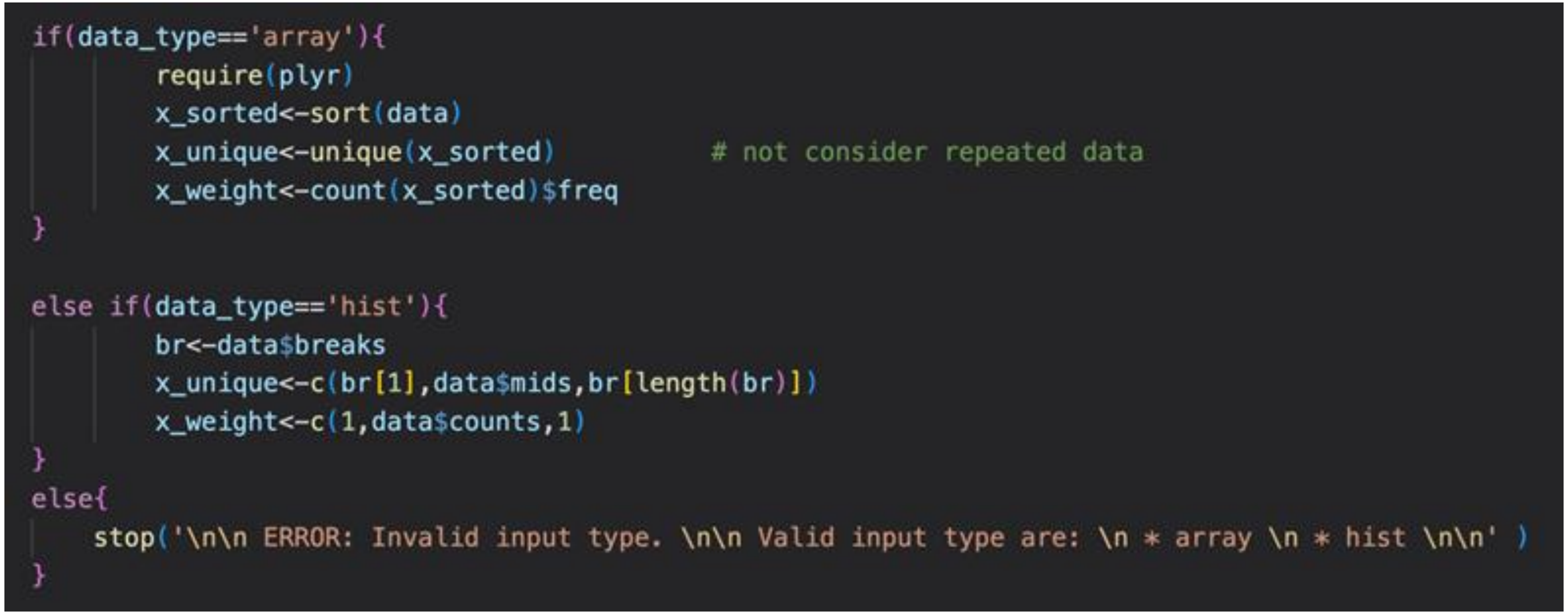
## Algorithm structure

The algorithm takes in several parameters:

- `data`: the input data to be partitioned
- `data_type`: the type of input data (`array` or `histogram`)
- `prior`: the prior distribution to use (`geometric` or `calibrated`)
- `gamma`: the slope of the geometric prior
- `p0`: the correct detection rate parameter for the calibrated prior
- `verbose`: a boolean parameter that if set `True` prints useful informations on the algorithm

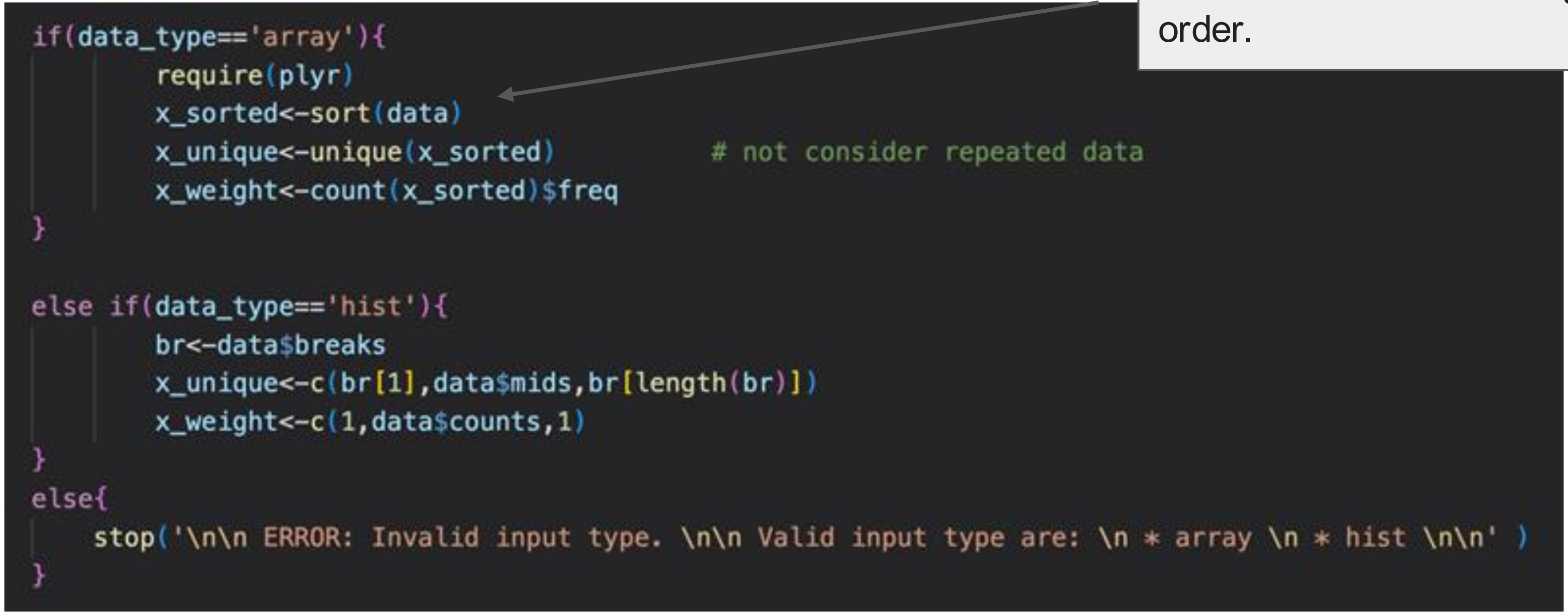
The algorithm starts by sorting and counting the data, then it creates a **set of possible bin edges**. The function then **evaluates the fitness function for each possible partition**, taking into account a prior distribution on the number of blocks. The algorithm then traces back through the previous partitions to find the **optimal partition that maximizes the fitness function**. The change points are then returned as the edges of the blocks.

## Algorithm structure



```
if(data_type=='array'){
        require(plyr)
        x_sorted<-sort(data)
        x_unique<-unique(x_sorted)          # not consider repeated data
        x_weight<-count(x_sorted)$freq
}

else if(data_type=='hist'){
        br<-data$breaks
        x_unique<-c(br[1],data$mids,br[length(br)])
        x_weight<-c(1,data$counts,1)
}
else{
    stop('\n\n ERROR: Invalid input type. \n\n Valid input type are: \n * array \n * hist \n\n' )
}
```

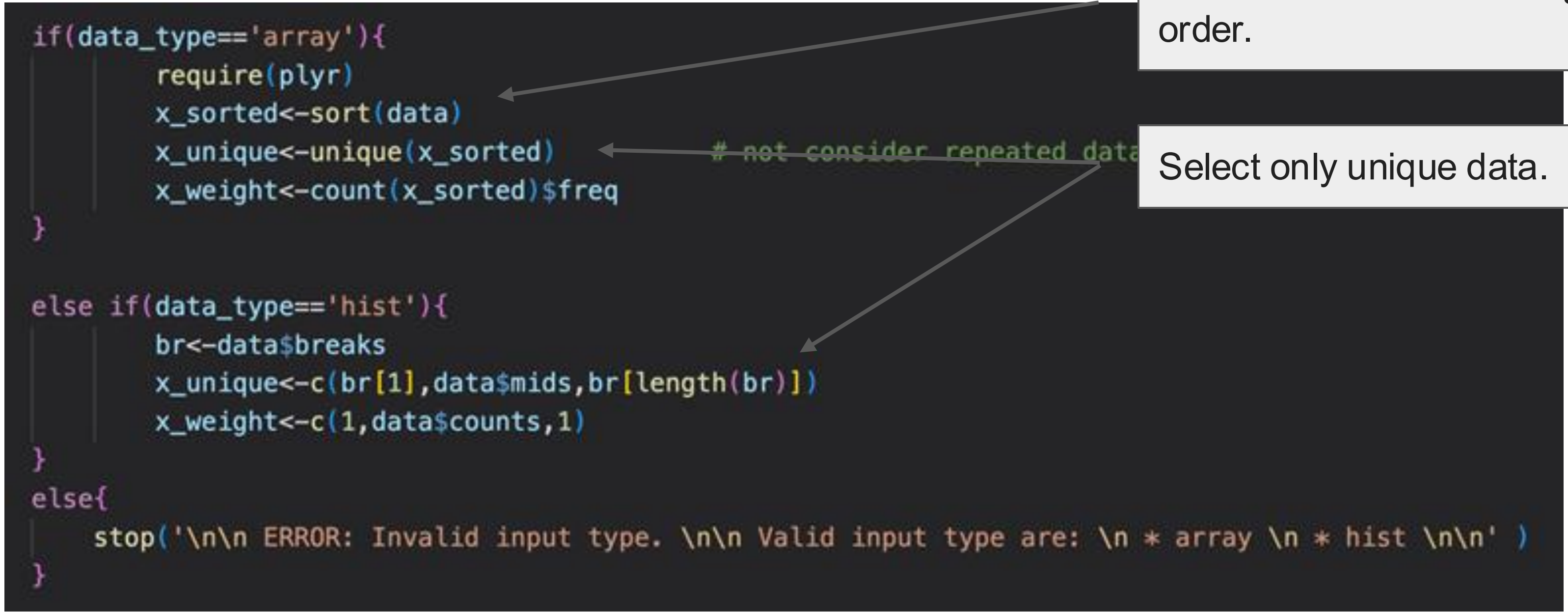
## Algorithm structure



```
if(data_type=='array'){
        require(plyr)
        x_sorted<-sort(data)
        x_unique<-unique(x_sorted)          # not consider repeated data
        x_weight<-count(x_sorted)$freq
}

else if(data_type=='hist'){
        br<-data$breaks
        x_unique<-c(br[1],data$mids,br[length(br)])
        x_weight<-c(1,data$counts,1)
}
else{
    stop('\n\n ERROR: Invalid input type. \n\n Valid input type are: \n * array \n * hist \n\n' )
}
```

Sort the data in ascending order.

## Algorithm structure

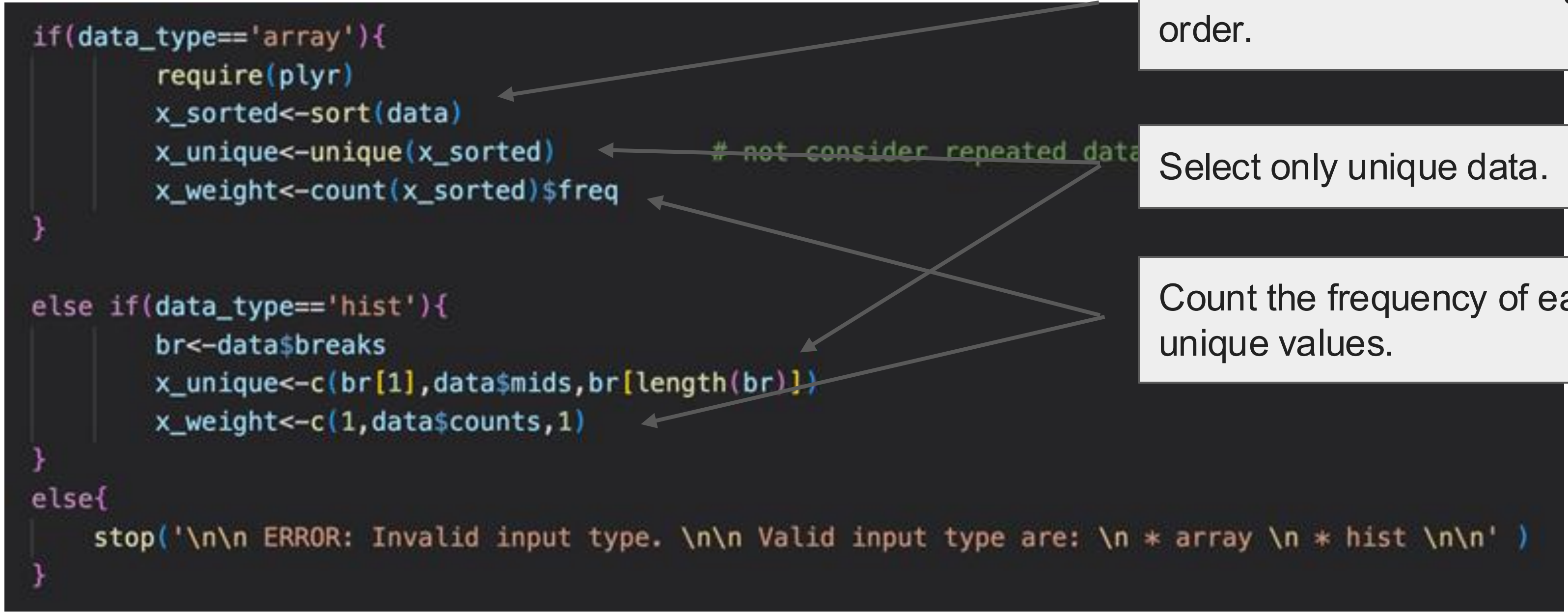


```
if(data_type=='array'){
        require(plyr)
        x_sorted<-sort(data)
        x_unique<-unique(x_sorted)            # not consider repeated data
        x_weight<-count(x_sorted)$freq
}

else if(data_type=='hist'){
        br<-data$breaks
        x_unique<-c(br[1],data$mids,br[length(br)])
        x_weight<-c(1,data$counts,1)
}
else{
    stop('\n\n ERROR: Invalid input type. \n\n Valid input type are: \n * array \n * hist \n\n' )
}
```

Sort the data in ascending order.

Select only unique data.

# Algorithm structure



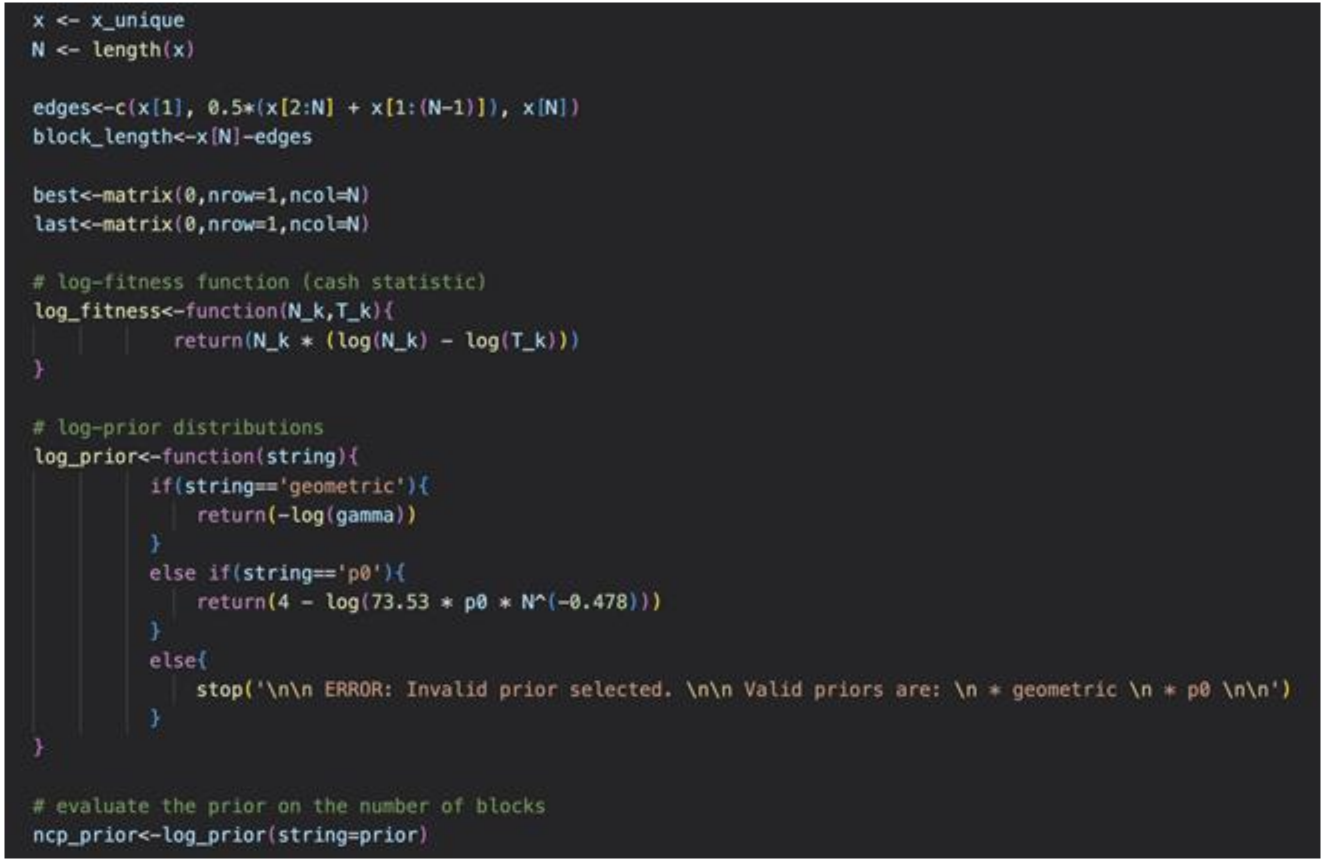
```
if(data_type=='array'){
        require(plyr)
        x_sorted<-sort(data)
        x_unique<-unique(x_sorted)          # not consider repeated data
        x_weight<-count(x_sorted)$freq
}

else if(data_type=='hist'){
        br<-data$breaks
        x_unique<-c(br[1],data$mids,br[length(br)])
        x_weight<-c(1,data$counts,1)
}
else{
    stop('\n\n ERROR: Invalid input type. \n\n Valid input type are: \n * array \n * hist \n\n' )
}
```

Sort the data in ascending order.

Select only unique data.

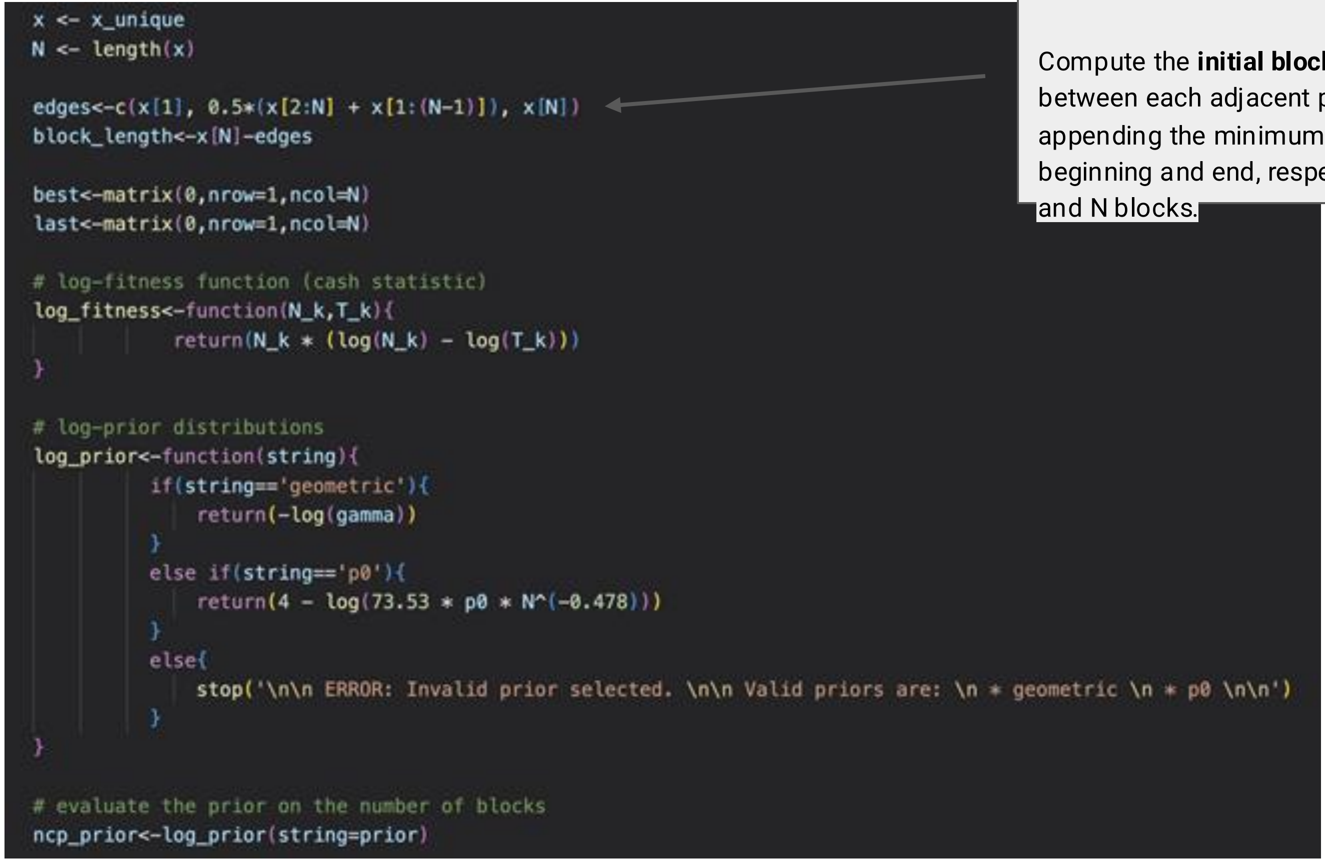
Count the frequency of each unique values.

# Algorithm structure



```r
x <- x_unique
N <- length(x)

edges<-c(x[1], 0.5*(x[2:N] + x[1:(N-1)]), x[N])
block_length<-x[N]-edges

best<-matrix(0,nrow=1,ncol=N)
last<-matrix(0,nrow=1,ncol=N)

# log-fitness function (cash statistic)
log_fitness<-function(N_k,T_k){
            return(N_k * (log(N_k) - log(T_k)))
}

# log-prior distributions
log_prior<-function(string){
          if(string=='geometric'){
              return(-log(gamma))
          }
          else if(string=='p0'){
              return(4 - log(73.53 * p0 * N^(-0.478)))
          }
          else{
              stop('\n\n ERROR: Invalid prior selected. \n\n Valid priors are: \n * geometric \n * p0 \n\n')
          }
}

# evaluate the prior on the number of blocks
ncp_prior<-log_prior(string=prior)
```

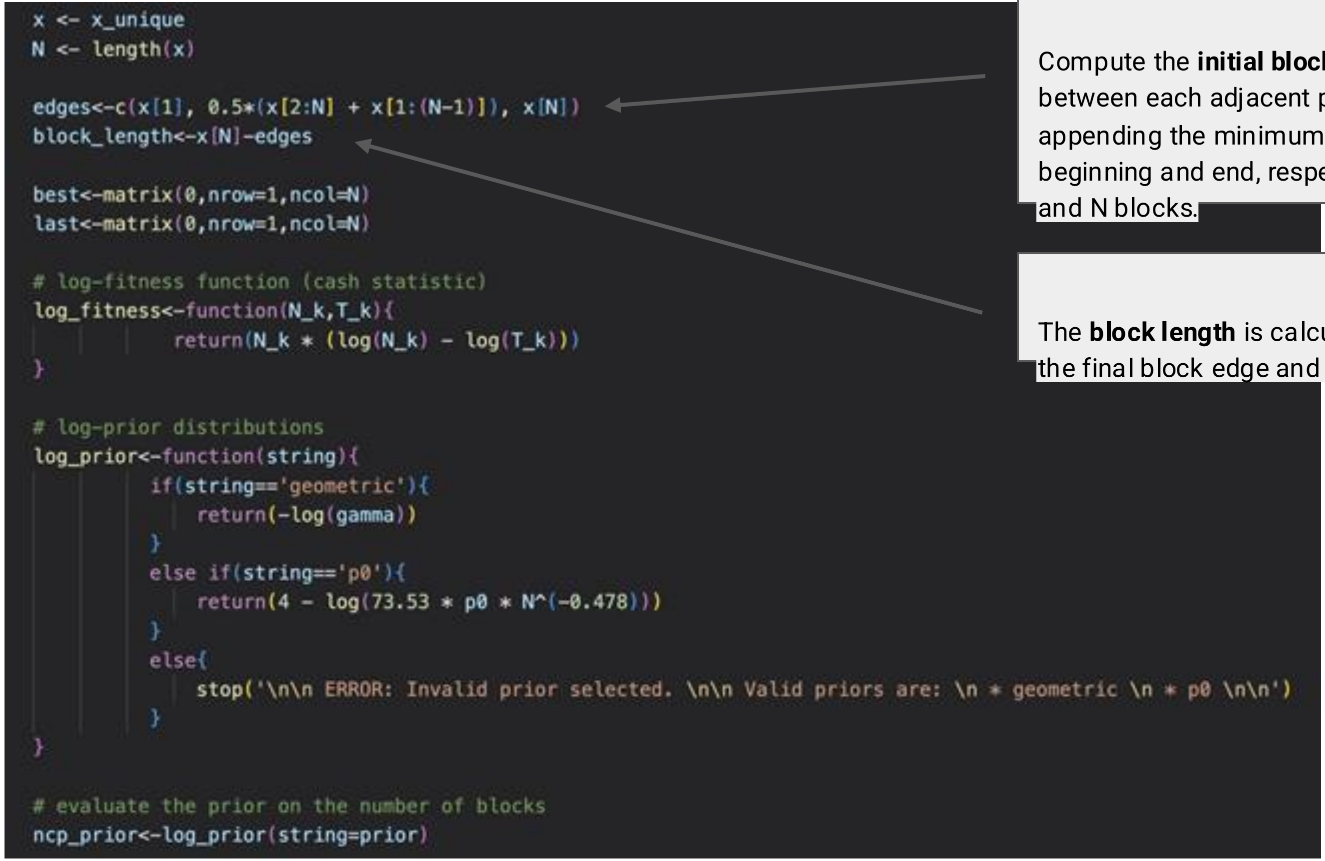
# Algorithm structure



```
x <- x_unique
N <- length(x)

edges<-c(x[1], 0.5*(x[2:N] + x[1:(N-1)]), x[N])
block_length<-x[N]-edges

best<-matrix(0,nrow=1,ncol=N)
last<-matrix(0,nrow=1,ncol=N)

# log-fitness function (cash statistic)
log_fitness<-function(N_k,T_k){
            return(N_k * (log(N_k) - log(T_k)))
}

# log-prior distributions
log_prior<-function(string){
          if(string=='geometric'){
              return(-log(gamma))
          }
          else if(string=='p0'){
              return(4 - log(73.53 * p0 * N^(-0.478)))
          }
          else{
              stop('\n\n ERROR: Invalid prior selected. \n\n Valid priors are: \n * geometric \n * p0 \n\n')
          }
}

# evaluate the prior on the number of blocks
ncp_prior<-log_prior(string=prior)
```

Compute the **initial block edges** by taking the midpoint between each adjacent pair of unique values in x, and appending the minimum and maximum values of x to the beginning and end, respectively. This results in N+1 edges and N blocks.

# Algorithm structure

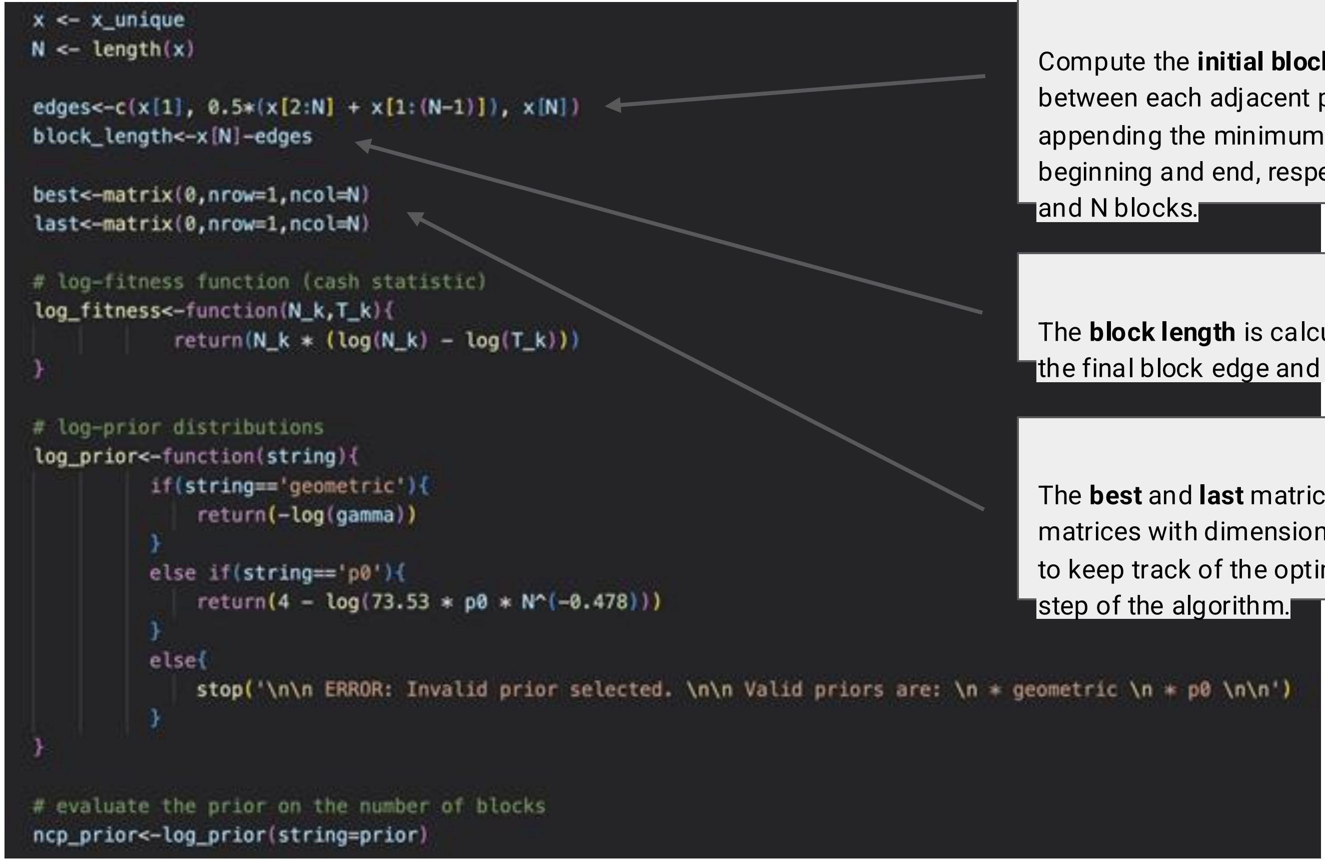


```r
x <- x_unique
N <- length(x)

edges<-c(x[1], 0.5*(x[2:N] + x[1:(N-1)]), x[N])
block_length<-x[N]-edges

best<-matrix(0,nrow=1,ncol=N)
last<-matrix(0,nrow=1,ncol=N)

# log-fitness function (cash statistic)
log_fitness<-function(N_k,T_k){
      return(N_k * (log(N_k) - log(T_k)))
}

# log-prior distributions
log_prior<-function(string){
    if(string=='geometric'){
      return(-log(gamma))
    }
    else if(string=='p0'){
      return(4 - log(73.53 * p0 * N^(-0.478)))
    }
    else{
      stop('\n\n ERROR: Invalid prior selected. \n\n Valid priors are: \n * geometric \n * p0 \n\n')
    }
}

# evaluate the prior on the number of blocks
ncp_prior<-log_prior(string=prior)
```

Compute the **initial block edges** by taking the midpoint between each adjacent pair of unique values in x, and appending the minimum and maximum values of x to the beginning and end, respectively. This results in N+1 edges and N blocks.

The **block length** is calculated as the difference between the final block edge and each of the block edges.

# Algorithm structure



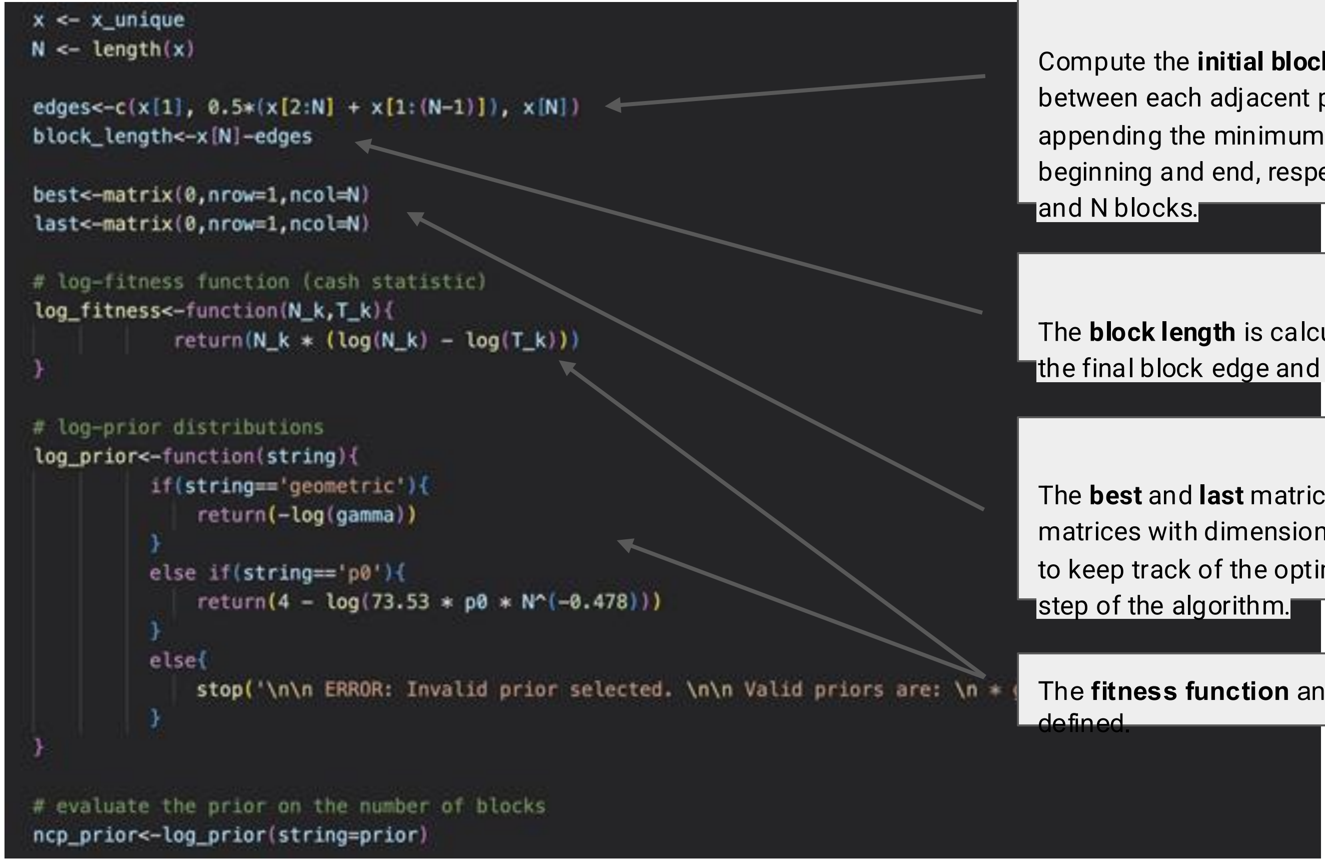
```
x <- x_unique
N <- length(x)

edges<-c(x[1], 0.5*(x[2:N] + x[1:(N-1)]), x[N])
block_length<-x[N]-edges

best<-matrix(0,nrow=1,ncol=N)
last<-matrix(0,nrow=1,ncol=N)

# log-fitness function (cash statistic)
log_fitness<-function(N_k,T_k){
            return(N_k * (log(N_k) - log(T_k)))
}

# log-prior distributions
log_prior<-function(string){
          if(string=='geometric'){
              return(-log(gamma))
          }
          else if(string=='p0'){
              return(4 - log(73.53 * p0 * N^(-0.478)))
          }
          else{
              stop('\n\n ERROR: Invalid prior selected. \n\n Valid priors are: \n * geometric \n * p0 \n\n')
          }
}

# evaluate the prior on the number of blocks
ncp_prior<-log_prior(string=prior)
```

Compute the **initial block edges** by taking the midpoint between each adjacent pair of unique values in x, and appending the minimum and maximum values of x to the beginning and end, respectively. This results in N+1 edges and N blocks.

The **block length** is calculated as the difference between the final block edge and each of the block edges.

The **best** and **last** matrices are initialized to be zero matrices with dimensions (1, N). These matrices are used to keep track of the optimal partition of the data at each step of the algorithm.

# Algorithm structure



```r
x <- x_unique
N <- length(x)

edges<-c(x[1], 0.5*(x[2:N] + x[1:(N-1)]), x[N])
block_length<-x[N]-edges

best<-matrix(0,nrow=1,ncol=N)
last<-matrix(0,nrow=1,ncol=N)

# log-fitness function (cash statistic)
log_fitness<-function(N_k,T_k){
            return(N_k * (log(N_k) - log(T_k)))
}

# log-prior distributions
log_prior<-function(string){
          if(string=='geometric'){
            return(-log(gamma))
          }
          else if(string=='p0'){
            return(4 - log(73.53 * p0 * N^(-0.478)))
          }
          else{
            stop('\n\n ERROR: Invalid prior selected. \n\n Valid priors are: \n *
          }
}

# evaluate the prior on the number of blocks
ncp_prior<-log_prior(string=prior)
```

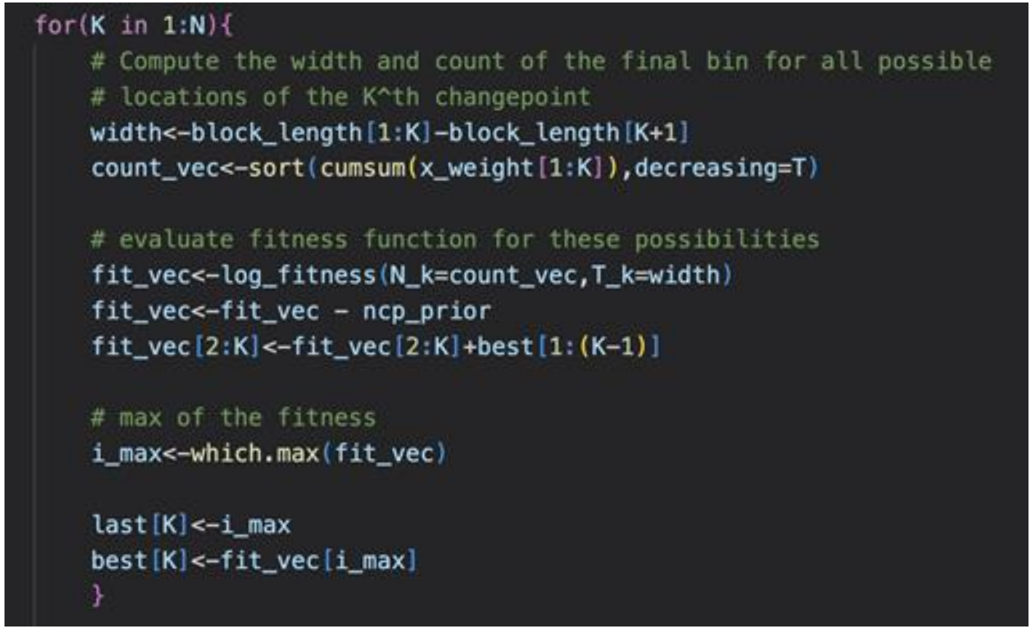
Compute the **initial block edges** by taking the midpoint between each adjacent pair of unique values in x, and appending the minimum and maximum values of x to the beginning and end, respectively. This results in N+1 edges and N blocks.

The **block length** is calculated as the difference between the final block edge and each of the block edges.

The **best** and **last** matrices are initialized to be zero matrices with dimensions (1, N). These matrices are used to keep track of the optimal partition of the data at each step of the algorithm.

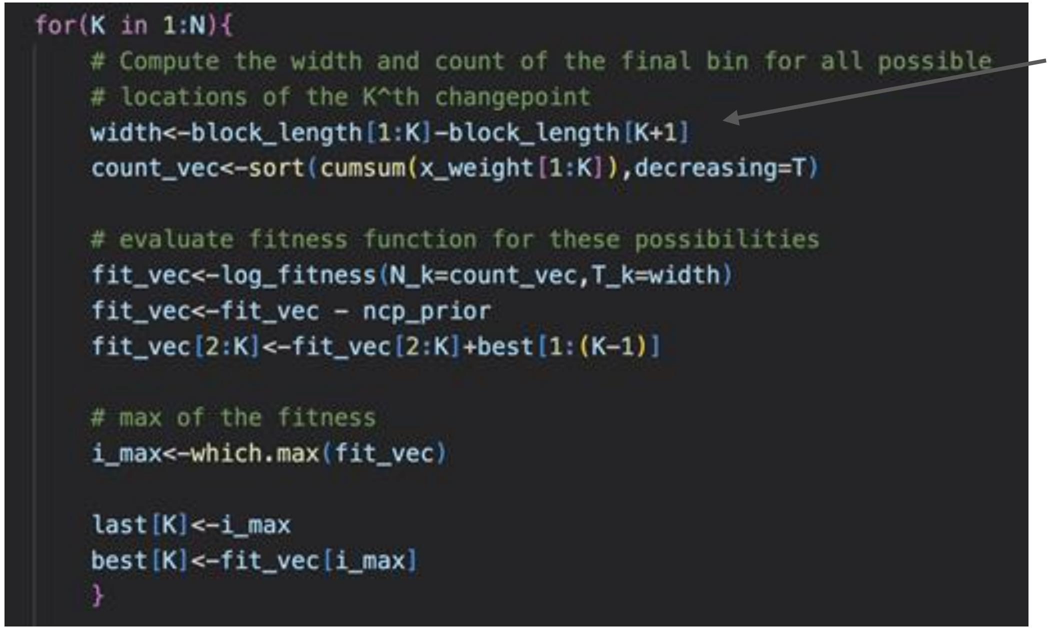
The **fitness function** and the **priors functions** are defined.

## Algorithm structure



```
for(K in 1:N){
    # Compute the width and count of the final bin for all possible
    # locations of the K^th changepoint
    width<-block_length[1:K]-block_length[K+1]
    count_vec<-sort(cumsum(x_weight[1:K]),decreasing=T)

    # evaluate fitness function for these possibilities
    fit_vec<-log_fitness(N_k=count_vec,T_k=width)
    fit_vec<-fit_vec - ncp_prior
    fit_vec[2:K]<-fit_vec[2:K]+best[1:(K-1)]

    # max of the fitness
    i_max<-which.max(fit_vec)

    last[K]<-i_max
    best[K]<-fit_vec[i_max]
    }
```

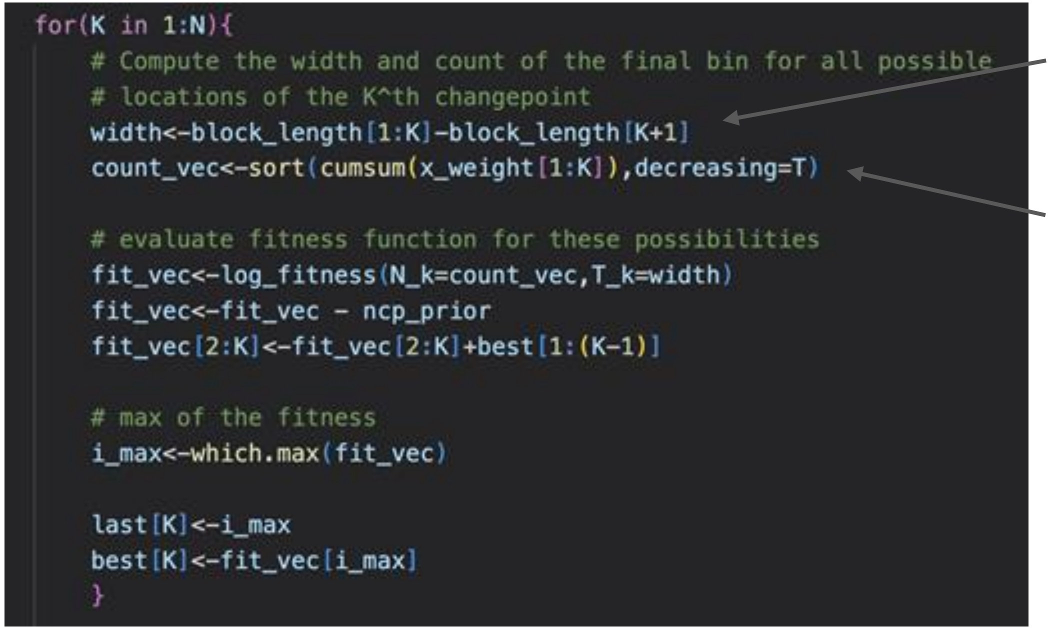
# Algorithm structure



```
for(K in 1:N){
    # Compute the width and count of the final bin for all possible
    # locations of the K^th changepoint
    width<-block_length[1:K]-block_length[K+1]
    count_vec<-sort(cumsum(x_weight[1:K]),decreasing=T)

    # evaluate fitness function for these possibilities
    fit_vec<-log_fitness(N_k=count_vec,T_k=width)
    fit_vec<-fit_vec - ncp_prior
    fit_vec[2:K]<-fit_vec[2:K]+best[1:(K-1)]

    # max of the fitness
    i_max<-which.max(fit_vec)

    last[K]<-i_max
    best[K]<-fit_vec[i_max]
    }
```

Evaluate the **width** of each block

## Algorithm structure

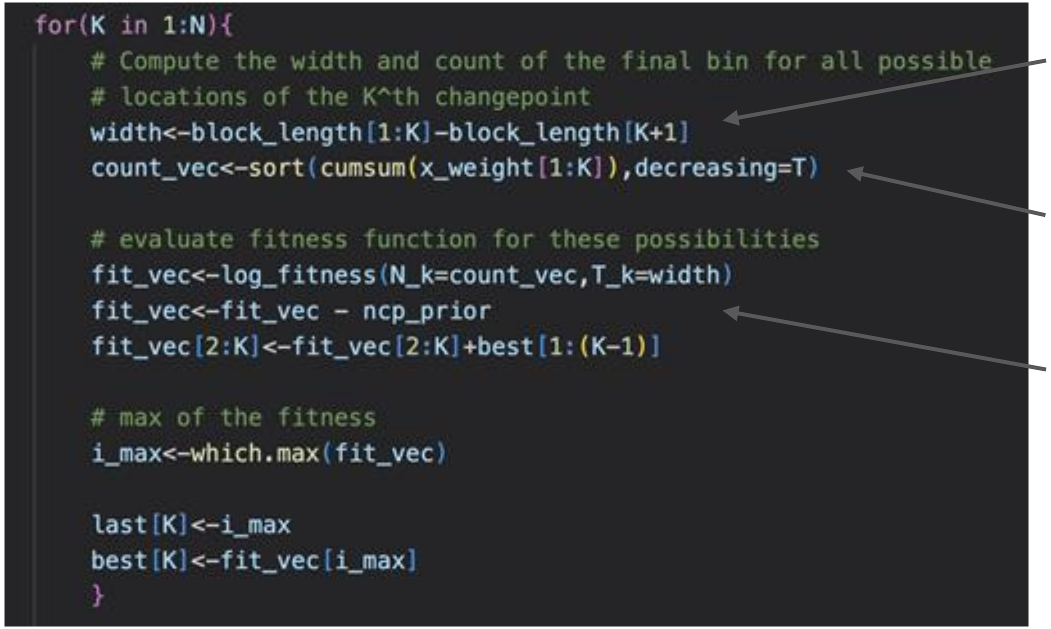


```
for(K in 1:N){
    # Compute the width and count of the final bin for all possible
    # locations of the K^th changepoint
    width<-block_length[1:K]-block_length[K+1]
    count_vec<-sort(cumsum(x_weight[1:K]),decreasing=T)

    # evaluate fitness function for these possibilities
    fit_vec<-log_fitness(N_k=count_vec,T_k=width)
    fit_vec<-fit_vec - ncp_prior
    fit_vec[2:K]<-fit_vec[2:K]+best[1:(K-1)]

    # max of the fitness
    i_max<-which.max(fit_vec)

    last[K]<-i_max
    best[K]<-fit_vec[i_max]
    }
```

Evaluate the **width** of each block

Compute the **cumulative sum of the weights** of the unique values up to the Kth index, and then sorts it in descending order.

# Algorithm structure



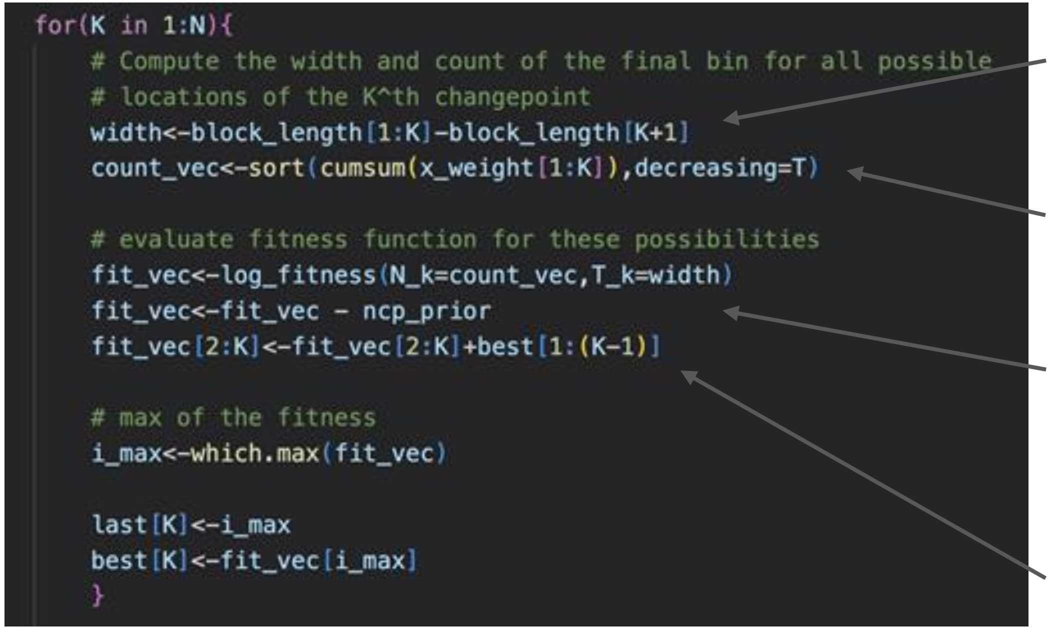
```
for(K in 1:N){
    # Compute the width and count of the final bin for all possible
    # locations of the K^th changepoint
    width<-block_length[1:K]-block_length[K+1]
    count_vec<-sort(cumsum(x_weight[1:K]),decreasing=T)

    # evaluate fitness function for these possibilities
    fit_vec<-log_fitness(N_k=count_vec,T_k=width)
    fit_vec<-fit_vec - ncp_prior
    fit_vec[2:K]<-fit_vec[2:K]+best[1:(K-1)]

    # max of the fitness
    i_max<-which.max(fit_vec)

    last[K]<-i_max
    best[K]<-fit_vec[i_max]
    }
```

Evaluate the **width** of each block

Compute the **cumulative sum of the weights** of the unique values up to the Kth index, and then sorts it in descending order.

Compute the **fitness.**

# Algorithm structure



```
for(K in 1:N){
    # Compute the width and count of the final bin for all possible
    # locations of the K^th changepoint
    width<-block_length[1:K]-block_length[K+1]
    count_vec<-sort(cumsum(x_weight[1:K]),decreasing=T)

    # evaluate fitness function for these possibilities
    fit_vec<-log_fitness(N_k=count_vec,T_k=width)
    fit_vec<-fit_vec - ncp_prior
    fit_vec[2:K]<-fit_vec[2:K]+best[1:(K-1)]

    # max of the fitness
    i_max<-which.max(fit_vec)

    last[K]<-i_max
    best[K]<-fit_vec[i_max]
    }
```

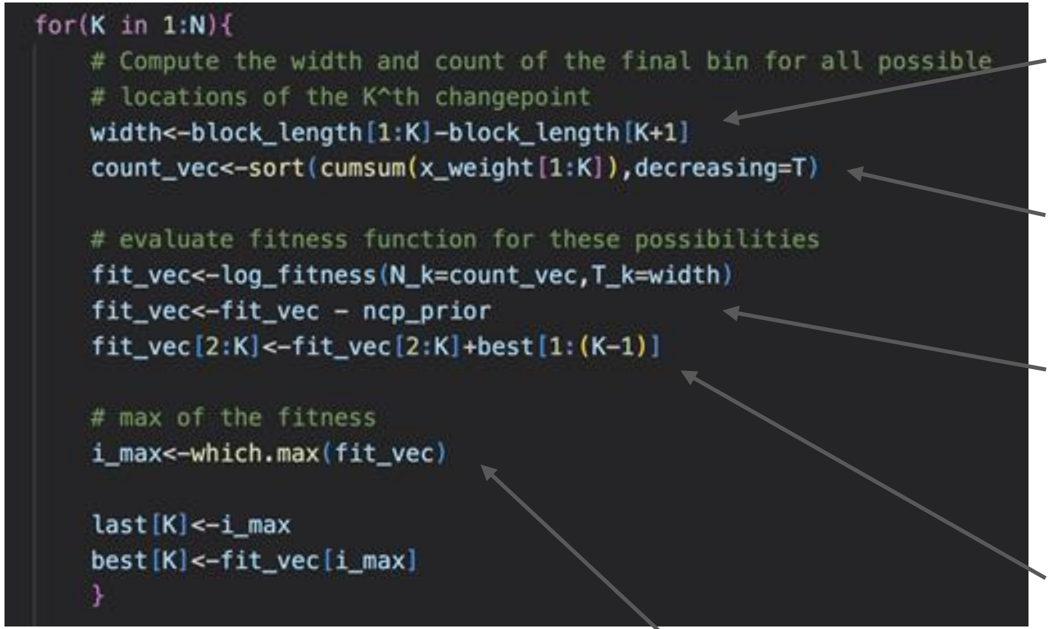
Evaluate the **width** of each block

Compute the **cumulative sum of the weights** of the unique values up to the Kth index, and then sorts it in descending order.

Compute the **fitness.**

**Update the fitness** values of all but the first possible changepoint locations by adding the best fitness values of all previous possible change-point locations.

# Algorithm structure



```
for(K in 1:N){
    # Compute the width and count of the final bin for all possible
    # locations of the K^th changepoint
    width<-block_length[1:K]-block_length[K+1]
    count_vec<-sort(cumsum(x_weight[1:K]),decreasing=T)

    # evaluate fitness function for these possibilities
    fit_vec<-log_fitness(N_k=count_vec,T_k=width)
    fit_vec<-fit_vec - ncp_prior
    fit_vec[2:K]<-fit_vec[2:K]+best[1:(K-1)]

    # max of the fitness
    i_max<-which.max(fit_vec)

    last[K]<-i_max
    best[K]<-fit_vec[i_max]
    }
```

Evaluate the **width** of each block

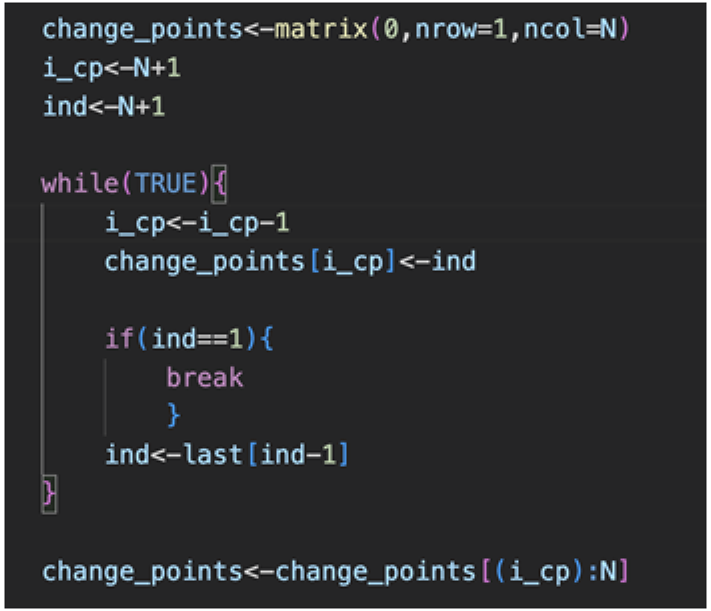
Compute the **cumulative sum of the weights** of the unique values up to the Kth index, and then sorts it in descending order.

Compute the **fitness.**

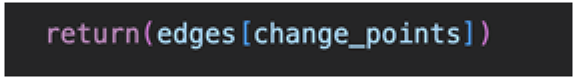
**Update the fitness** values of all but the first possible changepoint locations by adding the best fitness values of all previous possible change-point locations.

Find the location of the maximum value in the fit_vec vector, which corresponds to the **location of the Kth change-point** that maximizes the log-likelihood of the data.

## Algorithm structure

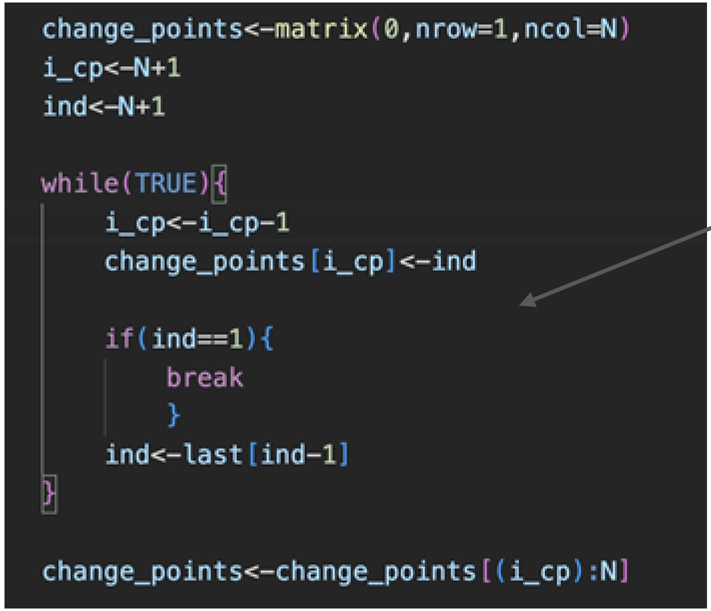


```
change_points<-matrix(0,nrow=1,ncol=N)
i_cp<-N+1
ind<-N+1

while(TRUE){
    i_cp<-i_cp-1
    change_points[i_cp]<-ind

    if(ind==1){
        break
        }
    ind<-last[ind-1]
}

change_points<-change_points[(i_cp):N]
```

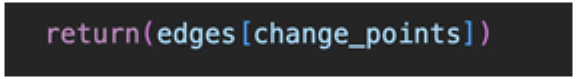


```
return(edges[change_points])
```

## Algorithm structure



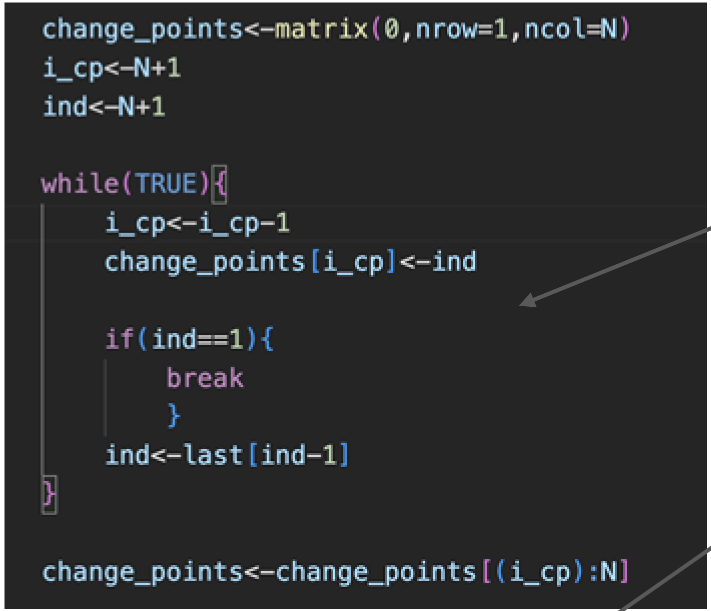
```
change_points<-matrix(0,nrow=1,ncol=N)
i_cp<-N+1
ind<-N+1

while(TRUE){
    i_cp<-i_cp-1
    change_points[i_cp]<-ind

    if(ind==1){
        break
        }
    ind<-last[ind-1]
}

change_points<-change_points[(i_cp):N]
```



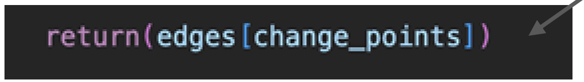
```
return(edges[change_points])
```

**Determining the locations of the changepoints** that form the optimal partitioning of the input data. The while loop starts from the last block and iteratively backtracks through the changepoints to identify their location, until it reaches the first block.

# Algorithm structure



```
change_points<-matrix(0,nrow=1,ncol=N)
i_cp<-N+1
ind<-N+1

while(TRUE){
    i_cp<-i_cp-1
    change_points[i_cp]<-ind

    if(ind==1){
        break
        }
    ind<-last[ind-1]
}

change_points<-change_points[(i_cp):N]
```



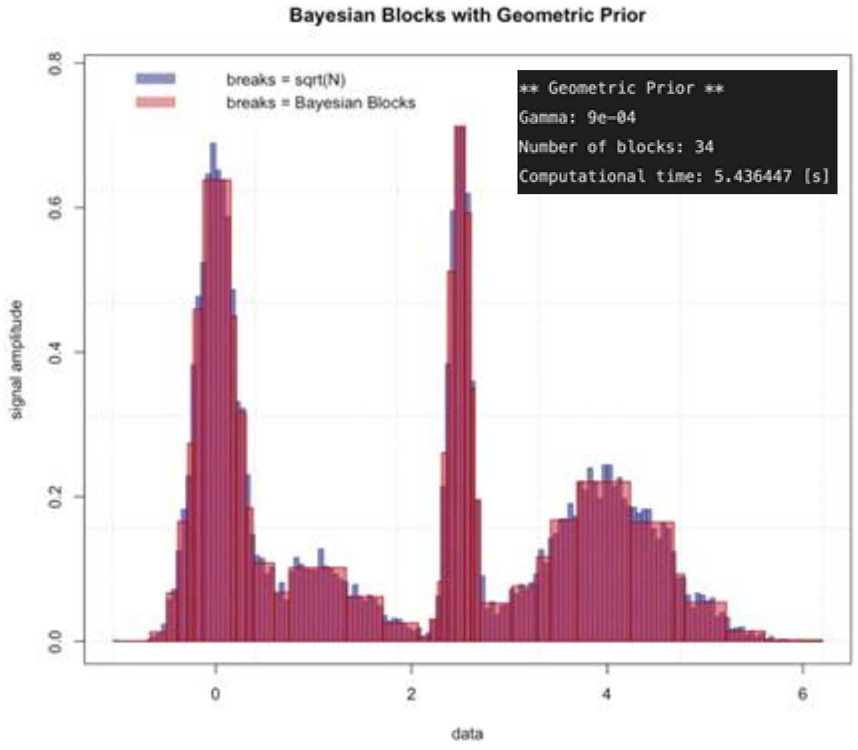
```
return(edges[change_points])
```

**Determining the locations of the changepoints** that form the optimal partitioning of the input data. The while loop starts from the last block and iteratively backtracks through the changepoints to identify their location, until it reaches the first block.

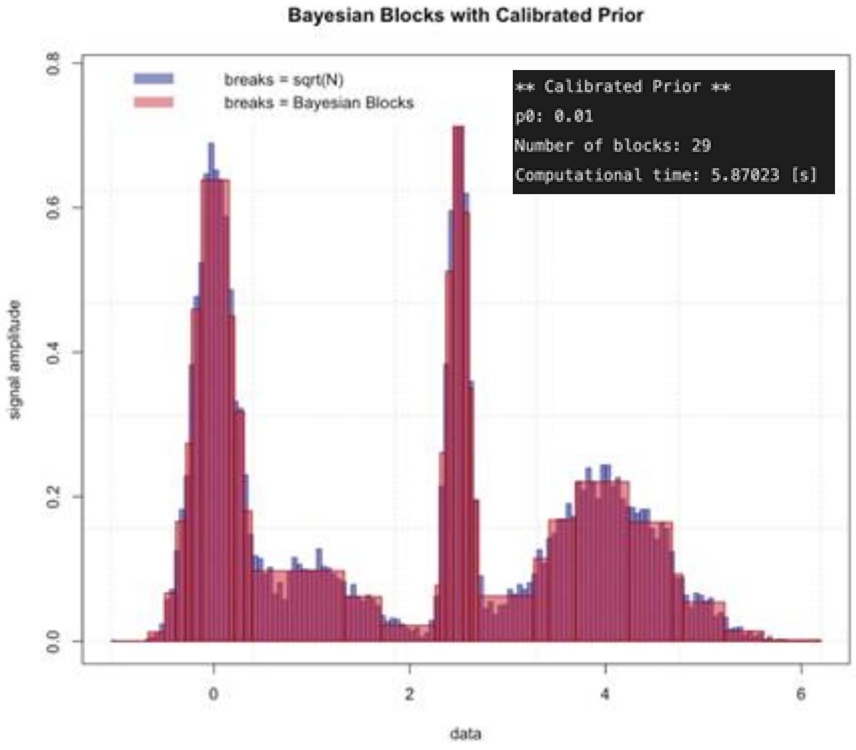
Return the vector of the **estimated bin edges**.

# Applications: Binning of histograms

1500 random toy data generated from a mixture of four normal distributions (122 bins)



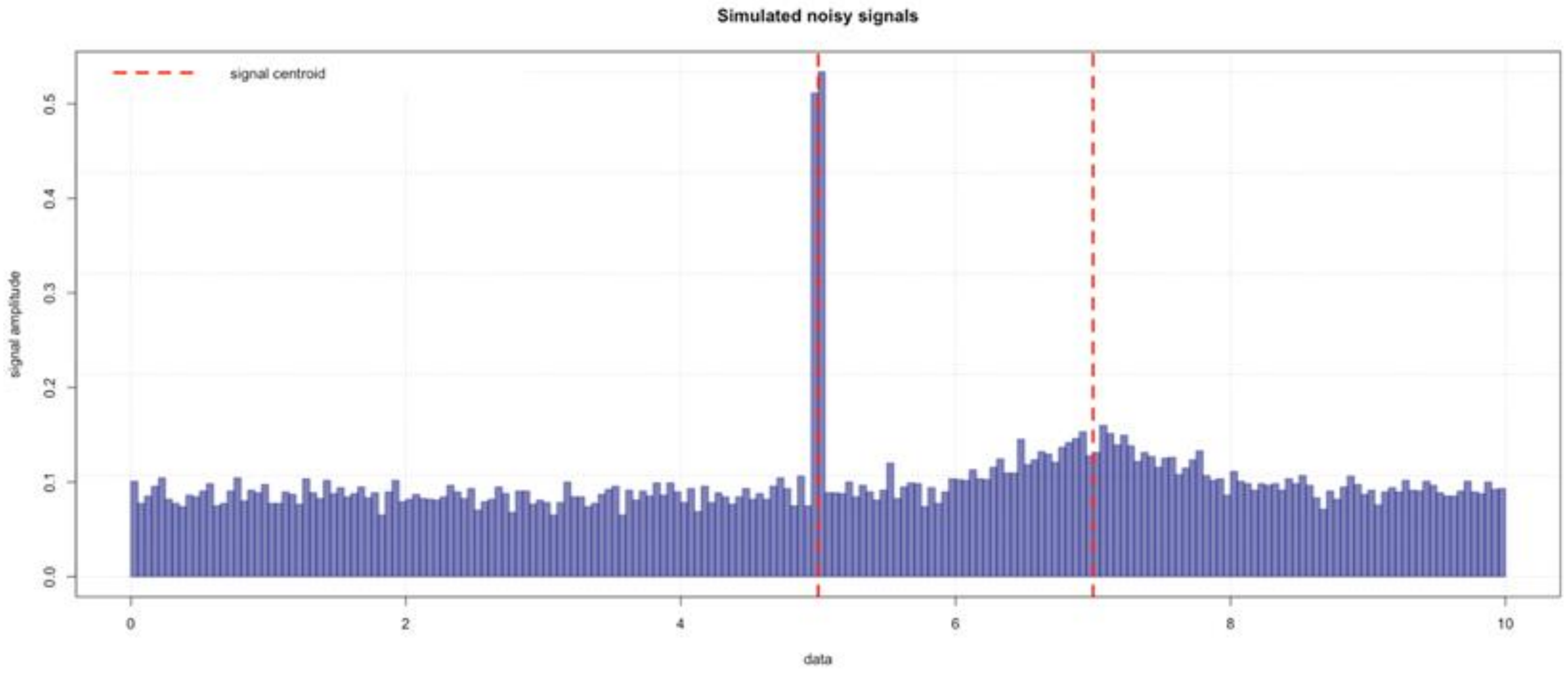
```
** Geometric Prior **
Gamma: 9e-04
Number of blocks: 34
Computational time: 5.436447 [s]
```



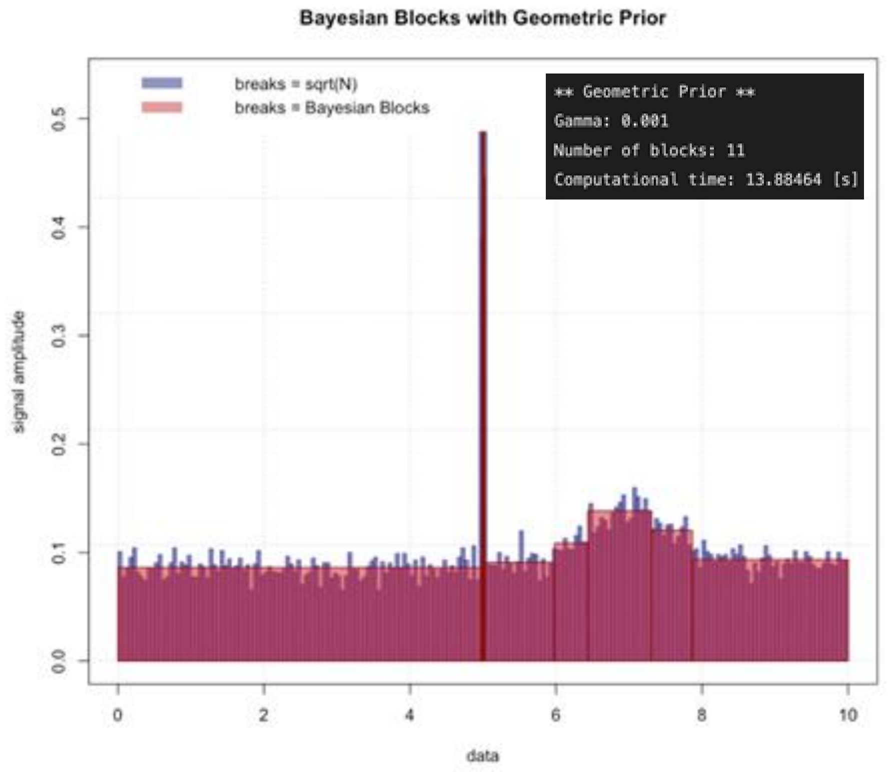
```
** Calibrated Prior **
p0: 0.01
Number of blocks: 29
Computational time: 5.87023 [s]
```

# Applications: Signal detection

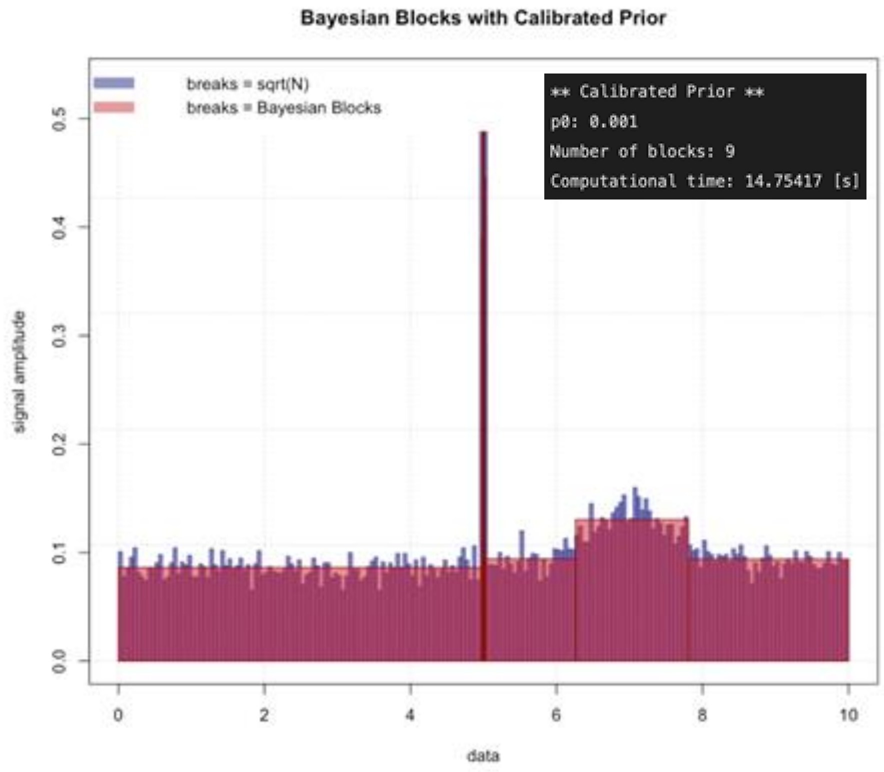
Uniform background noise with two gaussian signals (**N** (5,0.01), **N** (7,0.6))



# Applications: Signal detection



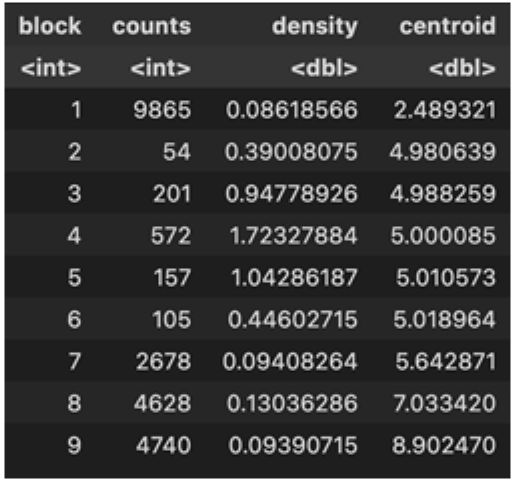
```
** Geometric Prior **
Gamma: 0.001
Number of blocks: 11
Computational time: 13.88464 [s]
```



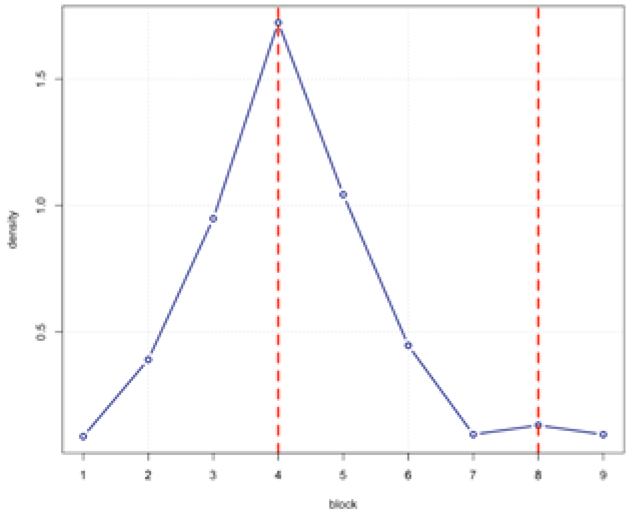
```
** Calibrated Prior **
p0: 0.001
Number of blocks: 9
Computational time: 14.75417 [s]
```

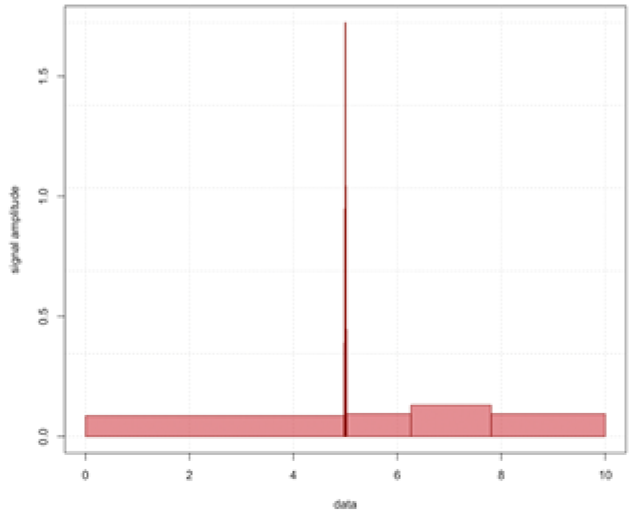
# Applications: Signal detection

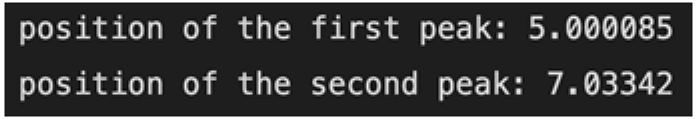
Information on single block can be easily inspected using the usual R’s syntax for histograms.



| block | counts | density | centroid |
|---|---|---|---|
| <int> | <int> | <dbl> | <dbl> |
| 1 | 9865 | 0.08618566 | 2.489321 |
| 2 | 54 | 0.39008075 | 4.980639 |
| 3 | 201 | 0.94778926 | 4.988259 |
| 4 | 572 | 1.72327884 | 5.000085 |
| 5 | 157 | 1.04286187 | 5.010573 |
| 6 | 105 | 0.44602715 | 5.018964 |
| 7 | 2678 | 0.09408264 | 5.642871 |
| 8 | 4628 | 0.13036286 | 7.033420 |
| 9 | 4740 | 0.09390715 | 8.902470 |







```
position of the first peak: 5.000085
position of the second peak: 7.03342
```

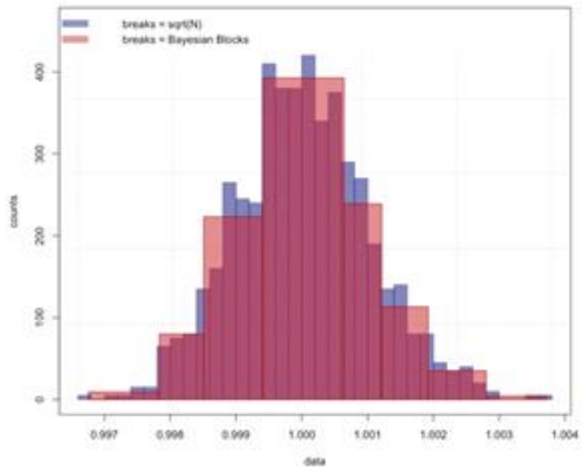
## Applications: Kernel density estimator for SBAM

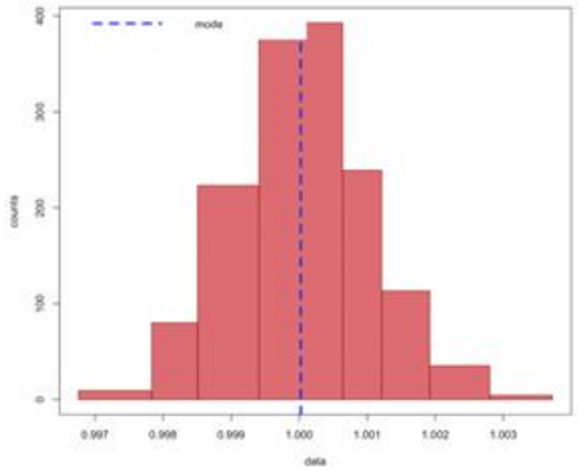
**PySINDy** is a sparse regression package with several implementations for the Sparse Identification of Nonlinear Dynamical systems (SINDy) [7].

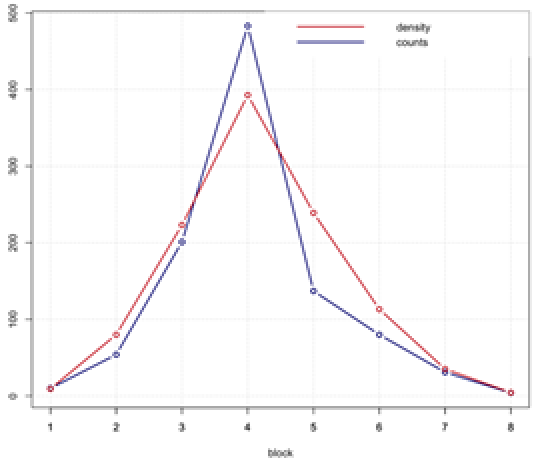
The **SBAM** (Sparse Bayesian Auxiliary Module ) [5] [6] module provides a bayesian implementation for the sparsity in the original PySindy package. Through the use of a Gibbs Sampler, the module provides a posterior probability distribution of the coefficients of the candidate functions (features) that will be used to reconstruct the dynamics of the system.

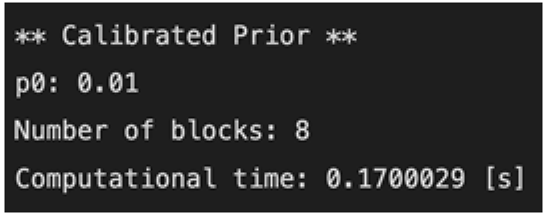
In the SBAM module, the **mean** was used as the best estimator for such posterior. With the Bayesian Blocks algorithm, **we can implement a kernel density estimator**; in this way, the **mode** can be easily calculated and used as the best estimator for the posterior distributions.

# Applications: Kernel density estimator for SBAM

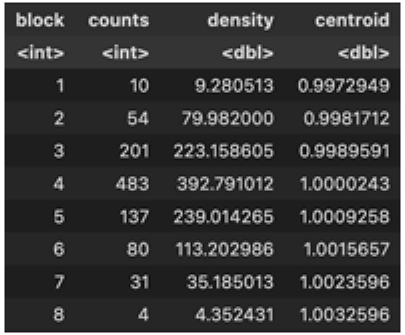




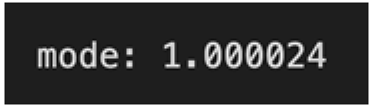




```
** Calibrated Prior **
p0: 0.01
Number of blocks: 8
Computational time: 0.1700029 [s]
```



| block | counts | density | centroid |
|---|---|---|---|
| <int> | <int> | <dbl> | <dbl> |
| 1 | 10 | 9.280513 | 0.9972949 |
| 2 | 54 | 79.982000 | 0.9981712 |
| 3 | 201 | 223.158605 | 0.9989591 |
| 4 | 483 | 392.791012 | 1.0000243 |
| 5 | 137 | 239.014265 | 1.0009258 |
| 6 | 80 | 113.202986 | 1.0015657 |
| 7 | 31 | 35.185013 | 1.0023596 |
| 8 | 4 | 4.352431 | 1.0032596 |



```
mode: 1.000024
```

## Conclusions

- The Bayesian Blocks algorithm provides an objective way for optimal data segmentation.
- The algorithm allows us to produce extremely understandable histograms that highlight the local background structure of the data by imposing as few preconditions as possible.
- The ease of code implementation and the possibility of customization by introducing new priors, make the Bayesian Block algorithm a powerful tool for studying time series, signal detection and as a kernel density estimator.

# References

[1] Studies in Astronomical Time Series Analysis. VI. Bayesian Block Representations, J. D. Scargle et al., Astrophys. J. 764 (2013) 167

[2] Bayesian Blocks in High Energy Physics: Better Binning made easy! B. Pollack et al. (2017), arXiv:1708.00810

[3] Studies in astronomical time series analysis: 5. Bayesian blocks, a new method to analyze structure in photon counting data, J. D. Scargle, Astrophys. J. 504 (1998) 405

[4] https://gist.github.com/gipert/4d245d97e8f7eebbe7cfc3eddbf27b7b (link to the original Julia code for the algorithm)

[5] https://docs.google.com/presentation/d/13ISUiQMHrTVek67IuxL47fNgRsP5JgQvKNSatCu-L6E/edit#slide=id.p (link to the SBAM presentation)

[6] https://github.com/zanocrate/LCPB_2219/tree/master/project (link to the GitHub repo of the SBAM module)

[7] https://pysindy.readthedocs.io/en/latest/# (link to the PySindy documentation)