# Package ‘xegaGeGene’

April 17, 2025

**Title** Grammatical Evolution

**Version** 1.0.0.3

**Description** Grammatical evolution (see O'Neil, M. and
Ryan, C. (2003,ISBN:1-4020-7444-1)) uses decoders to
convert linear (binary or integer genes) into programs.
In addition, automatic determination of codon precision
with a limited rule choice bias is provided.
For a recent survey of grammatical evolution,
see Ryan, C., O'Neill, M., and Collins, J. J. (2018)
<doi:10.1007/978-3-319-78717-6>.

**License** MIT + file LICENSE

**URL** https://github.com/ageyerschulz/xegaGeGene

**Encoding** UTF-8

**RoxygenNote** 7.3.2

**Suggests** testthat (>= 3.0.0)

**Imports** numbers, xegaSelectGene, xegaBNF, xegaDerivationTrees

**NeedsCompilation** no

**Author** Andreas Geyer-Schulz [aut, cre]
(<https://orcid.org/0009-0000-5237-3579>)

**Maintainer** Andreas Geyer-Schulz <Andreas.Geyer-Schulz@kit.edu>

**Repository** CRAN

**Date/Publication** 2025-04-17 07:20:01 UTC

# Contents

---

`ChoiceVector` *Choice vector of a grammar.*

---

## Description

Choice vector of a grammar.

## Usage

```
ChoiceVector(LHS)
```

## Arguments

| | |
|---|---|
| `LHS` | Vector of Integers. The left-hand side `G$LHS` of a grammar object `G`. |

## Value

Vector of the number of choices for non-terminal symbols.

## See Also

Other Utility: `mLCMG()`

## Examples

```
NT<-sample(5, 50, replace=TRUE)
ChoiceVector(NT)
```

CodonChoiceBiases *Biases in Rule Choice.*

## Description

Measures the biases in rule choice for each non-terminal. Statistics computed:

- dP: Difference in probability between random choice with equal probability and modulo rule with a codon `precision`.
- dH: Difference in entropy between random choice with equal probability and modulo rule with a codon `precision`.

## Usage

```
CodonChoiceBiases(cv, precision)
```

## Arguments

| | |
|---|---|
| `cv` | Choice vector of grammar. |
| `precision` | Number of bits of codon. |

## Value

Data frame with the following columns

- `$precision`: Number of bits.
- `$cv`: i-th element of the choice vector.
- `$dp`: Deviation from choice with equal probability for `$precision`.
- `$dH`: Entropy difference between choice with equal probability and biased choice for `$precision`.

## See Also

Other Diagnostics: `CodonChoiceBiasesDeprecated()`, `CodonPrecision()`, `tLCM()`

## Examples

```
CodonChoiceBiases(c(1, 2, 3, 5), 3)
CodonChoiceBiases(c(1, 2, 3, 5), 5)
```

CodonChoiceBiasesDeprecated
*Biases in Rule Choice (Deprecated)*

## Description

See `CodonChoiceBiases`. The use of the outer product leads to memory problems for `precision>31` and becomes slow for `precision>24`.

## Usage

```
CodonChoiceBiasesDeprecated(cv, precision)
```

## Arguments

| | |
|---|---|
| `cv` | Choice vector of grammar. |
| `precision` | Number of bits of codon. |

## Value

Data frame with the following columns

- `$precision`: Number of bits.
- `$cv`: i-th element of choice the vector.
- `$dp`: Deviation from choice with equal probability for `$precision`.
- `$dH`: Entropy difference between choice with equal probability and biased choice for `$precision`.

## See Also

Other Diagnostics: `CodonChoiceBiases()`, `CodonPrecision()`, `tLCM()`

## Examples

```
CodonChoiceBiasesDeprecated(c(1, 2, 3, 5), 3)
CodonChoiceBiasesDeprecated(c(1, 2, 3, 5), 5)
```

`CodonPrecision` *Compute codon precision with the choice bias of rules below a threshold.*

## Description

For automatic determination of the least codon precision for grammar evolution with an upper threshold on the choice bias for the substitution of all non-terminal symbols.

## Usage

```
CodonPrecision(cv, pCrit)
```

## Arguments

| | |
|---|---|
| `cv` | Choice vector of a context-free grammar. |
| `pCrit` | Threshold for choice bias. |

## Value

Precision of codon.

## See Also

Other Diagnostics: `CodonChoiceBiases()`, `CodonChoiceBiasesDeprecated()`, `tLCM()`

## Examples

```
CodonPrecision(c(1, 2, 3, 5), 0.1)
CodonPrecision(c(1, 2, 3, 5), 0.01)
```

`CodonPrecisionWithThreshold` *Precision of a codon which has a choice bias below a probability threshold.*

## Description

The choice bias is the sum of the absolute values of the difference between a k equally probable choices and the probability distribution of the modulo choice rule.

## Usage

```
CodonPrecisionWithThreshold(LHS, pCrit)
```

### Arguments

| | |
|---|---|
| `LHS` | The left-hand side of a grammar object `G`. |
| `pCrit` | Threshold for the choice bias for a single non-terminal. |

### Details

For the computation of the precision, the function `CodonPrecision()` is used.

### Value

The precision of a codon which guarantees that the choice bias for all non-terminals is below a probability threshold of `pCrit`.

### See Also

Other Precision: `MinCodonPrecision()`, `mLCMGCodonPrecision()`

### Examples

```
NT<-sample(5, 50, replace=TRUE)
CodonPrecisionWithThreshold(NT, 0.1)
CodonPrecisionWithThreshold(NT, 0.01)
```

---

## `lFxegaGeGene` *The local function list lFxegaGeGene.*

---

### Description

We enhance the configurability of our code by introducing a function factory. The function factory contains all the functions that are needed for defining local functions in genetic operators. The local function list keeps the signatures of functions (e.g. mutation functions) uniform and small. At the same time, variants of functions can use different local functions.

### Usage

```
lFxegaGeGene
```

### Format

An object of class `list` of length 23.

### Details

We use the local function list for

1. replacing all constants by constant functions.

   Rationale: We need one formal argument (the local function list lF) and we can dispatch multiple functions. E.g. `lF$verbose()`

2. dynamically binding a local function with a definition from a proper function factory. E.g., the selection methods `lF$SelectGene()` and `lF$SelectMate()`.

3. gene representations which require special functions to handle them: For example, `lF$InitGene()`, `lF$DecodeGene()`, `lF$EvalGene()`, `lF$ReplicateGene()`, ...

---

`MinCodonPrecision` *Minimal precision of codon.*

---

### Description

The minimal precision of the codon needed for generating a working decoder for a context-free grammar `G`. However, the decoder has some choice bias which reduces the efficiency of grammar evolution.

### Usage

```
MinCodonPrecision(LHS, ...)
```

### Arguments

| | |
|---|---|
| `LHS` | Vector of Integers. The left-hand side of a grammar object `G`. |
| `...` | Unused. Needed for the common abstract interface of precision functions. |

### Value

Integer. The precision of a codon whose upper bound is the least power of 2 above the maximum number of rules for a non-terminal of a grammar.

### See Also

Other Precision: `CodonPrecisionWithThreshold()`, `mLCMGCodonPrecision()`

### Examples

```
NT<-sample(5, 50, replace=TRUE)
MinCodonPrecision(NT)
```

`mLCMG` *Compute the mLCM of the vector of the number of production rules in a production table.*

## Description

Compute the least common multiple of the prime factors of the vector of the number of rules applicable for each non-terminal symbol.

## Usage

```
mLCMG(LHS)
```

## Arguments

| | |
|---|---|
| `LHS` | Vector of integers. The left-hand side of a grammar object `G`. |

## Details

For removing the bias of the modulo rule in grammatical evolution, see Keijzer, M., O'Neill, M., Ryan, C. and Cattolico, M. (2002). This version works for integer genes coded in the domain `1:mlCM` without bias in choosing a rule. See Keijzer et al. (2002). However, if the mLCM and `2^k` are relative prime, it is impossible to find an unbiased binary coding. The choice bias is considerably lower than for `MinCodonPrecision()`.

## Value

Integer. The least common multiple of the vector of the available rules for each non-terminal.

## References

Keijzer, M., O'Neill, M., Ryan, C. and Cattolico, M. (2002) Grammatical Evolution Rules: The Mod and the Bucket Rule, pp. 123-130. In: Foster, J. A., Lutton, E., Miller, J., Ryan, C. and Tettamanzi, A. (Eds.): Genetic Programming. Lecture Notes in Computer Science, Vol.2278, Springer, Heidelberg. <doi:10.1007/3-540-45984-7_12>

## See Also

Other Utility: `ChoiceVector()`

## Examples

```
library(xegaBNF)
g<-compileBNF(booleanGrammar())
mLCMG(g$PT$LHS)
```

## mLCMGCodonPrecision — *mLCMG precision of codon.*

### Description

mLCMG precision of codon.

### Usage

```
mLCMGCodonPrecision(LHS, ...)
```

### Arguments

| | |
|---|---|
| LHS | Vector of Integers. The left-hand side of a grammar object G. |
| ... | Unused. Needed for the common abstract interface of precision functions. |

### Value

Integer. The precision of a codon whose upper bound is larger than the least common multiple of the prime factors of the vector of the available rules for each non-terminal of a grammar.

### See Also

Other Precision: CodonPrecisionWithThreshold(), MinCodonPrecision()

### Examples

```
NT<-sample(5, 50, replace=TRUE)
mLCMGCodonPrecision(NT)
```

## tLCM — *Computes the largest least common multiple of all prime factors of the integers in the interval 1:m for k-bit integers.*

### Description

For 64 bit numbers, numerically stable up to m==42. The modulo rule in grammatical evolution assigns to the choices of substitutions for a non-terminal slightly (biased) probabilities. For an integer coding, the least common multiple of all rule choices from no choice (1) to the maximal number of substitutions of a non-terminal removes this bias completely. However, whenever the prime factors of the least common multiple contain a prime different from 2, the bias cannot be removed completely for a binary gene coding. However, each additional bit used for coding approximately halves the bias.

### Usage

```
tLCM(k)
```

### Arguments

| | |
|---|---|
| `k` | Number of bits. |

### Details

This could be done with the help of the function mLCM of the R-package `numbers`. We implement this by enumerating the vector of prime factors in `1:42`.

### Value

A list of three elements:

- `$k`: The number of bits.
- `$m`: Maximal number of substitutions for a non-terminal symbol in a grammar.
- `$mLCM`: Least common multiple of the prime factors of all rule choices from 1 to `$m`.

### See Also

Other Diagnostics: `CodonChoiceBiases()`, `CodonChoiceBiasesDeprecated()`, `CodonPrecision()`

### Examples

```
tLCM(8)
tLCM(16)
tLCM(32)
```

---

## `xegaGeDecodeGene` *Decode a binary gene for a context-free grammar.*

---

### Description

`xegaGeDecodeGene()` decodes a binary gene with a context-free grammar.

### Usage

```
xegaGeDecodeGene(gene, lF)
```

### Arguments

| | |
|---|---|
| `gene` | Binary gene. |
| `lF` | Local configuration of the genetic algorithm. |

### Details

The codons (k-bit sequences) of the binary gene determine the choices of non-terminal symbols of a depth-first left-to-right tree traversal. Decoding works in 3 steps:

1. From the binary gene and a grammar a potentially incomplete derivation tree is built.
2. The leaves of the derivation tree are extracted.
3. By reusing the random integer vector and by using a non-recursive grammar, all non-terminals are randomly completed.

It is guaranteed that a complete and syntactically correct program is returned.

### Value

Decoded gene.

### See Also

Other Decoder: `xegaGeDecodeGeneDT()`, `xegaGeGeneMapMod()`, `xegaGeGeneMapmLCM()`

### Examples

```
lFxegaGeGene$GeneMap<-xegaGeGeneMapFactory("Mod")
gene<-xegaGeInitGene(lFxegaGeGene)
xegaGeDecodeGene(gene, lFxegaGeGene)
```

---

## `xegaGeDecodeGeneDT` *Decode a binary gene for a context-free grammar.*

---

### Description

`xegaGeDecodeGene()` decodes a binary gene with a context-free grammar.

### Usage

```
xegaGeDecodeGeneDT(gene, lF)
```

### Arguments

| | |
|---|---|
| `gene` | Binary gene. |
| `lF` | Local configuration of the genetic algorithm. |

## Details

The codons (k-bit sequences) of the binary gene determine the choices of non-terminal symbols of a depth-first left-to-right tree traversal. Decoding works in 2 steps:

1. From the binary gene and a grammar a potentially incomplete derivation tree is built.
2. The leaves of the derivation tree are extracted.

It is not guaranteed that a complete derivation trees is returned. Therefore, the generated program may fail.

## Value

Decoded gene (a program)

## See Also

Other Decoder: `xegaGeDecodeGene()`, `xegaGeGeneMapMod()`, `xegaGeGeneMapmLCM()`

## Examples

```
lFxegaGeGene$GeneMap<-xegaGeGeneMapFactory("Mod")
gene<-xegaGeInitGene(lFxegaGeGene)
xegaGeDecodeGeneDT(gene, lFxegaGeGene)
```

---

# xegaGeDecodeGeneFactory

*Configure the decoder function of a genetic algorithm.*

---

## Description

`xegaGeDecodeGeneFactory()` implements the selection of one of a decoder function by specifying a text string. The selection fails ungracefully (produces a runtime error) if the label does not match. The functions are specified locally.

Current support:

1. "DecodeGene" returns `xegaGeDecodeGene()`. (Default).
2. "DecodeGeneDT" returns `xegaGeDecodeGeneDT()`. This decoder does not guarantee complete programs.

## Usage

```
xegaGeDecodeGeneFactory(method = "DecodeGene")
```

## Arguments

| | |
|---|---|
| `method` | A string specifying a decoder for genes |

### Value

A decoder for genes.

### See Also

Other Configuration: `xegaGeGeneMapFactory()`, `xegaGePrecisionFactory()`

### Examples

```
lFxegaGeGene$GeneMap<-xegaGeGeneMapFactory("Mod")
gene<-xegaGeInitGene(lFxegaGeGene)
DecodeGene<-xegaGeDecodeGeneFactory("DecodeGene")
DecodeGene(gene, lFxegaGeGene)
DecodeGeneDT<-xegaGeDecodeGeneFactory("DecodeGeneDT")
DecodeGeneDT(gene, lFxegaGeGene)
```

---

## `xegaGeGene` *Package xegaGeGene.*

---

### Description

The `xegaGeGene` package provides functions implementing grammatical evolution with binary-coded genes:

### Details

- Gene initialization.
- Gene maps for the mod and (approximately) for the bucket rule.
- Grammar-based decoders for binary coded genes.
- Analysis of the interaction of codon precision with the rule choice bias for a given grammar.
- Automatic determination of codon precision with a limited rule choice bias.

### Gene Initialization

The number of bits of a gene are specified by `lF$BitsOnGene()`.

The number of bits of a codon are specified by `lF$CodonPrecision()`.

### Binary Gene Representation

A binary gene is a named list:

- `$gene1`: The gene must be a binary vector.
- `$fit`: The fitness value of the gene (for EvalGeneDet and EvalGeneU) or the mean fitness (for stochastic functions evaluated with EvalGeneStoch).
- `$evaluated`: Has the gene been evaluated?
- `$evalFail`: Has the evaluation of the gene failed?

- `$var`: The cumulative variance of the fitness of all evaluations of a gene. (For stochastic functions)
- `$sigma`: The standard deviation of the fitness of all evaluations of a gene. (For stochastic functions)
- `$obs`: The number of evaluations of a gene. (For stochastic functions)

### Abstract Interface of Problem Environment

A problem environment `penv` must provide:

- `$f(parameters, gene, lF)`: Function with a real parameter vector as the first argument which returns a gene with evaluated fitness.
- $genelength(): The number of bits of the binary-coded integer parameter vector. Used in `InitGene`.
- $bitlength(): A vector specifying the number of bits used for coding each integer parameter. If `penv$bitlength()[1]` is `20`, then `parameters[1]` is coded by 20 bits. Used in `GeneMap`.
- $lb(): The lower bound vector of each parameter. Used in `GeneMap`.
- $ub(): The upper bound vector of each parameter. Used in `GeneMap`.

### The Architecture of the xegaX-Packages

The xegaX-packages are a family of R-packages which implement eXtended Evolutionary and Genetic Algorithms (xega). The architecture has 3 layers, namely the user interface layer, the population layer, and the gene layer:

- The user interface layer (package `xega`) provides a function call interface and configuration support for several algorithms: genetic algorithms (sga), permutation-based genetic algorithms (sgPerm), derivation-free algorithms as e.g. differential evolution (sgde), grammar-based genetic programming (sgp) and grammatical evolution (sge).
- The population layer (package `xegaPopulation`) contains population-related functionality as well as support for population statistics dependent adaptive mechanisms and parallelization.
- The gene layer is split in a representation-independent and a representation-dependent part:
  1. The representation-indendent part (package `xegaSelectGene`) is responsible for variants of selection operators, evaluation strategies for genes, as well as profiling and timing capabilities.
  2. The representation-dependent part consists of the following packages:
     - `xegaGaGene` for binary coded genetic algorithms.
     - `xegaPermGene` for permutation-based genetic algorithms.
     - `xegaDfGene` for derivation-free algorithms as e.g. differential evolution.
     - `xegaGpGene` for grammar-based genetic algorithms.
     - `xegaGeGene` for grammatical evolution algorithms.

     The packages `xegaDerivationTrees` and `xegaBNF` support the last two packages: `xegaBNF` essentially provides a grammar compiler and `xegaDerivationTrees` is an abstract data type for derivation trees.

### Copyright



### License



### URL

https://github.com/ageyerschulz/xegaGeGene

### Installation

From CRAN by `install.packages('xegaGeGene')`

### Author(s)

Andreas Geyer-Schulz

### References

Ryan, Conor and Collins, J. J. AND Neill, Michael O. (1998) Grammatical evolution: Evolving programs for an arbitrary language. In: Banzhaf, Wolfgang and Poli, Riccardo, Schoenauer, Marc and Fogarty, Terence C. (1998): Genetic Programming. First European Workshop, EuroGP' 98 Paris, France, April 14-15, 1998 Proceedings, Lecture Notes in Computer Science, 1391, Springer, Heidelberg. <doi:10.1007/BFb0055930>

O'Neil, Michael AND Ryan, Conor (2003) Grammatical Evolution: Evolutionary Automatic Programming in an Arbitrary Language. Kluwer, Dordrecht. <ISBN:1-4020-7444-1>

Ryan, Conor and O'Neill, Michael and Collins, J. J. (2018) Handbook of Grammatical Evolution. Springer International Publishing, Cham. <doi:10.1007/978-3-319-78717-6>

### See Also

Useful links:

- `https://github.com/ageyerschulz/xegaGeGene`

---

## `xegaGeGeneMapFactory` *Configure the gene map function of a genetic algorithm for grammar evolution.*

## Description

`xegaGeGeneMapFactory()` implements the selection of one of the GeneMap functions in this package by specifying a text string. The selection fails ungracefully (produces a runtime error), if the label does not match. The functions are specified locally.

Current support:

1. "Mod" returns `xegaGeGeneMapMod()`. (Default).
2. "Bucket" returns `xegaGeGeneMapmLCM()`.

## Usage

```
xegaGeGeneMapFactory(method = "Mod")
```

## Arguments

| | |
|---|---|
| `method` | String specifying the GeneMap function. |

## Value

Gene map function for genes.

## See Also

Other Configuration: `xegaGeDecodeGeneFactory()`, `xegaGePrecisionFactory()`

## Examples

```
XGene<-xegaGeGeneMapFactory("Mod")
gene<-xegaGeInitGene(lFxegaGeGene)
XGene(gene$gene1, lFxegaGeGene)
```

---

`xegaGeGeneMapmLCM` *Map the bit strings of a binary gene to integer parameters in the interval* 1 *to* `numbers::mLCM(x) < 2^k`.

---

## Description

`xegaGeGeneMapmLCM()` maps the bit strings of a binary string to integers in the interval 1 to `lF$CodonPrecision()`. Bit vectors are mapped into equispaced numbers in the interval.

## Usage

```
xegaGeGeneMapmLCM(gene, lF)
```

## Arguments

| | |
|---|---|
| `gene` | Binary gene (the genotype). |
| `lF` | Local configuration. |

### Details

Using the interval of `1` to `numbers::mLCM(1:m)` provides the least common multiple of all prime factors of the numbers in the interval `1:m`. This corresponds to the bucket rule of Keijzer et al. (2002). For 16-bit precision, the highest number of rules for the same non-terminal symbols is 12. For 8-bit precision, this reduces to 6. With 64-bit integer arithmetic, the bucket rule works up to 42 rules starting with the same non-terminal.

### Value

Integer vector.

### References

Keijzer, M., O'Neill, M., Ryan, C. and Cattolico, M. (2002) Grammatical Evolution Rules: The Mod and the Bucket Rule, pp. 123-130. In: Foster, J. A., Lutton, E., Miller, J., Ryan, C. and Tettamanzi, A. (Eds.): Genetic Programming. Lecture Notes in Computer Science, Vol.2278, Springer, Heidelberg. <doi:10.1007/3-540-45984-7_12>

### See Also

Other Decoder: `xegaGeDecodeGene()`, `xegaGeDecodeGeneDT()`, `xegaGeGeneMapMod()`

### Examples

```
gene<-xegaGeInitGene(lFxegaGeGene)
xegaGeGeneMapmLCM(gene$gene1, lFxegaGeGene)
```

---

## `xegaGeGeneMapMod` *Map the bit strings of a binary gene to parameters in the interval* `1:2^k`.

---

### Description

`xegaGeGeneMapMod()` maps the bit strings of a binary string to integers in the interval `1` to `lF$CodonPrecision()`. Bit vectors are mapped into equispaced numbers in the interval.

### Usage

```
xegaGeGeneMapMod(gene, lF)
```

### Arguments

| | |
|---|---|
| `gene` | Binary gene (the genotype). |
| `lF` | Local configuration. |

### Details

The modulo rule of grammatical evolution produces (slightly) biased choices of rules with this mapping. The bias goes to zero as `lF$CodonPrecision()` >> number of rules to choose from.

### Value

Integer vector.

### References

Keijzer, M., O'Neill, M., Ryan, C. and Cattolico, M. (2002) Grammatical Evolution Rules: The Mod and the Bucket Rule, pp. 123-130. In: Foster, J. A., Lutton, E., Miller, J., Ryan, C. and Tettamanzi, A. (Eds.): Genetic Programming. Lecture Notes in Computer Science, Vol.2278, Springer, Heidelberg. <doi:10.1007/3-540-45984-7_12>

### See Also

Other Decoder: `xegaGeDecodeGene()`, `xegaGeDecodeGeneDT()`, `xegaGeGeneMapmLCM()`

### Examples

```
gene<-xegaGeInitGene(lFxegaGeGene)
xegaGeGeneMapMod(gene$gene1, lFxegaGeGene)
```

---

## `xegaGeInitGene` *Initialize a binary gene*

---

### Description

`xegaGeInitGene()` generates a random binary gene with a given length.

### Usage

```
xegaGeInitGene(lF)
```

### Arguments

| | |
|---|---|
| `lF` | the local configuration of the genetic algorithm |

### Details

In the binary representation of package `xegaGeGene`, a *gene* is a list with

1. `$evaluated` Boolean: TRUE if the fitness is known.
2. `$fit` The fitness of the genotype of `$gene1`
3. `$gene1` a bit string (the genetopye).

### Value

A binary gene (a named list):

- `$evaluated`: FALSE. See package `xegaSelectGene`
- `$evalFail`: FALSE. Set by the error handler(s) in package `xegaSelectGene` in the case of failure.
- `$fit`: Fitness vector.
- `$gene1`: Binary gene.

### Examples

```
xegaGeInitGene(lFxegaGeGene)
```

---

## xegaGePrecisionFactory

*Configure the function for computing the codon precision for grammar evolution.*

---

### Description

`xegaGePrecisionFactory()` implements the selection of one of the functions for computing the codon precision in this package by specifying a text string. The selection fails ungracefully (produces a runtime error), if the label does not match. The functions are specified locally.

Current support:

1. "Min" returns `MinCodonPrecision`. Shortest coding, but some choice bias.
2. "LCM" returns `mLCMGCodonPrecision`. (Default)
3. "MaxPBias" returns `CodonPrecisionWithThreshold`.

### Usage

```
xegaGePrecisionFactory(method = "LCM")
```

### Arguments

| | |
|---|---|
| `method` | String specifying the GeneMap function. |

### Value

Precision of codon function.

### See Also

Other Configuration: `xegaGeDecodeGeneFactory()`, `xegaGeGeneMapFactory()`

## Examples

```
CodonPrecision<-xegaGePrecisionFactory("Min")
NT<-sample(5, 50, replace=TRUE)
CodonPrecision(NT)
CodonPrecision<-xegaGePrecisionFactory("MaxPBias")
CodonPrecision(NT, 0.1)
```

# Index