EVOLVER

User Guide

Table of Contents

Evolver Overview

Evolver is a collection of programs designed to simulate the evolution of the nucleotide sequence of a whole genome.

Caveat emptor

Evolver is a powerful and complex tool. It is not easy to learn or to use, and requires days or weeks on a typical compute cluster to run large genomes over interesting evolutionary distances. Users should expect to devote much more time and energy to Evolver than to a typical bioinformatics tool. A sophisticated understanding of genome evolution and computer science is required and is assumed in this guide.

What is a genome?

Evolver simulates the evolution of a representative genome of a species over periods long compared with its generation time. From Evolver's perspective, a "genome" is thus a population average rather than a single individual. It is not designed to simulate population genetics; there is no explicit model of allele frequencies, gene flow and so on. Rather, it simulates the long-term averaged effects of mutation and selection over an entire species.

Inter- and intra-chromosome modules

The core components simulate inter- and intra-chromosome evolution, respectively. The inter-chromosome module (*inter*) simulates events involving two chromosomes, including chromosome fission, fusion and segmental moves and copies in which the target chromosome is different from the source chromosome. The intra-chromosome module (*intra*) simulates events occurring within a single chromosome, including substitutions, insertions, deletions, moves, copies and so on. This division is driven by software engineering rather than biological considerations: it is currently possible to simulate evolution of a single chromosome on a commodity computer typically found in a compute cluster, but an entire genome would require too much memory and time. This design also enables intra be run on each chromosome in parallel, reducing the wall-clock time for a typical simulation by an order of magnitude—just enough to make mammals tractable.

Input data

Evolver requires:

- an ancestral genome sequence,
- annotations describing the ancestral genome including genes, non-gene conserved elements, tandem arrays / microsatellites and CpG islands,
- a library of mobile element and retroposed pseudo-gene sequences, and

- a parameter file specifying a model of evolution including rates for each type of mutation, amino acid substitution probabilities and so on.

Output data

Evolver produces:

- alignments of the evolved genomes to each other and to their common ancestor,
- annotations of the evolved genome(s), and
- statistics of evolutionary events (e.g. number of accepted and rejected substitutions) and genome characteristics (e.g. intron length distribution).

Events

There are two classes of evolutionary event: *mutations*, which modify the primary sequence, and *constraint changes*, which modify annotations while leaving the primary sequence unchanged. Examples of mutations are substitutions, insertions and deletions. Examples of constraint changes include exon gain and loss.

Accept probability

Every base in the genome has an *accept probability*, denoted α, with a value $k/255$, k = 0... 255. If a mutation affects one base then it will usually be accepted with probability α. If a mutation affects multiple bases then it will generally be accepted with a probability that is the product of α for each base (making the probability of accepting a multiple-base event equal to the probability of accepting all of the equivalent sequence of single-base events). Bases with α=0 are fully constrained and will never undergo a mutation, bases with α=1 are neutral and will always undergo a proposed mutation (unless the mutation also affects a base with $\alpha < 1$). The default state of a base is neutral; no annotation is needed for unconstrained regions.

Mutations

Evolver proposes mutations at rates specified by parameters in the model. Each mutation is accepted or rejected with a probability determined by constraints on affected bases. If no annotations are provided, there are no constraints and all mutations will be accepted. Types of mutation are listed in the following tables.

Name	Length distribution	Description
InterChrCopy	Yes	A segment of one chromosome is duplicated and inserted into a different chromosome.
InterChrMove	Yes	A segment of one chromosome is deleted and inserted into a different chromosome.
ChrSplit	No	A chromosome is divided into two new chromosomes.
ChrFuse	No	Two chromosomes are fused into one new chromosome.
RecipTransloc	No	Reciprocal translocation.
NonRecipTransloc	No	Non-reciprocal translocation.

Inter-chromosome mutation events.

Name	Length distribution	Description
Substitute	No	A single base is changed.
Delete	Yes	Deletion.
Invert	Yes	Inversion.
Move	Yes	A segment is moved to a new location.
Copy	Yes	A copy of a segment is made and inserted at a new location.
Tandem	Yes	Special case of Copy in which the insertion immediately follows the copied segment.
TandemExpand	No	An existing tandem array is expanded by duplicating one instance of the repeated motif.
TandemContract	No	An existing tandem array is contracted by deleting one instance of the repeated motif.
Insert	Yes	A random sequence is inserted.
MEInsert	No	A library sequence is inserted. This is used to model both mobile elements and retroposed pseudo-genes.

Intra-chromosome mutation events.

When an insertion point is required (Insert, MEInsert, [IterChr]Copy and [InterChr]Move events), the location is selected with uniform probability over the entire chromosome. Events that require a segment (Delete, Invert, [InterChr]Move, [InterChr]Copy, Tandem and Insert) similarly sample the start of the segment uniformly over the chromosome.

Constraint change events

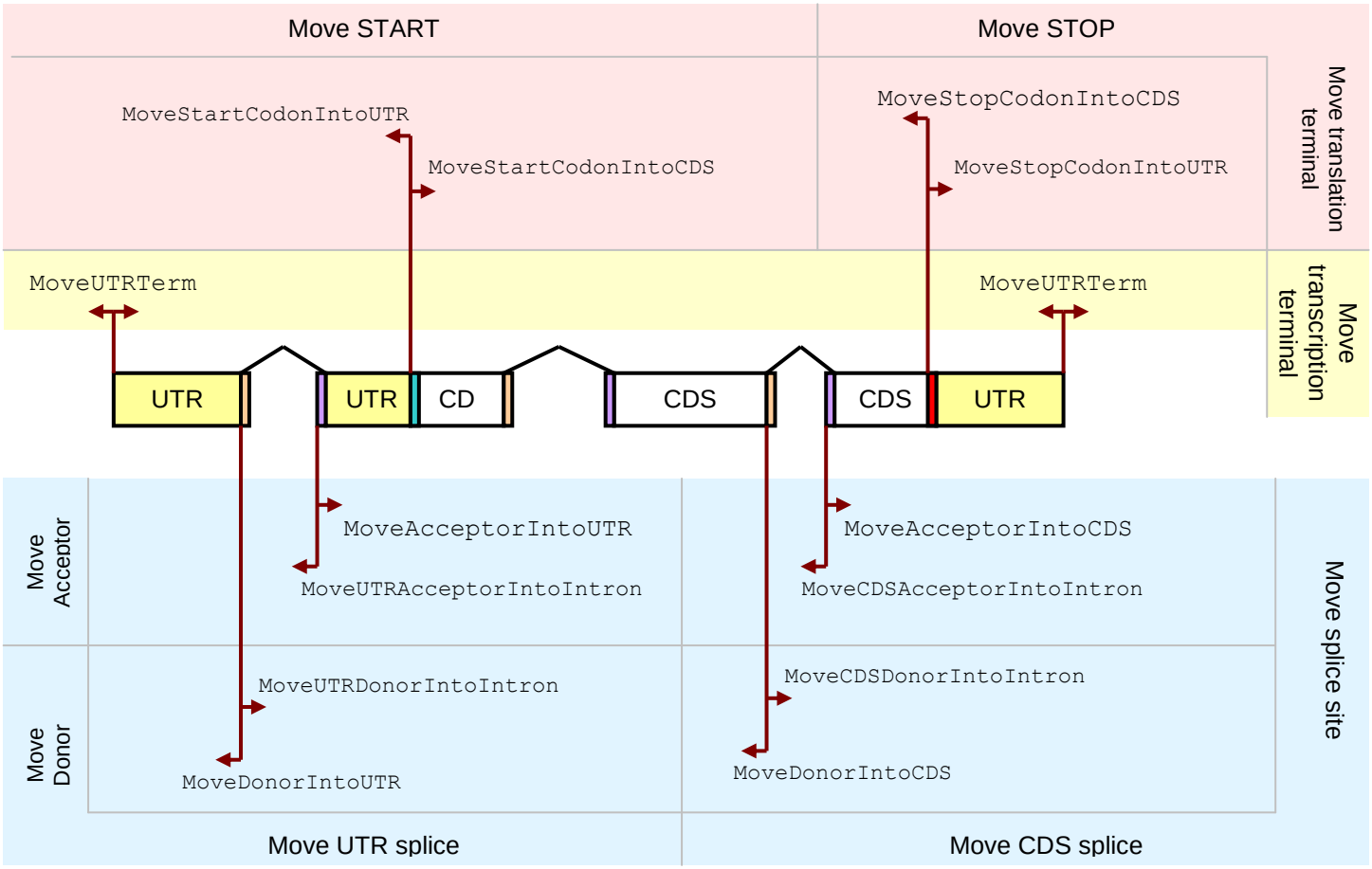
Constraint change events modify genome annotation, leaving the primary sequence unchanged. These events are unconditionally accepted as the "reject according to constraint" paradigm does not apply. Typically a region is selected with a weight equal to the mean accept probability (MAC) of its constrained bases. Thus, more rapidly evolving elements are more likely to be affected. For example, an exon loss event starts by

selecting a gene weighted by the MAC of all bases in all CEs of that gene. If any exons can be deleted while leaving a valid protein coding gene, one is selected at random (with uniform probability in this case) and deleted; otherwise another gene is selected, again weighted by MAC, until a suitable gene is found. Constraint change events are listed in the following table.

Event	**Description**
CreateCDS	Create a new coding exon from existing intron sequence.
CreateNGE	Create a new NGE by copying constraint from an existing NGE.
CreateNontermUTR	Create a non-terminal UTR exon in an existing intron by finding new splice site signals (two-base donor and two-base acceptor) that match the current sites.
CreateNXE	Create a new NXE (non-exon gene element) by copying constraint from an existing NXE
CreateTermUTR	Create a new first or last UTR exon.
DeleteCDS	Convert a coding exon into neutral sequence.
DeleteNGE	Convert an NGE to neutral sequence.
DeleteNXE	Convert an NXE to neutral sequence.
DeleteUTR	Convert a UTR to neutral sequence.
MoveAcceptorCDS	Move an acceptor splice site into a coding exon.
MoveAcceptorIntron	Move an acceptor splice site into its intron.
MoveDonorCDS	Move a donor splice site into a coding exon.
MoveDonorIntron	Move donor splice site into its intron.
MoveNGE	Move an NGE.
MoveNXE	Move an NXE.
MoveStartCDS	Move a START codon into a CDS, shortening the CDS.

Event	Description
MoveStartUTR	Move start codon into UTR, lengthening the CDS.
MoveStopCDS	Move stop codon into CDS, shortening the CDS.
MoveStopUTR	Move stop codon into UTR, lengthening the CDS.
MoveUTREnd	Move beginning or end of transcription.

Constraint change events.



Constraint change events that modify gene structure.

Time

Evolver uses an arbitrary unit of time we call a *tick*. We typically set rates in our models so that one tick is approximately equal to one neutral substitution per site (*NSPS*). However, the NSPS unit is fraught with subtle problems and we prefer to regard measures of neutral substitution rate such as four-fold degenerate sites as emergent properties of the genome and model rather than a fundamental measure of time.

Event rates

The rate of an event is generally specified as:

Number of events per object per tick.

The "object" will be a base, a chromosome, a gene etc., as appropriate. Where possible, the object is chosen to allow the rate parameter to be independent of the genome sequence and annotations. For example, the rate of UTR loss is specified as UTRs lost per exon per tick. Thus if the genome has 10^5 UTRs and the rate of UTR loss is 10^{-4} the average number of UTRs lost in the genome will be 10 per tick.

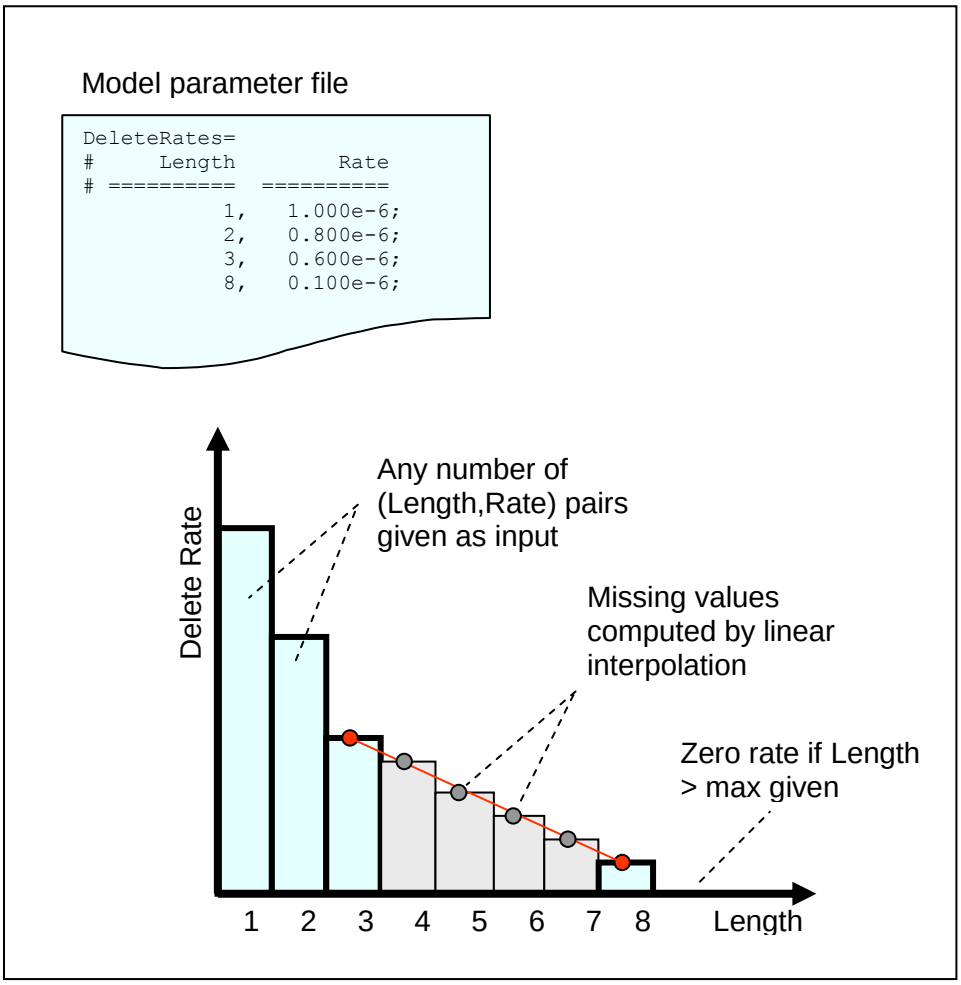
Most events use rates per base. The following table lists the exceptions. A *tandem base* is a base that was annotated as being in a tandem array or was created by a *Tandem* or *TandemExpand* event.

Event	Object
TandemExpand	Tandem base
TandemContract	Tandem base
MoveStartCodonIntoUTR	Gene
MoveStartCodonIntoCDS	Gene
MoveStopCodonIntoUTR	Gene
MoveStopCodonIntoCDS	Gene
MoveUTRTerm	Gene
DeleteUTR	UTR
CreateNontermUTR	UTR
CreateTermUTR	UTR
DeleteCDS	CDS
CreateCDS	CDS
MoveDonorIntoCDS	CDS
MoveCDSDonorIntoIntron	CDS
MoveAcceptorIntoCDS	CDS
MoveCDSAcceptorIntoIntron	CDS
MoveDonorIntoUTR	UTR
MoveUTRDonorIntoIntron	UTR
MoveAcceptorIntoUTR	UTR
MoveUTRAcceptorIntoIntron	UTR
DeleteNXE	NXE
CreateNXE	NXE
MoveNXE	NXE
DeleteNGE	NGE
CreateNGE	NGE
MoveNGE	NGE
DeleteIsland	Island
CreateIsland	Island
MoveIsland	Island
ChangeGeneSpeed	Gene
ChangeNGESpeed	NGE

Length distributions

Some events, such as deletions and inversions, can occur at any length scale from a single base to a large segment. In these cases each length is conceptually a separate type of

event, each with its own rate. The model specifies rates for some subset of possible lengths; rates for lengths not specified in the model are determined by linear interpolation as shown in the following figure.



Constrained elements and gene model

Constrained elements (CEs) are sets of consecutive bases with $\alpha<1$. Evolver recognizes four CE types:

- CDS (protein-coding sequence),
- UTR (untranslated exonic sequence),
- NXE (non-exonic CE in a gene), and
- NGE (non-gene element, i.e. a CE that is not part of a gene).

Genes are protein coding; there is no explicit model of RNA genes so base pairing etc. is not modeled. Genes are defined by a range of positions: a CE belongs to a gene if and

only if it is found within the coordinate range of a gene. The two bases at the start and end of each intron are designated as splice sites and are fully constrained (α=0). Constraints are imposed in order to maintain gene structure, including that inversions must include all CEs of any affected gene (with one exception: a single NXE may be inverted with an accept probability that is a parameter of the model), and introns and UTRs may not become shorter than minimum lengths that are also parameters of the model. Annotations are required for all CEs except splice sites, which are implied by introns, which are in term implied by gaps between CDS and UTR annotations belonging to a single gene.

Proteins

Evolver explicitly models protein evolution. The CDS of a gene must begin with a start codon and must contain exactly one stop codon which is the last codon in the CDS. Rare anomalies such as in-frame stop codons are thus not modeled. Frame is maintained: the CDS length must always be a multiple of three and mutations cannot introduce an in-frame stop, change the start codon or substitute a stop codon with a translated codon. Constraint change events may change gene structure, however. Constraint change events leave the primary sequence unchanged, with the exception of those that move the position of the stop codon in which case one or two substitutions are made to preserve frame.

The accept probability of a substitution within a codon is computed as a special case as follows:

> α x (codon substitution probability) x *AminoAcidProbMultiplier*.

Here, α is the usual accept probability for the base, the codon substitution probability is specified by a table *AminoAcidAcceptProbs* in the model, and *AminoAcidProbMultiplier* is a parameter of the model. This allows the codon substitution probabilities to be computed directly from known, closely-related genomes. Then if *AminoAcidProb-Multiplier* is set to 1/(mean value of α in CDS bases) the effective codon substitution probabilities resulting from the simulation will be approximately those in *AminoAcid-AcceptProbs*.

Mobile elements

A mobile element (ME) is modeled as a nucleotide sequence that is inserted at a randomly chosen point in the chromosome. Optionally, a random segment of the ME is deleted before the sequence is inserted. ME sequences are provided in a FASTA file for the intra module. The FASTA annotation line specifies the insertion rate and deletion parameters, for example:

```
>LINE1; rate "1e-6"; avgdel "20.0"; stddev "5.0"; pct "25.0";
```

The name of the ME, which may not include a semi-colon, is followed by a list of attributes in GTF format:

> http://mblab.wustl.edu/GTF2.html

Attributes are:

rate
The insertion rate in units of MEs per chromosome base per tick.

rateperbase
The insertion rate in units of ME bases per chromosome base per tick.

avgdel
The mean number of bases to delete, if a deletion is done prior to insertion.

stddev
Standard deviation of the number of bases to delete.

pct
Percentage (0...100) of insertions that undergo a deletion prior to insertion.

Exactly one of *rate* or *rateperbase* must be specified. If one of *avgdel*, *stddev* or *pct* are specified, all three must be given.

If *rate* is given, the number of MEs N inserted into a chromosome of length L in one run of intra for t ticks will be approximately:

$$N = L \times rate \times t$$

The average number of bases included when an ME of length m is inserted is:

$$b = m - (avgdel \times pct) / 100$$

Thus, the number of ME bases T inserted into the chromosome will be approximately:

$$T = Nb = L \times rate \times t \times [\, m - (avgdel \times pct) / 100 \,].$$

If *rateperbase* is given, T will be approximately:

$$T = L \times rateperbase \times t.$$

This gives a more direct calculation of the proportional increase in chromosome size due to ME insertions, with a less direct calculation of the expected number of insertions N:

$$N = T/b = L \times rateperbase \times t / [\, m - (avgdel \times pct) / 100 \,].$$

See also the later section *Mobile Element Evolution*.

Retroposed pseudo-genes

Retroposed pseudo-genes are sequences derived from transcribed and spliced genes that are inserted back into the genome at random locations. They are implemented by extracting spliced sequences from the genome and adding them to the ME library.

Gene duplication

As happens in nature, gene duplication arises as a side effect of large segmental duplications. These are implemented in Evolver as *Copy* and *InterChrCopy* events. If a complete gene is copied by one of these events, then the new (copied) gene may remain active and / or the old (original) gene may become inactive according to a table specified in the model parameter file. For example,

```
GeneDupeWeights=
            OldSlower   OldSame OldFaster   OldLost
 NewSlower          0         0         0         0
   NewSame          0         0         1        10
 NewFaster          0        10        10         0
   NewLost          0       100         0         0
```

The numerical values give the relative probability of a given outcome; i.e. the probability is the number divided by the sum of all numbers. In the above table the sum is 131 so, for example, the probability of *NewFaster* and *OldSame* is 10/131 = 0.076. "Faster", "Slower" and "Same" refer to the rate at which the gene evolves, i.e. its mean accept probability (MAC). If a gene is faster, the accept probabilities of its bases are increased, and so on in the obvious way. "Lost" means that its bases are converted to neutral.

Cycles

The fundamental step in executing Evolver is to invoke the inter-chromosome simulator (*inter*) once for the entire genome, then the intra-chromosome simulator (*intra*) once for each chromosome. This process is called a *cycle*. The output from one cycle can be used as input to another cycle. It is generally better to run many short cycles rather than one or a few long cycles as longer cycles are less biologically accurate. This is because, viewed as operators that transform the genome sequence, inter and intra do not commute. For example, it is possible to estimate the time at which a segmental duplication occurred by measuring the sequence divergence between the two copies. If the entire simulation was run as one cycle, all segmental duplications would have approximately the same sequence divergence and thus appear to have happened at the same time.

File formats

Evolver uses the following file formats:

- FASTA files for sequence data,
- GFF files for annotations,
- Rev files, a binary file format for storing sequences and alignments specifically designed for Evolver and related projects,

- Model parameter files, text files designed to be human readable and writeable that specify parameters of the evolutionary model, and

- Log, report and statistics text files designed to be human or computer readable.

Rev files

We have developed our own file format for use in Evolver and some related projects. Called Rev and conventionally given the file extension *.rev*, this format stores:

- Information about one or more genomes, including the genome name and the name and length of one or more of its chromosomes,

- Zero or more chromosome sequences,

- Zero or more structures (*AlnInfo*'s or *AI*'s) describing alignments between two or more chromosome segments, and

- Zero or more sequence alignments (explicit row/column matrix representations specifying all gaps).

An *AlnInfo* (*AI*) stores data concerning one sequence alignment. For each segment (contiguous region of one chromosome), the AI stores the genome index (0, 1 … *G*–1 where *G* is the number of genomes), chromosome id (an arbitrary integer identifying a chromosome within its genome), start and stop coordinates (positions within the chromosome starting at zero for the first base), and strand (plus or minus). In addition, the AI may store one or more *attributes* of the alignment. An attribute is an (*id*, *value*) pair where *id* is a small integer identifying the attribute and *value* is an integer or floating-point constant.

An alignment always has an AI, but an AI does not necessarily have an alignment.

The binary format of a Rev file is complicated, and we do not at present intend to document it. The primary reasons for its complexity are (a) indexes that allow certain kinds of random access to be made efficiently, and (b) the need to achieve a high degree of compression: at larger evolutionary distances, most alignments are short segments of neutral DNA; in fact, the median length rapidly approaches one base. Existing file formats have a large overhead for short alignments. A back of the envelope calculation shows why. Take the MAF format used by UCSC as an example:

http://genome.ucsc.edu/FAQ/FAQformat#format5

Here is a short example:

```
s hg16.chr7    27707221 13 + 158545518 gcagctgaaaaca
s panTro1.chr6 28869787 13 + 161576975 gcagctgaaaaca
s baboon         249182 13 +   4622798 gcagctgaaaaca
s mm4.chr6     53310102 13 + 151104725 ACAGCTGAAAATA
```

Each segment has an 's', genome.chromosome, two numbers of the order of the chromosome length, a strand, spaces for padding, and the gapped sequence. Suppose the alignment is pair-wise and contains one letter per segment, the chromosome is ~100Mb long (9 digits), and genome.chromosome is 3.3=7 characters. The minimum number of spaces for padding is 6. The overhead per segment is therefore approximately 's'=1 + genome.chromosome=7 + 2 x chromsosome length=18 + segment length=1 + strand=1 + letters=1 + spacing=6 + 1 new-line byte for a total of 36 bytes per segment = 72 bytes per alignment. If we now suppose the genome is 3Gb long and the number of alignments is ~10% of the genome length, i.e. 300M, then the file size is ~300M x 72 = 20 Gb.

GFF files

Annotations are stored in GFF files. Evolver follows the GTF2 specification:

http://mblab.wustl.edu/GTF2.html

Note that GFF records use 1-based chromosome coordinates while Evolver generally uses 0-based coordinates. Evolver is aware of this and converts as needed.

The *source* and *score* fields are ignored by Evolver.

GFF record types

Evolver uses the following GFF feature types:

- *UTR*, for untranslated exon sequences,
- *CDS*, for coding sequences,
- *NXE*, for non-exon conserved elements in a gene,
- *NGE*, for non-gene conserved elements,
- *tandem*, for tandem arrays / micro-satellites, and
- *island*, for CpG islands (more properly, regions that are unmethylated in the germ line and therefore do not have an elevated C→T transition rate).

Accept probabilities

CE records (UTR, CDS, NXE and NGE) must have a *probs* attribute which specifies the accept probability of each base. Probabilities are represented as a string of hex digits, two for each base. The two digits give a value $k = 0 \ldots 255$; the probability is then $1/k$. So, for example, "00" represents α=0 (fully constrained) and "FF", if legal, would represent α=1 (neutral). However, Evolver forbids neutral bases within a CE, so the largest permitted

probability is "FE" = 254/255 = 0.996. The number of probabilities specified must exactly match the number of bases in the coordinate range specified in the record. Here is an example:

```
chr20 evo NGE 5870 5878 0 . . probs "b0f4aca8e3b3e3fda9";
```

Gene indexes

Each gene must be assigned an integer identifier that is unique within its chromosome. Every CE must give its gene using the *gene_index* attribute, for example:

```
chr20 evo NXE 9501 9514 0 . . gene_index 14; probs "200e0e0e1a0e0e0e400e0e180e7a";
```

Exons

Exons are specified using CDS and UTR records, each of which typically specifies one exon. If a single exon contains both CDS and UTR bases, this is implied by having records with consecutive coordinates. The *frame* field must be set in a CDS record.

Tandems

Tandem arrays and micro-satellites are specified using records with the *feature* field set to *tandem*. The required *replen* attribute specifies the length of the repeated motif. There is no provision for specifying truncated or extended copies of the motif (i.e., indels within the array). For example,

```
chr20 trf tandem 3858 3884 54 . . replen 12;
```

Evolver uses tandem annotations when executing *TandemExpand* and *TandemContract* events, which operate on existing tandem arrays. If no tandem annotations are included in the input to intra, the rates of these events will initially be zero, but new tandem arrays may be created by *Tandem* events which are then subject to *TandemExpand* and *Tandem-Contract* mutations.

The intra module keeps track of tandem arrays internally as the simulation progresses, but the quality of this internal annotation degrades over time because micro-mutations within an existing array are not taken into account and new exact or approximate arrays that are created by chance rather than by a *Tandem* event are not detected (this would require implementing a tandem finder algorithm and executing it on the fly). It is therefore recommended that a tandem finder be run at the beginning of each cycle. We typically use Gary Benson's Tandem Repeat Finder (TRF); other programs are also available. We provide a script *trf2gff.py* that converts the *.dat* files generated by TRF into the GFF format required by Evolver.

CpG islands

Island records have no attributes; they simply specify the coordinate range of an unmethylated region. For example,

```
chr20 evo island 187576 187851 0 . .
```

Gene structure

The user must ensure that CDS records for the ancestral genome specify a valid coding sequence for each gene. The first codon must be ATG, the last codon must be a stop, and there must be no in-frame stops. The beginning and end of a gene are determined as the lowest and highest coordinate in records for its gene index. It is illegal for an NGE to appear within a gene. Genes may not overlap, and in particular there is no provision for specifying alternative splicing structures.

Executables

Evolver includes three executable binary programs written in C++:

- *evo*: inter, intra and several utilities.
- *cvt*: manipulate Rev and other files, e.g. conversion to and from FASTA and GFF.
- *transalign*: compute a transitive alignment of genomes A and B from pair-wise alignments AC and CB with a third genome C.

Command-line reference for the inter module

Here is a typical command line for running the inter-chromosome simulator:

```
evo -interchr anc.seq.rev -inannots anc.annots.gff –aln inter.aln.rev        \
  -outchrnames chrnames.txt -outgenome ev -outannots inter.outannots.gff     \
  -outseq inter.outseq.rev –branchlength 0.001 -statsfile inter.stats        \
  -model model.txt -seed 1 -log inter.log
```

Options are as follows. Note that here the term "ancestral genome" means the genome used as input to this step; "evolved genome" means the genome generated by inter as a result of the simulation. The "original ancestral genome" means the input to the first cycle.

–interchr *revfilename*

[Input] The name of a Rev file containing the ancestral genome sequence. By default, the first genome in the file (with genome index 0) is used, this may be overridden by specifying *–genix g* where *g* is the desired genome index.

–inannots *gfffilename*

[Input] The name of a GFF file containing annotations of the ancestral genome. Island records should be included—while they have no effect on the simulation, the coordinates are mapped to the evolved genome and included in the output annotations that will be needed as input to the intra step. Tandem annotations will also be mapped, but it would be more natural to run a tandem finder after inter and before intra.

–model *filename*

[Input] The name of the file containing the evolutionary model parameters.

–branchlength *ticks*

[Input] The length of time to simulate in "ticks", Evolver's arbitrary unit of time. The number of ticks is a floating-point number and may be specified using any string acceptable to the *atof* function in C.

–seed *k*

[Input] A random number seed. This may be specified as an integer constant or as the string *stochastic* (the default). If an integer seed is specified, the simulation will be reproducible by giving the same command line. A stochastic seed is computed as *time*(0)**getpid*(), which won't impress your cryptographer friends but is adequate for this application. The seed is written to the log file, which is helpful, for example, when trouble-shooting a crash by enabling a second attempt at the same run. If a bug in Evolver is suspected (surely not!), then a reasonable strategy may be to attempt to avoid it by using a different seed.

–aln *revfilename*

[Output] The name of a Rev file to contain alignments of the ancestor to the evolved genome.

–outchrnames *textfilename*

[Output] The name of a text file to contain the names of the chromosomes in the evolved genome, one per line. Typically, these are the same as the ancestral chromosome, but the number and / or names of chromosomes may change due to fusion and fission events. This file is typically used by a script that starts intra-chromosome simulation following completion of the inter step. It can loop over chromosomes using something like (*bash* syntax): *for chrname in `cat chrnames.txt`; do submit_to_cluster run_intra $chrname ; done*.

–outgenome *genomename*

[Output] The name for the evolved genome. Defaults to *evolved* if not specified.

–outseq *revfilename*

[Output] The name of a file to store the evolved genome sequence.

–statsfile *filename*

[Output] The name of a file to store statistics data. This enables statistics from a complete cycle, or multiple cycles, to be consolidated into a single report. The format of this file is designed to be easily parsed by a script rather than read by a person.

–log *filename*

[Output] The name of a file for miscellaneous logging. Much of the information in this log file is also written to the stats file for later consolidation.

–targetgenecount *k*

[Input] A desired number of genes. Over the course of a simulation, the total number of genes in a genome will tend to increase due to gene duplications. There is no "deactivate gene" event to balance duplications because we felt it was unrealistically hard to develop model parameters that would achieve balance in gene number. As an alternative, we allow deletion of excess genes by the inter module. Typically the target gene count *k* will be set to the number of genes in the original ancestral genome, perhaps allowing some random fluctuation. If the number of genes $N > k$, then $N–k$ genes are deactivated (bases converted to neutral) by random selection, weighted by their mean accept probability. This is done before starting the simulation. If the *–annotsminus gfffilename* option is also given, GFF records for the deleted genes will be written to the given file.

Command-line reference for the intra module

Here is a typical command line for running the intra-chromosome simulator:

```
evo -inseq inter.outseq.rev -chrname chr20 -branchlength 0.001 -mes mes.fa        \
      -inannots chr20.intra.outannots.gff -statsfile chr20.intra.stats            \
      -outannots chr20.intra.outannots.gff -model model.txt -seed 1               \
      -aln chr20.aln.rev -outseq chr20.intra.outseq.rev -log chr20.intra.log
```

Note that here the "ancestral" sequence refers to the sequence that is input to this step; it is typically the "evolved" sequence from the point of view of a preceding inter run. The "evolved" sequence now refers to the sequence that is output by the intra step.

The following options are as for the inter module:

–inannots *gfffilename*
–outannots *gfffilename*
–branchlength *ticks*
–model *filename*
–seed *k*
–statsfile *filename*
–aln *revfilename*
–outseq *revfilename*
–outgenome *name*
–log *filename*

Other options include:

–inseq *revfilename*
[Input] The name of a Rev file containing the ancestral chromosome sequence (other chromosomes may also be present, if so they are ignored). By default, the first genome in the file (with genome index 0) is used, this may be overridden by specifying *–genix g* where *g* is the desired genome index. Typically this is the output sequence file from an inter run.

–chrname *name*
[Input] The name of the chromosome to evolve. Alternatively the chromosome id may be specified using *–chrid id*.

–mes *fastafilename*
[Input] The name of a FASTA file containing mobile element and retroposed pseudogene sequences. The insertion rates and other parameters for a sequence is specified in its annotation line, as described previously.

Mobile Element Evolution

Mobile elements (MEs) evolve, typically at a significantly faster rate than the host genome. This process can be modeled using the intra module, though some non-trivial scripting is required. We have developed a framework to implement this, which is described below. As with many aspects of Evolver, users may to use our solution, modify our scripts, or develop their own.

In outline, an ME is modeled as a short chromosome. ORFs that code for proteins are modeled using CDS annotations. "Speciation" and "extinction" of ME subtypes is modeled by a binary tree that is generated on the fly according to birth / death rates specified by parameters in a file read by the scripts. Each time intra is invoked, a few MEs are selected from the current population and designated as active, meaning that they are included in the ME library given to intra. MEs with long terminal repeats (LTRs) must be treated as special cases because intra lacks a mechanism for allowing mutations while maintaining 100% sequence identity between the repeats as required for a biologically active ME. This is handled by evolving the repeat sequence and intervening sequence ("body") as two separate "chromosomes". When the full ME sequence is required it is assembled by concatenating (repeat-body-repeat).

To perform mobile element evolution using our framework, each cycle requires the following files:

- a *mobile element configuration file*
- a *mobile element sequence file*
- a *long terminal repeat sequence file*
- a *mobile element annotation file*
- a *mobile element evolutionary model file*

The framework supports different mobile element *classes*. Each class can contain one or more *members*, one of which is *active* at any point. Each class is associated with a series of rates that describe how often members of that class get deleted from the class (or duplicated, giving birth to new elements in the class), and how often they are inserted in the genome. These are described in the configuration file, with a series of `Rate` directives. The syntax is the following:

```
Rate <ClassName> <RelInsertionRate> <AvgDelFraction>     \
 <StdDevFraction> <Pct> <MinLen> <MaxLen> <MaxCount>     \
 <DupRate> <DelRate> <LifeTime>
```

The mobile element sequence file must have FASTA entries with headers of form `ClassName.ID` or `ClassName.ID:ACTIVE`, where `ID` is an index number that distinguishes that element within the class (IDs need not be contiguous), and `:ACTIVE` distinguishes the previously active element (if any).

Some mobile element classes can be designated *LTR-like classes*. The members of these LTR-like classes must have a corresponding LTR sequence in the long terminal repeat sequence file, and when these members are used to generate the ME library, their LTR sequence is prepended and appended. These classes are distinguished in the configuration file with the `LTRClass` directive (apart from the `Rate` directive):

```
LTRClass <ClassName> <MinLTRLen> <MaxLTRLen>
```

If multiple LTR-like classes are needed, the `LTRClass` directive can appear more than once. The LTR sequence file must have FASTA entries with headers identical to the ones in the mobile element sequence file, minus the `:ACTIVE` suffix which should not appear in the annotations.

At the beginning of each cycle, within each class each ME is either duplicated, deleted, or kept as-is, in what is called a *birth-death process*. The rate at which members of a class will be duplicated or deleted is determined by `DupRate` and `DelRate` which are in units of duplications or deletions per tick. The algorithm for the birth-death process is outlined here:

```
If (DeleteRate*Branch + DupRate*Branch > 1.0)
  Normalize such that (DeleteRate*Branch + DupRate*Branch == 1.0)
For each class C
{
  For each member E of class C
  {
    With probability DeleteRate*Branch do
      If (!E.Active) Delete E;
    Else with probability DupRate*Branch do
      Duplicate E;
    }
  }
  If all members of the class have been deleted
  {
    Resurrect a random member of the class
  }
}
```

After the birth-death process, the sequences of the mobile elements (and, separately, the LTR sequences) are independently evolved. This is achieved by running intra on the ME sequence file using an annotation file describing ORFs and / or other CEs. The model used will typically be different from the model used to evolver the host genome. The simulation is performed for the branch length of the host multiplied by the `BranchLengthFactor` parameter supplied in the configuration file:

```
BranchLengthFactor <Factor>
```

This is a convenient way to arrange for ME evolution to be faster than host evolution without adjusting a large number of parameters in the model file.

After the ME are evolved, a *boundary checking process* takes place in order to filter sequences that the simulation made too long or too short. Specifically, the algorithm is the following:

```
For each class C
{
  For each member E of class C
  {
    If (E.SeqLen < C.MinLen)
      Append ((C.MinLen + C.MaxLen)/2 – E.SeqLen) random bases
    ElseIf (E.SeqLen > C.MaxLen)
    {
      If (E.HasGFFEntries)
        Delete E;
      Else
        Chop (E.SeqLen - (C.MinLen + C.MaxLen)/2) bases
    }
    If (C is LTR-Like)
    {
      If (LTR(E).SeqLen < C.MinLTRLen)
        Append ((C.MinLTRLen + C.MaxLTRLen)/2 – LTR(E).SeqLen)
      Else
        Chop (LTR(E).SeqLen - (C.MinLTRLen + C.MaxLTRLen)/2)
    }
  }
  If (C.Empty)
    Consider class C as dead
}
```

After the boundary check, each class is checked for *excessive element count*. Classes whose element count is greater than `MaxCount` are subject to random deletion of some members, so that in the end they have exactly as many as `MaxCount`. When randomly deleting elements, the active element is excluded.

During the subsequent *activation process*, an active element is selected for each class. If the class already has an active element, it is kept active with probability `1.0 - Branch*Class.LifeTime`, otherwise it is deactivated and deleted (unless it is the final remaining element in that class, and that class has no other members). If there is no previously active element, or if it just got deleted, a new one is selected randomly.

Regardless of what happens, in the end there will be one active mobile element for each class, and these elements will be exported to the mobile element library for intra.

When the sequences of active MEs are exported, the `rateperbase`, `avgdel`, `stddev` and `pct` parameters are set in each annotation line. These parameters are calculated from those provided in the configuration file in the `Rate` directive, with the `avgdel` and `stddev` being calculated by multiplying `AvgDelFraction` and `StdDevFraction` with the element's sequence length, which will include twice the LTR length where appropriate.

The value for the `rateperbase` parameter, which corresponds to the insertion rate of the mobile element, is calculated by this formula:

$$\texttt{rateperbase} = \texttt{TotalInsertRate} * \texttt{C.RelInsRate} / \Sigma_C\{\texttt{C.RelInsRate}\}$$

Where `TotalInsertRate` corresponds to the total insertion rate of a mobile element base, per chromosome base per tick, and is given in the configuration file by the following directive:

```
TotalInsertRate <TotalInsertRate>
```

This rate is multiplied by the relative insertion rate of the class (normalized by the sum of the relative insertion rates of all the classes). The library of active elements is further augmented by the retroposed pseudo-genes in the next step. Also, all the sequences of all the elements from all the classes (including the active ones) are output in a FASTA file to be used by the next cycle, along with the propagated annotations (in GFF) and LTR sequences (in FASTA).

Retroposed Pseudo-Genes

The following parameters in the configuration file affect the way retroposed pseudo-genes are handled:

```
RPGHeader <AvgDelFraction> <StdDevFraction> <Pct>
PolyATail <PolyATailLength>
MaxRPGSize <MaxRPGSize>
CountPerTick <Count>
```

During each simulation cycle, a total of `CountPerTick*Branch` approximately copies will be inserted into the genome. These are distributed among a potentially smaller set of RPG elements; this set is chosen in the following way:

```
Initialize CopiesLeft = CountPerTick*Branch
While (CopiesLeft > 0)
{
  ThisRPGCopies = PickFromGersteinCurve()
```

```
  CopiesLeft = Copiesleft – ThisRPGCopies
  Select a random gene whose spliced size is < MaxRPGSize,
    and append PolyATailLength bases of A
  Add this sequence to the ME library
}
```

When the element is added to the library, the `avgdel`, `stddev` and `pct` values are calculated using the values from the `RPGHeader` directive and the `rateperbase` value is calculated as follows:

```
rateperbase = ThisRPGCopies * RPG.SeqLen / (GenomeSize * Branch)
```

Preparing the ancestral genome

An ancestral genome is required at the start of a simulation. The sequence is required to be in a *.rev* file, annotations in GFF files are also required (unless a simulation of neutral DNA is desired).

The sequence must consist of the letters A, C, G and T only. Wildcards and spacers such as N are not permitted.

The inter and intra modules accept annotations describing constrained elements (CEs), tandem arrays and CpG islands. CEs include CDS, UTR, NXE and NGE records, as described in detail elsewhere. CE records must include a *probs* attribute, and all CEs except NGEs must also have a *gene_index* attribute. CE annotations must conform to Evolver rules, including: no overlapping genes, no NGEs inside genes, and the concatenated CDS records for a gene must begin with ATG, have an exact number of codons, end with a stop codon, and must have no in-frame stops.

The user is free to construct an ancestral sequence and annotations in any way they choose. In our work, we have developed a procedure for this with some associated tools. We use a well-annotated model organism sequence, such as human, as our starting point. Protein-coding gene annotations are extracted from the UCSC genome browser. NXE and NGE annotations are generated according to a stochastic model. Accept probabilities are assigned to CE bases, also using a stochastic model.

In outline, we prepare annotations as follows. The various *evo* options and script mentioned in the outline are described in more detail shortly.

Chromosome sequences are downloaded from the UCSC genome browser.

- The following UCSC genome browser tracks are downloaded: UCSC genes, MGC genes, old known genes, Ensemble genes, CpGs.

- The *–findns* option to *evo* is used to eliminate runs of Ns and replace other non-ACGT letters with randomly-chosen valid letters. This command outputs a GFF file documenting where blocks of Ns were removed; this GFF file is used later by the *–xgff* command to re-map annotation coordinates from the original sequence to the "N-less" version that will be used as input to the simulation.

- The *–cvtannots* option to *evo* is used to extract a valid subset of protein-coding genes.

- Non-coding conserved elements (NCCEs, meaning NGEs and NXEs) are generated using the *–genncces* option to *evo*.

- The *–assncces* assigns a subset of NCCEs to genes, converting those to NXEs and the remainder to NGEs.

- The *–assprobs* option to *evo* is used to assign accept probabilities (i.e., generate *probs* attributes) for all types of CE.

- Tandem annotations are generated using Gary Benson's Tandem Repeat Finder.

The *findns* command

The *–findns* command removes non-ACGT letters from a sequence. Invalid letters are replaced by randomly-chosen ACGT letters (chosen with uniform probability). Optionally, consecutive runs of Ns can also be excised, producing a shorter sequence. In the latter case, a GFF file is produced to document which regions were deleted; this can be used by the *–xgff* command to adjust annotation coordinates to correspond to the new sequence. Typical usage is as follows.

```
evo -findns seq.fasta -out_randns randns.fasta -out_x x.fasta -out_gff ns.gff   \
  -label_randns label1 -label_x label2 -minns 32 -log findns.log
```

–findns *fastafilename*
: [Input] The name of a FASTA file containing one or more sequences to be processed.

–min_ns *k*
: [Input] The minimum number of consecutive Ns to be deleted (excised) rather than being replaced by random letters. Default is 32.

–out_randns *fastafilename*
: [Output] The name of a FASTA file to write the sequence with invalid letters, including Ns, replaced by random letters.

–out_x *fastafilename*
: [Output] The name of a FASTA file to write the sequence with invalid letters replaced by random letters and runs of Ns excised.

–out_gff *gfffilename*
: [Output] The name of a GFF file to write the input coordinates of runs of Ns that were deleted. The GFF feature is *nblock*. This is used as input by the *–xgff* command.

–out_randns *fastafilename*
: [Output] The name of a FASTA file to write the sequence with invalid letters, including Ns, replaced by random letters.

–label_randns *label*
: [Output] The FASTA label to use in the *–outrandns* file. By default, the label is the input label with ".randns" appended. Note that if there are two or more sequence in the input file, they will all get the same label, so this option should only be used with a single sequence in the input file.

–label_x *label*
: [Output] As for *–label_randns* for the *–out_x* file.

The *xgff* command

The *–xgff* command adjusts coordinates in a GFF file to match a sequence from which Ns have been removed by *–findns*.

```
evo -xgff annots.gff -gff_ns ns.gff -out annots.xns.gff -log xgff.log
```

–xgff *gfffilename*

[Input] The name of a GFF file containing annotation records for a sequence before Ns were excised.

–gff_ns *gfffilename*

[Input] The name of the GFF file documenting blocks of Ns; output by the *–outgff* option to *–findns*.

–out *gfffilename*

[Output] The name of a GFF file to write the adjusted records.

The *cvtannots* command

The –*cvtannots* command extracts a valid subset from a set of candidate annotations, eliminating overlapping genes, genes with invalid CDSs, etc. Typical usage is as follows.

```
evo -cvtannots inannots.gff -out outannots.gff -chrname chr -seq seq.rev   \
  -log cvtannots.log
```

–inannots *gfffilename*

[Input] The name of a GFF file containing candidate annotation records.

–seq *revfilename*

[Input] The name of a Rev file containing the sequence. By default the first genome (i.e., genome with id 0) is assumed; this can be overridden by specifying –*genix id*.

–chrname *name*

[Input] The name of the chromosome. By default, the first chromosome in the genome is used.

–outannots *gfffilename*

[Output] The name of a GFF file to write output records.

The *genncces* command

The –*genncces* command generates a set of non-coding conserved elements (NCCEs) according to a stochastic model. Here, "non-coding" means NXEs and NGEs; UTRs are not included. A better term would perhaps be "non-exon", but this could be confused with NXEs=non-exon elements that do belong to a gene, and we are de facto stuck with the existing terminology.

The –*genncces* command does not classify NCCEs as NXEs and NGEs; this can be done later by the –*assncces* command. The primary output from the command is a GFF file with records having a feature type of *ncce*. They have no *probs* or *gene_index* attributes; these can be assigned later using the –*assncces* and –*assprobs* commands. The output thus requires significant further processing before it can be used as input to the inter or intra modules.

Records are generated with a length distribution specified in exactly the same way as for events such as *Delete*. Coordinates are randomly distributed throughout the genome, except that they may not overlap regions given in an "exclude" file which will typically contain CDS and UTR records for protein-coding genes. Records are generated until a given fraction of the genome is covered by NCCEs; this fraction is given by the *GenomeNCCEPct* parameter in a model file. We typically use a value of 10 with the result that one tenth of the genome is covered by NCCEs.

Typical usage is:

```
evo -genncces -excl_gff annots.gff -out ncces.gff -length 100000000   \
  -model model.txt -log genncces.log
```

–excl_gff *gfffilename*

[Input] The name of a GFF file containing annotation records. Regions in this file are excluded from NCCE generation; i.e., no output record may overlap any record in this file. At least one record must be present as the sequence label to use for output is taken from this file. There must be exactly one chromosome present; multiple sequences are not supported.

–model *modelfile*

[Input] Model parameter file. The length distribution is read from the *NCCELengthDist* distribution and the fraction of the genome to cover from the *GenomeNCCEPct* parameter.

–length *k*

[Input] The length of the chromosome in bases. Only records for a single chromosome can be generated at one time.

–out *gfffilename*

[Output] The name of a GFF file to write the generated NCCE records. The sequence label to use for the output records is the one used in the *–excl_gff* file.

The *assncces* command

The –*assncces* command converts NCCE records into NXE and NGE records, thus assigning some to genes and designating the rest as non-gene.

Genes are identified from a GFF file containing records with *gene_index* attributes. This gives the initial start and end coordinates of each gene. Typically the gene file will contain CDS and UTR records and the initial start-end coordinates will therefore specify the transcribed part of the gene. Genes are extended to include non-transcribed regions. This process is controlled by three parameters: *MeanInterGeneFract*, *StdDevInterGeneFract* and *MaxGeneMargin*. Typical values we use are *MeanInterGeneFract* = 0.2, *StdDevInterGeneFract* = 0.2 and *MaxGeneMargin* = 200000. A region extending from the last transcribed base in one gene to the first transcribed base in the following gene is called an *inter-gene* region. Each gene has two *margins*, one from the beginning of the gene to the first transcribed base, the other from the last transcribed base to the end of the gene. Genes are extended so that, on average, a fraction *MeanInterGeneFract* of bases are converted to margins. The variation in this fraction for individual genes is controlled by the *StdDevInterGeneFract* parameter, which is the standard deviation of a truncated normal distribution with mean *MeanInterGeneFract*. NCCE records found within margins become NXEs, the remainder become NGEs.

Typical usage is:

```
evo -assncces ncce.gff -genes genes.gff -out nxenge.gff      \
  -model model.txt -log assncces.gff
```

–inannots *gfffilename*
: [Input] The name of a GFF file containing candidate annotation records.

–genes *gfffilename*
: [Input] The name of a GFF file containing records with *gene_index* attributes.

–model *modelfilename*
: [Input] The name of the model parameter file.

–out *gfffilename*
: [Input] The name of a GFF file to write the NXE and NGE records.

The *assprobs* command

The –*assprobs* command assigns accept probabilities to conserved element (CE) records in a GFF file.

The following parameters, specified in the model file, control how probabilities are generated. Typical values are shown.

```
# Mean and std. deviation of mean probabilities generated for non-coding CEs.
NCCEMeanMean = 0.8
NCCEMeanStdDev = 0.2

# Mean and std. deviation of mean probabilities generated for CDSs.
CDSMeanMean = 0.5
CDSMeanStdDev = 0.4

# Percentage of probabilties to be shuffled when clumping.
ClumpShufflePct = 25

# Mean and std. deviation of CE clump length (~= correlation distance).
CEClumpMeanLength = 32
CEClumpLengthStdDev = 16

# Accel factor to convert CDSMeanMean to gene NCE mean.
GeneCDSToNCEAccel = 3.5
```

Probabilities are sampled from truncated normal distributions:

http://en.wikipedia.org/wiki/Truncated_normal_distribution

A truncated distribution is required since valid probabilities have values from zero to one, but standard probability density functions (PDFs) have domains including all real values ($-\infty$ to $+\infty$). A truncated normal distribution $T(x; a, b, \mu', \sigma)$ is derived from a normal (Gaussian) distribution $N(x; \mu, \sigma)$ with mean μ and standard deviation σ by excluding values $x<a$ and $x>b$. Note that in our convention, μ' is the true mean of T, which is not in general the same as μ, the mean of normal distribution N from which it is derived. However, the "standard deviation" σ of T is by definition the standard deviation of its N. When probabilities are being generated, we always use $a=0$ and $b=1$. The process of generating probabilities is described in more detail under Clumping.

Typical usage is as follows.

```
evo -assprobs inannots.gff -out outannots.gff -model model.txt -log assprobs.log
```

–inannots *gfffilename*
[Input] The name of a GFF file annotation records.

–model *modelfilename*
[Input] The name of a model parameter file.

–outannots *gfffilename*
[Output] The name of a GFF file to write output records.

Clumping

Base accept probabilities are generated using a *clumping* method designed to fulfill the following requirements.

- Base accept probabilities (BAPs) should be correlated within a CE, e.g. a BAP lower than the mean should tend to follow another low BAP.

- There should be a known distribution over gene mean accept probabilities (GMAPs) with a pre-defined mean and standard deviation.

- Mean accept probabilities (MAPs) for CEs within a gene should be "spread" so that there is a reasonable chance that the mean for a CE will be significantly different from the mean for its gene.

Informal introduction

The idea is to take a set of probabilities and divide it into subsets so that the subsets (clumps) will tend to have a spread of means. This is done by 1. sorting, 2. partitioning into disjoint ranges according to clump length, then 3. introducing outliers by shuffling.

In the case of a gene, a clump will be a CE.

To introduce correlations within a CE, we divide the CE into randomly chosen subsequences. Each subsequence is then a clump and a CE built in this way will tend to have contiguous regions of probabilities higher and lower than its mean.

The clumping algorithm

Inputs to clumping are:

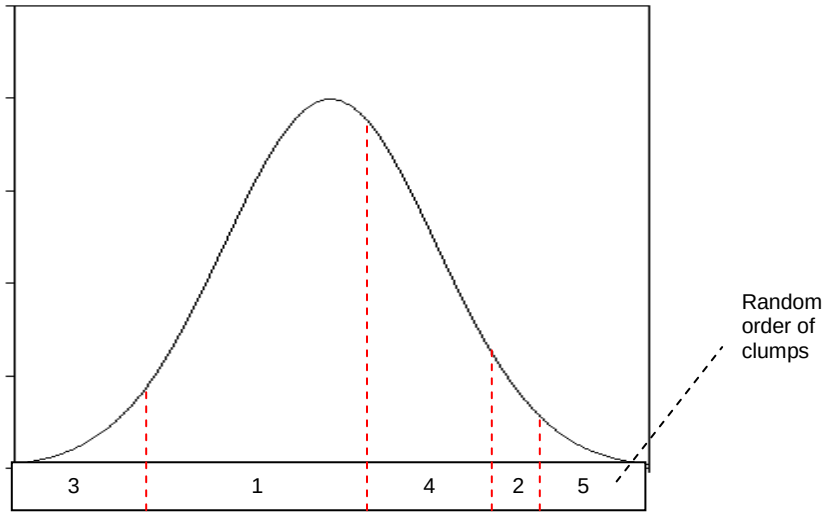
A set of lengths L_i, $i = 1 .. N$.

A set of probabilities P_i, $i = 1 .. L$, where $L = \sum L_i$.

Each length represents one *clump*. There are N clumps with a total of L positions. Each position i has a probability P_i. The output is a set of probability vectors C_i, one for each clump i, such that the means of each vector tend to differ from the overall mean:

C_{ij}, $i = 1 .. N$, $j = 1 .. L_i$.

The algorithm proceeds as follows. The probabilities P_i are sorted. A random order is chosen for the clumps, and probabilities are assigned to clumps in increasing order of probability and random order of clump. A clump is filled before proceeding to the next clump. At this point, the ranges of probabilities found in clumps are disjoint.



We suspect that creating disjoint subsets in this way maximizes the spread of clump means, but have not attempted to prove it. This is desirable, but we also want to allow some chance of outlier values in each clump. Spreading means and allowing outliers are conflicting goals. A compromise is implemented via the next step: shuffling.

To shuffle, pairs of values are selected at random and exchanged between clumps. The more this is done, the more outliers will be introduced and the more the mean of each clump will tend to the overall mean. This amount of shuffling can be a user-settable parameter; by default 25% of bases participate in a shuffle ($L/8$ swaps). Based on a few experiments, our sense is that this is qualitatively enough to introduce outliers without losing the partitioning.

At this stage, each clump is approximately sorted in order of increasing probability—it was exactly sorted when created, exceptions are introduced by shuffling. As a final step, the vector of probabilities for each clump is therefore randomized.

Assigning BAPs to a CE of length *L*

The motivation for clumping an individual CE is to produce correlations between BAPs.

1. *L* probabilities are generated either according to some PDF (in the case of an NGE) or obtained from the output from a previous clumping step (in the case of a gene CE).

2. *L* is partitioned into some small number of clumps of random lengths.

3. The probabilities are clumped.

4. Probabilities are assigned to bases in increasing genome order by first using all probabilities from the first clump then moving to the next.

Assigning BAPs to CEs in a gene

1. *L* probabilities are generated according to some PDF, where *L* is the total length of all CEs in the gene.

2. Each CE is considered one clump.

3. Clumping is performed using the given probabilities and CE lengths. This yields a set of probabilities for each CE.

4. Those probabilities are assigned to each CE using a second round of clumping as described in the previous section.

Substitution rates

Evolver assigns separate rates to each possible substitution rate: A→T, C→A and so on. Strand symmetry is assumed, so C→T has the same rate as A→G etc. This gives a total of six independent rates, specified by the following model parameters:

```
AC_Rate = 0.2
AG_Rate = 0.6
AT_Rate = 0.2
CA_Rate = 0.2
CG_Rate = 0.2
CT_Rate = 0.6
```

Unmethylated regions and CpG islands

The rate parameters above are for unmethylated regions which do not have accelerated C→T rates in CpG dinucleotides. Methylated regions (most of the genome in the case of mammals) have elevated rates of C→T substitution in CpG dinucleotides. The increase in rate is specified by this parameter:

```
CpG_C_to_T_Ratio = 10.0
```

CpG sweeps

For efficiency, accelerated C→T substitution in CpG dinucleotides are implemented using *CpG sweeps*. A few times during an intra run the chromosome is scanned in coordinate order searching for CpGs outside CpG islands. Each time one is found, a C→T substitution is made with the appropriate probability that will result in the rate implied by *CpG_C_to_T_Ratio*.

Sweeps must be done often enough that only a small fraction of CpGs undergo a mutation to TpG. (Denote this fraction by f). This is because substitution does not commute with duplication: you can estimate the age of a duplication by looking at the number of substitutions between the two copies.

The input parameter is q=*CpG_C_to_T_Ratio*, which is defined as:

```
q = (C→T rate outside CpG islands) / (C→T rate inside).
```

We expect a typical q to be around 10. The C→T rate in CpG islands is:

```
s = 1 * P(C→T | C substitutes) * (1 - CGRejectProb)
```

With typical values, s = 0.6*0.76 = 0.46. The rate outside an island is then Q = qs, with typical value 10*0.46 = 4.6.

A rate of s is produced by "standard" substitutions, so the added rate to be implemented by CpG sweeps is:

```
R = Q - s
```

with typical value R = 4.6 – 0.46 = 4.1.

In one sweep representing a duration of time t, the probability P of a given C→T substitution is then Rt, providing t << R.

The sweep duration is then determined by f = P = Rt, hence:

```
t = f/R
```

We expect R ~ 5, and choose f = 0.1 as a reasonable default; this gives t = 0.02.

If the branch length b <~ t, then this will result in only a single sweep and we are back to the problem with duplications. We therefore set a minimum number m of sweeps, say 10, to ensure a detectable age distribution over duplications. Then:

```
t = min(f/R, b/m)
```

The additional parameters used in implementing this scheme are:

```
CpGFraction = 0.1
MinCpGSweeps = 10
```

Unmethylated genomes

To specify that the genome is unmethylated, such as for *Drosophila*, the *MinCpGSweeps* parameter is set to zero:

```
MinCpGSweeps = 0
```

Stationary composition state

In our simulations we aim to keep the nucleotide composition of the genome approximately constant. We now address the question of how to achieve this given the Evolver model that specifies substition and CpG rates.

Let N_x be the number of letters of type x, $x = A, C, G$ or T.

Let r_{xy} $x \neq y$ be the substitution rate for $x \rightarrow y$, i.e. the probability per unit time that a given letter x becomes a different letter y.

Let $S(x)$ be the complementary base, e.g. $S(A) = T$.

By strand symmetry,

$$N_A = N_T \quad (1)$$
$$N_C = N_G \quad (2)$$
$$r_{xy} = r_{S(x)S(y)} \quad (3)$$

By (3) above, there are 6 independent rate parameters:

	A	C	G	T
A		r_{AC}	r_{AG}	r_{AT}
C	r_{CA}		r_{CG}	r_{CT}
G	$r_{GA}=r_{CT}$	$r_{GC}=r_{CG}$		$r_{GT}=r_{CA}$
T	$r_{TA}=r_{AT}$	$r_{TC}=r_{AG}$	$r_{TG}=r_{AC}$	

Over a length of time t small enough that multiple substitutions at a single site can be ignored, the net change in number of *A*s is:

$\Delta A =$ (number of changes to A) – (number of changes from A)

$$= N_C r_{CA}\, t + N_G r_{GA}\, t + N_T r_{TA}\, t - N_A\, (r_{AC} + r_{AG} + r_{AT})\, t.$$

Dividing both sides by t and using eqs. (1) – (3), the net change in number of *A*s per unit time is:

$$\Delta A/t = N_C r_{CA} + N_C r_{CT} + N_A r_{AT} - N_A\, (r_{AC} + r_{AG} + r_{AT})$$

$$= N_C\, (r_{CA} + r_{CT}) - N_A\, (r_{AC} + r_{AG}).$$

Stationarity requires $\Delta A/t = 0$, which gives:

$$N_C\, (r_{CA} + r_{CT}) = N_A\, (r_{AC} + r_{AG}). \qquad (4)$$

Similarly,

$$\Delta C/t = N_A r_{AC} + N_G r_{GC} + N_T r_{TC} - N_C\, (r_{CA} + r_{CG} + r_{CT})$$

$$= N_A r_{AC} + N_C r_{CG} + N_A r_{AG} - N_C\, (r_{CA} + r_{CG} + r_{CT})$$

$$= N_A r_{AC} + N_C r_{CG} + N_A r_{AG} - N_C\, (r_{CA} + r_{CG} + r_{CT}).$$

and $\Delta C/t = 0$ gives:

$$N_A\, (r_{AC} + r_{AG}) = N_C\, (r_{CA} + r_{CT}),$$

which is identical to (4).

Re-writing (4) in terms of the composition ratio,

$$(r_{CA} + r_{CT}) / (r_{AC} + r_{AG}) = N_A / N_C. \qquad (5)$$

Define p_{xy} to be the probability that a base of type x mutates to a different type y, given that it undergoes a single substitution; this might be written $p_{xy} = \mathrm{P}(x \rightarrow y \mid x \text{ substitutes})$. Then $r_{xy} = R_x\, p_{xy}$ where R_x is the total substitution rate for x, i.e. $R_x = \sum_y r_{xy}$. Then we can re-write (5) as:

$$R_C\,(p_{CA} + p_{CT}) / R_A\,(p_{AC} + p_{AG}) = N_A / N_C. \quad (6)$$

Hence, given the rate R_A, composition and conditional probabilities p_{xy}:

$$R_C = R_A\,(N_A / N_C)\,(p_{AC} + p_{AG}) / (p_{CA} + p_{CT}). \quad (7)$$

Alternatively we can use $p_{AG} = p_{TC}$ by eq. (3), then,

$$R_C = R_A\,(N_A / N_C)\,(p_{AC} + p_{TC}) / (p_{CA} + p_{CT}). \quad (8)$$

We could additionally assume that:

$$p_{AC} = p_{CA} \text{ and } p_{TC} = p_{CT} \quad (9)$$

Then (8) becomes:

$$R_C = R_A\,N_A / N_C. \quad (10)$$

However, (9) is only approximately true.

Accounting for CpG Effects

We will call a *C* that is in a CpG dinucleotide and not in a CpG island an "accelerated" *C* due to its elevated $C \rightarrow T$ rate compared with other *C*s. The CpG acceleration factor q gives the relative $C \rightarrow T$ rate for accelerated *C*s vs. other *C*s. Typically we expect $q \approx 10$. We will assume that p_{CT} = P($C \rightarrow T$ | *C* substitutes) does *not* include CpG effects, i.e. it is more properly described as:

P($C \rightarrow T$ | *C* substitutes and *C* is not accelerated).

Similarly of course for the p_{GA}, which we assume to be equal to p_{CT} by strand symmetry.

We use a prime (') to indicate rates that include CpG effects. Hence,

R_A = rate of $A \rightarrow x$ for all x != *A*.
R_C = rate of non-accelerated $C \rightarrow x$ for all x != *C*.
R'_C = rate of $C \rightarrow x$ for all x != *C*, averaged over all *C*s, including accelerated.
r_{CT} = rate of non-accelerated $C \rightarrow x$.
r'_{CT} = rate of $C \rightarrow x$, averaged over all *C*s, including accelerated.

Define f_A = fraction of genome that is *A* or *T*, f_C = fraction of genome that is *C* or *G*. Stationary composition requires that the total rate of *A*s changing to *C*s and *G*s, averaged over all bases in the genome, is equal to the total rate of *C*s changing to *A*s and *T*s,

$$f_A\,(r_{AC} + r_{AG}) = f_C\,(r_{CA} + r'_{CT}). \quad (11)$$

Using $r_{xy} = R_x p_{xy}$, and defining F to be the fraction of Cs that are accelerated and γ to be the increase in average rate $C \rightarrow T$ when CpG effects are included, i.e.:

$$\gamma = 1 + F\,(q - 1). \tag{12}$$

Note that the increase $C \rightarrow T$ rate for accelerated Cs is $q - 1$, not q, because accelerated Cs are also subject to "normal" $C \rightarrow T$ substitutions. Now, using $r_{AC} = R_A p_{AC}$ etc., (11) can be re-written:

$$f_A R_A (p_{AC} + p_{AG}) = f_C R_C (p_{CA} + \gamma\, p_{CT}). \tag{13}$$

Setting the tick to be one substitution per site

Now we introduce the additional requirement that the average substitution rate is one,

$$f_A R_A + f_C R'_C = 1. \tag{14}$$

Solving for R_A and R_C,

$$R_A = (1 - f_C R'_C) / f_A, \tag{15}$$

$$R'_C = (1 - f_A R_A) / f_C. \tag{16}$$

We can relate R_C and R'_C as follows:

$$R'_C = R_C + F\,(q - 1)\, r_{CT} \tag{17}$$

$$= R_C\,(p_{CA} + p_{CG} + \gamma\, p_{CT}) \tag{18}$$

Define $\beta = (p_{CA} + p_{CG} + \gamma\, p_{CT})$, then

$$R'_C = \beta\, R_C. \tag{19}$$

Substituting (19) and (15) into (13) and solving for R_C,

$$R_C = w / (f_C\,(v + \beta\, w)). \tag{20}$$

Where $v = p_{CA} + \gamma\, p_{CT}$ and $w = p_{AC} + p_{AG}$. We can then get R_A from (15) and the individual rates using $r_{xy} = R_x p_{xy}$.

The *evo_subst_rates.py* script

The *evo_subst_rates.py* script computes substitution rates that should, by the above calculations, give stationary composition and a (neutral, unmethylated) substitution rate of 1 per site per tick. Inputs are the numbers of A/T and C/G bases in the genome, six independent substitution probabilities, F (the fraction of Cs that are accelerated) and q (*CpG_C_to_T_Ratio*). These input parameters are hard-coded into the source and are adjusted by editing the script:

```
############################
# INPUT PARAMETERS
# Edit these as desired.
############################

N_AT = 1689122543  # for hg18
N_CG = 1168890263  # for hg18

f_CpG = 0.048      # for hg18
q = 10.0

p_AC = 0.2
p_AG = 0.6
p_AT = 0.2

p_CA = 0.2
p_CG = 0.2
p_CT = 0.6
```

The script writes the computed rate parameters to standard output in a format acceptable in a model parameter file:

```
#  1.4320  gamma = (C->T rate including CpGs) / (rate without CpGs)
#  1.2592  beta = total C->x rate, scaled to non-CpG C rate = 1.0
#  0.8672  R_A = sum of rates Ax_Rate
#  0.9465  R_C = sum of rates Cx_Rate (excludes CpGs)
#  1.1919  R_C_CpG = avg rate C->x over all bases including CpG
#  1.0000  Subs/site/tick (should be 1.0)
#  8.3778% AT fail percent
#
# For composition balance these should be equal:
#         0.4100  Rate of A or T -> C or G per site
#         0.4100  Rate of C or G -> A or T per site
#
# Probs (should match input values):
#        AC=0.2000  AG=0.6000  AT=0.2000
#        CA=0.2000  CG=0.2000  CT=0.6000

AC_Rate =  0.1734
AG_Rate =  0.5203
AT_Rate =  0.1734

CA_Rate =  0.1893
CG_Rate =  0.1893
CT_Rate =  0.5679
```

Python scripts

The following python scripts are provided. Input file names are specified as command-line arguments, output is written to standard output.

`gff.py`
: A module used by scripts that manipulate GFF records.

`trf2gff.py`
: Convert a Tandem Repeat Finder *.out* file into a GFF file for Evolver.

`gff_featurestats.py`
: Report a table derived from an annotation GFF file showing the number of records of each feature type and how many bases they cover.

`gff_featurestats2.py`
: Similar to *gff_featurestats.py*, but compares two annotation files, e.g. ancestral and evolved or two different genomes evolved from the same ancestor. To include exon and intron statistics the *gff_cdsutr2exons.py* and *gff_exons2introns.py* scripts can be used.

`compost.py`
: Report nucleotide and dinucleotide composition statistics for a FASTA file containing one or more sequences.

`compost2.py`
: Similar to *compost.py*, but compares two FASTA files.

`gff_cdsutr2exons.py`
: Input is a GFF file containing CDS and UTR records. Output is a GFF file containing records of type *exon*. There is typically a one-to-one correspondence between CDS-exon and UTR-exon, but there are exceptions where a single exon has adjacent UTR+CDS (5') or CDS+UTR (3').

`gff_exons2introns.py`
: Input is a GFF file containing exon records, which must have the *gene_index* attribute. Output is a GFF file containing records of type *intron*.

`merge_evostats.py`
: Input is one or more statistics files produced by the *–stats* option to inter or intra. Output is a single statistics file in the same format produced by summing over records of each type. Used to consolidate statistics from multiple runs prior to producing a report using *evostats_report.py*.

`evostats_report.py`
: Input is one statistics file, usually produced by *merge_evostats.py*. Output is a human-readable report.

The *probstats* command

The *–probstats* command generates statistics on accept probabilities found in a GFF annotation file. Typical usage is:

```
evo -probstats annots.gff -log probstats.log
```

The log file contains a report with probability distributions for genes and for individual conserved element types. For example:

```
NGE:
Mean 0.787, std.dev 0.163
 0.0000 -  0.1000         331  *
 0.1000 -  0.2000           9
 0.2000 -  0.3000          54
 0.3000 -  0.4000         226
 0.4000 -  0.5000         688  **
 0.5000 -  0.6000        1787  *******
 0.6000 -  0.7000        3675  ***************
 0.7000 -  0.8000        5986  ************************
 0.8000 -  0.9000        7708  *******************************
 0.9000 -  1.0000        7648  ******************************
```

The histogram shows the number of records of the given type (in this case, NGE) having a mean accept probability falling within each bin. A final table summarizes the mean and standard deviation for each record type:

Type	Nr	Mean	StdDev	1	2	4	8	16	32	64	128
Gene	129	0.7323	0.1771								
CDS	1784	0.4984	0.2949	0.30	0.29	0.25	0.19	0.09	-0.01	-0.03	0.00
GeneNCE	34743	0.8076	0.2083	0.04	0.03	0.02	0.01	-0.01	-0.02	-0.01	-0.01
NGE	28112	0.7865	0.1628	0.10	0.09	0.06	0.03	-0.01	-0.04	-0.04	-0.02

The columns 1, 2 ... 128 give the correlation coefficient between bases at the given distance within a single record. A correlation coefficient significantly greater than zero indicates clustering of probabilities within a record, which is the desired effect of "clumping".

The *alnstats* command

The –*alnstats* command generates statistics derived from alignments and annotations of a pair of genomes.

```
evo -probstats annots.gff -log probstats.log
```

The log file contains a report with probability distributions for genes and for individual conserved element types.