# Selenoprofiles 3 manual

## Pipeline for profile-based protein finding in genomes


contact: marco.mariotti@crg.eu

Last update: March 6th 2015
selenoprofiles version 3.2

# Table of contents

If this program is useful to your research, please cite:

Mariotti M, Guigó R (2010)
**Selenoprofiles**: profile-based scanning of eukaryotic genome sequences for selenoprotein genes.
Bioinformatics. 2010 Nov 1;26(21):2656-63. Epub 2010 Sep 21

Cover image created with Wordle using the text of this manual - http://www.wordle.net/

# Introduction

Selenoprofiles is a pipeline for profile-based protein prediction in genomes.
The program takes two inputs per run:

- one or more profile alignments, representing the protein families to search for,

- a genome (or any other nucleotide database), the target you want to scan.

Selenoprofiles runs internally a number of "slave" programs, whose predictions are analyzed and combined. The main programs used are: blast (psitblastn flavor, from blastall NCBI package), exonerate (utilized in protein-to-genome mode) and genewise. All these programs, although different in the algorithm and in speed, are based on the same principle: the target (nucleotide) is translated in all possible frames, and the query (protein) is aligned to such translated sequences, searching for high-scoring matches. The procedures of exonerate and genewise include also the prediction of splice sites, to bridge the matches into more complete, multi-exonic gene predictions.
Selenoprofiles use blast as first step, and attempts to refine its predictions with exonerate and genewise. It then processes the candidate gene structures, finally producing non-overlapping gene predictions for all input profiles.

The main purpose of selenoprofiles is the accurate search of a set protein families in a wide range of sequenced species. Nonetheless, it has been used also for the complete annotation of genomes. In this case a comprehensive, large set of input profiles has to be provided. A virtue of selenoprofiles is flexibility: its workflow can be substantially modified using options and configuration files, allowing in particular a finely tuned filtering of results. Also, the user can also easily plug-in its own code for specific annotations, analysis, or modifications to gene structures. Finally, the selenoprofiles package includes a few additional programs to collect and visualize the results of searches along the phylogenetic tree of target species.

Selenoprofiles can be used with any input protein family, but it was initially developed for selenoproteins. These peculiar proteins contain selenocysteine, the 21st amino acid. Selenocysteine (Sec, or U) is inserted in correspondence to specific UGA codons, which normally signal translation termination. In selenoprotein transcripts we find specific secondary structures (SECIS elements), which targets a specific UGA to be read as Sec instead that as a stop. Since selenoproteins possess this peculiar feature (recoding of specific stop codons), normal gene prediction programs fail to predict them. Selenoprofiles in contrast is able to correctly include selenocysteine positions, by using technical expedients detailed in this manual. The key concept is that selenocysteine positions in the proteins (alignment columns) are known a priori. Selenoprofiles includes built-in profiles for selenoproteins and other proteins related to selenocysteine, allowing out-of-the-box prediction of these families.

This manual describes the selenoprofiles pipeline starting from the simplest usage, moving then to most complex customization methods. It covers almost the totality of selenoprofiles options. The full list can be inspected running the command *Selenoprofiles --help full.*
The pipeline is also described in a paper in Bioinformatics (see references on Appendix 3), in which we also detail how we validated the method. Note that the paper refers to the version 1, while here we describe version 3, with several major improvements.

# Installation

Selenoprofiles can be installed on any unix system with python 2.6 or newer. A python command line installer (*install_selenoprofiles.py*) is provided inside the installation package that you can find at http://big.crg.cat/services/selenoprofiles. The user needs to take care of the installation of all slave programs: NCBI blast package 2.2.18[1], exonerate version 2.0.0 or newer, genewise from the Wise2 package, and also mafft. Gawk is also needed. These executables have to be available in the bash environment for the installer to work. Find useful links for their installation in Appendix 3. If you experience any problem with their installation, visit Appendix 4, troubleshooting.

Selenoprofiles provides a wide range of filtering functions. The standard protocol is to measure the similarity of the candidate protein sequence with the profile sequences, and compare it with the distribution of similarity among the profile sequences. This method (AWSI, described later) performs very well for most purposes. However, selenoprofiles offers more sophisticated methods, some of which scan a protein database (NCBI nr) to search the candidate sequences with blastp, and parse results to infer the goodness of the prediction. Since some of the built-in profiles for selenoproteins and Sec machinery utilize this kind of filtering, the database is needed for their use. The blast nr database is very large (>8 Gb uncompressed). It may take a long time to download it, and may take excessive disk space (>30 Gb, including the formatted files).
If you plan to use the program to search for your custom families, and you do not need to use the built-in profiles (for selenoproteins and Sec related proteins), then you may avoid using NCBI nr, and perform a minimal installation (*python install_selenoprofiles.py -min*).
If instead you require the built-in profiles or you plan to use the advanced filtering procedures, you will need a complete installation (*python install_selenoprofiles.py -full*). If you already have NCBI nr on your system, you can link it using installer option *-nrdb* (see *install_selenoprofiles.py —help*). For selenoproteins, the program SECISearch3 may also be useful; this is available on request.

After installation, you can test selenoprofiles using script *test_selenoprofiles.py*, located inside the installation directory. This script runs the pipeline on a few test sequences and checks that the output is as expected. You can also run anytime *selenoprofiles -test* to perform a check of all slave programs and modules used either by selenoprofiles, or by the additional programs included for visualization.
In particular, *selenoprofiles_build_profile.py* requires Pylab (http://www.scipy.org/PyLab) to plot the sequence identity characteristics of profiles, and *selenoprofiles_tree_drawer.py* requires ete2 (http://ete.cgenomics.org/) for tree-based visualization of results across species. Although none of this two modules is compulsory, we strongly suggest to install ete2 for projects aimed at searching certain protein families in a wide range of species, to conveniently visualize results as an annotated species tree.

[1] All 2.2.x versions are expected to work. The newer versions, called blast+, will not work

# Getting started

This chapter will cover the basic use of selenoprofiles. To begin, we will use a profile alignment included in selenoprofiles package. Let's get practical. Let's say that we want to scan the genome of the species *Macaca mulatta*, contained in the file */db/genome.fasta*, for the built-in AhpC profile.
Here's a basic command line:

```
Selenoprofiles results_folder  -t /db/genome.fasta  -s “Macaca mulatta" -p AhpC
```

The first argument of selenoprofiles is the folder where all results will be stored. If not existing, it will be created. It will be called **results folder** from now on.
The second argument, provided with option *-t*, is the **target file**. A multi-fasta file must be provided. This is formatted with formatdb and fastaindex to be used by the slave programs. The file name, without the extension, is used for naming in selenoprofiles and will be referenced as the target name (in the example, *genome*). Each short title (defined as the first word in a fasta header) must be unique, and no empty sequence should be present. The option **species** (or *-s*) allows to specify to which organism the genome belongs to. The definition of the species is highly recommended but not compulsory: if none is specified, the species will be set to *unidentified*. Note that the combination of species name and target name must be unique in a given results folder.
The other key argument to the program is the **profile**, or the profiles, that will be searched in the genome. If none is specified, the list of profiles is read from the configuration file, which defaults to the selenoproteins and Sec machinery families. The option -profile (or *-p* or *-P*) can accept multiple arguments, that must be comma separated with no space within. Each such argument can be the name of profile (which is searched into the profiles folder), the path to a profile fasta alignment, or a keyword indicating a list of families defined in the main configuration file. When a family alignment is provided for the first time to selenoprofiles, its profile is built on the fly (see building a profile).
For example, to scan the same genome with two custom profiles alignments you can use:

```
Selenoprofiles results_folder  -t /db/genome.fasta  -s "Macaca_mulatta" \
        -p /somewhere/profiles/family1.fa,/somewhere/profiles/family2.fa
```

Or alternatively, defining the profiles folder in the command line:

```
Selenoprofiles results_folder  -t /db/genome.fasta  -s "Macaca_mulatta"
        -profiles_folder /somewhere/profiles/ -p family1,family2
```

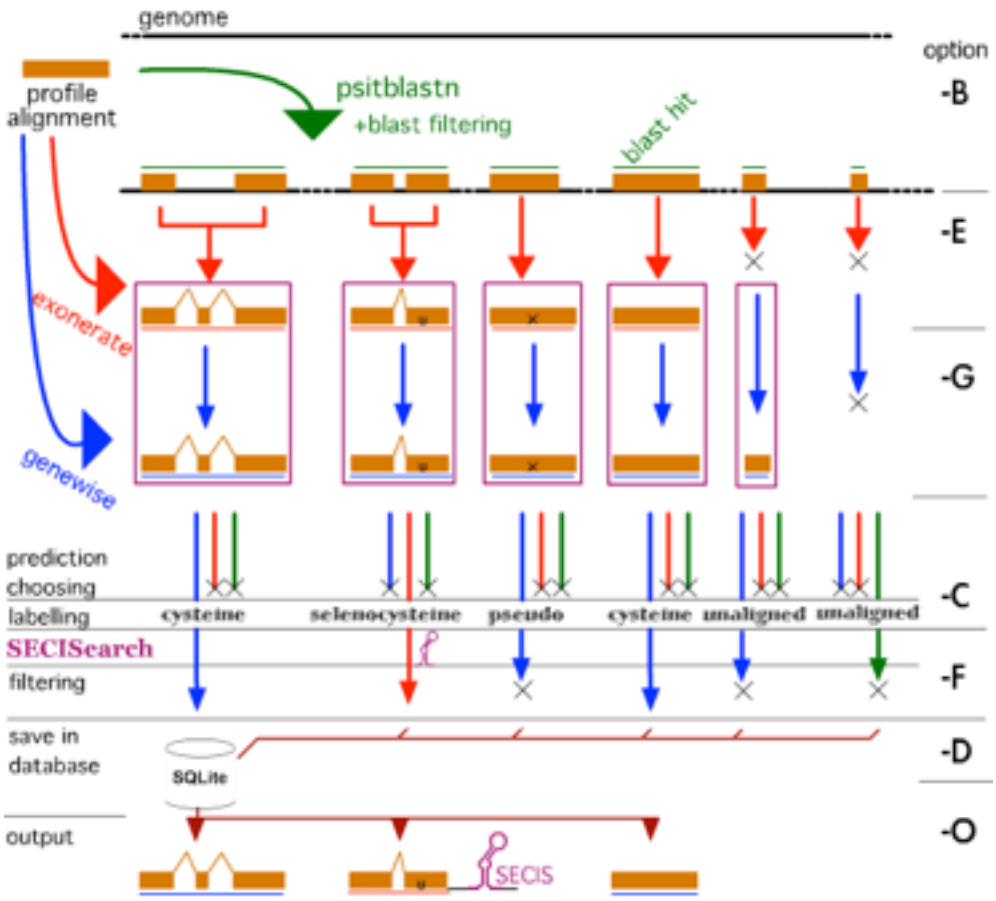
By default, selenoprofiles executes the full pipeline. The final output files will be found inside the results folder, inside the target subfolder, in a folder called output. For the example above, this folder would be:

```
results_folder/Macaca_mulatta.genome/output/
```

## The pipeline in summary

The pipeline workflow is detailed in the next section, and it is here summarized (see also figure below). The program psitblastn is used with a PSSM derived from the profile alignment to identify matches in the target genome. These matches are then used, through

the two splice alignment programs exonerate and genewise, to deduce the exonic structure of the candidate genes. The predictions of these three programs are analyzed to choose one, which is then labelled through a dedicated procedure. The predictions are sometimes improved by a few modification procedures. Through the entire pipeline a number of steps are performed to filter out likely false positives and to keep the number of potential candidates under manageable levels. There are three layers of filtering: at the top there is the blast filtering, which controls how many gene candidates will be processed. Then the (p2g) filtering and (p2g) refiltering, both of which are at the end of the pipeline. All filtering steps are user definable, which can create filters adapted to his/her protein family of interest. We provide a sensible default filtering for user input families: each alignment is examined and, based on its sequence conservation, a similarity threshold is chosen (AWSI filter). This means that a very conserved profile will output only very similar sequences. Also, when multiple profiles are searched, overlapping matches are assigned to one or the other family based on sequence similarity. For selenoprotein families, the program SECISearch3 (if installed) is also used to identify suitable SECIS elements downstream of the coding region of the candidate selenoprotein genes. Selenoprofiles workflow can be easily customized to perform similar operations: running custom code for specific gene candidates, then storing and outputing genomic annotations (see custom features).




*Graphical summary of the selenoprofiles pipeline.*

Selenoprofiles normally performs the full pipeline, taking care of skipping the steps executed previously. The steps of selenoprofiles are: blast, exonerate, genewise, prediction choice, prediction filtering, database storage, output; these are denoted respectively by the step-options *-B -E -G -C -F -D -O* (see figure). After the filtering step,

results are stored in a SQLite database. When selenoprofiles is run, it checks first if the results database contain already the results, and if it does, it passes directly to the output step. If the user specify any step-option, the execution of the corresponding step and of all next ones is forced. This is necessary if you changed parameters or profile specific procedures. If for example you changed some parameter relative to the filtering phase, you must force filtering and output with *-F*. **Important:** even when output is forced, selenoprofiles will overwrite previous output files, but it will never delete any. This may lead to overlapping predictions in the output, thus we recommend to always delete the output files before any second run on a certain genome. For the full chronological list of operations performed by selenoprofiles, see Appendix 2.

## Building a profile

A profile alignment is a set of aligned sequences which allows to find and predict genes that *fit* in it. This source of information is used in different forms by the slave programs to find regions of homology and model the genes found in the target.
Building a profile alignment means formatting it to be used with selenoprofiles. You need a protein sequence alignment named after your family, with only alphanumeric characters or underscores. The only format accepted is fasta (aligned, with gaps as "-"). The title names must have a unique starting word.
When you provide a fasta file as profile argument, selenoprofiles will attempt to build it with default options. Optionally, you can use the script *selenoprofiles_build_profile.py* (located inside the installation directory) to build the profile before running selenoprofiles. This script allow to control profile-specific parameters and procedures, using the library of functions described in this manual. It also provides other utilities, such as a tool to trim redundant sequences. For fast runs, alignments should be trimmed to less than 100 sequences. Small profiles are also discouraged, since the variation in profile sequence similarity is an important determinant for filtering. A minimum of 10 sequences is suggested. For a guide to build a good profile, see Appendix 1.
When a profile is built, its sequences are reordered (overwriting the input file) and two files are produced: a *.profile_data* file, containing data derived from its sequences for lazy computing, and a *.config* file, with all the non-sequence information associated to this profile. The sequences are ordered based on "completeness" respect to the whole profile. The *.config* file can be inspected and edited with any text editor to modify the profile attributes. Its content can vary a lot: every profile can have a high number of attributes associated; all the attributes that are not defined in the *.config* file of a profile are taken from the selenoprofiles main configuration file.
The only options in the *.config* file that the user typically wants to check are the filtering procedures. By default, a loose blast filtering is used (evalue < 0.01). For p2g filtering, only predictions spanning at least 40% of the profile length (or longer than 60 aminoacids) are kept. In the last layer of filtering (p2g refiltering), the AWSI measure is evaluated. As explained later (see AWSI score), this method computes a score of average similarity of the candidate with all profile sequences, and compares it with the average similarity within the profile itself. In this way, very conserved profile alignments will output only very conserved genes. The user can modify the filtering procedures by adding (or editing) lines in the profile *.config* files. It is also possible to edit the default values in the main configuration file, affecting all profiles with no procedures defined in their *.config* file.
For example, to tighten up the blast filtering for a certain family, open its its *.config* file and insert this line:

```
blast_filtering = x.evalue < 1e-8
```

In the next example, we modify the default p2g_filtering for all profiles; find and edit the line corresponding to this in the pipeline main configuration file (*selenoprofiles.config*):

```
p2g_filtering.DEFAULT =  x.coverage()>0.5 and x.label != 'pseudo'
```

This will require the predictions to span at least half of the profile width, and to possess a label different than pseudo. A single label is assigned to each result during the pipeline workflow. The labeling procedure can also be customized (see option -add section). By default, there are only two possible labels: *pseudo* (assigned to all results with in-frame stop codons, or with insertions or deletions creating frameshifts), and *homologue* (assigned to all others). Additional labels are possible for selenoprotein families.

**Other elements in the profile configuration file**
Let's inspect an example of a built-in profile: AhpC. Its *.config* file contains:

```
name = AhpC
queries = all
blast_options = SELENO
exonerate_options = SELENO
genewise_options = SELENO
```

- *name*: the name of the family. Taken from the input file name.
- *queries*: the queries in a profile are those eligible to be used with exonerate and genewise. In a well curated, clean alignment, all sequences are queries. The value of the *queries* attribute can accept various formats (see *selenoprofiles_build_profile.py --help*), but normally you won't need to change it from its default value, *all*. Just for selenoproteins, it is important to take particular care on the alignment of the position(s) with selenocysteine. Thus, by default a sequence is excluded from the queries if it has no residue aligned to the position of selenocysteine in the alignment, or to any of them if there are many such positions.

All other elements may or not be present in the file. In the case they are not, they are set to the defaults specified in the selenoprofiles main configuration file. All these options can be controlled by keywords. Keywords are defined in the main configuration file, in the form:

```
option_name.KEYWORD1 = value
```

This sets the keyword *KEYWORD1* for the option called *option_name*. This will allow you to refer to this keyword in any profile configuration file when defining that specific option.
For example, in the main configuration file you have this line:

```
blast_options.SELENO        =   -b 5000 -F F
```

which allows the profile configuration files to bear this:

```
blast_options = SELENO
```

This tells the program that it must refer to the keyword *SELENO* for the *blast_options* of this profile, which is translated to the value: *-b 5000 -F F*
This and some other elements in the profile configuration files are **program options**. These can be recognized by their suffix *_options*. These are basically strings which will be concatenated to the command line when the corresponding program is run: blast (psitblastn), exonerate, genewise or tag_blast (when a *tag_score* or *GO_score* method is called). *SELENO* is set as the value of all program options when at least a selenocysteine

(U) is detected in the alignment. This allows to use specific scoring schemes for these columns.
We have already seen examples of another type of profile configuration element, the **filtering procedures**. These can be recognized by their suffix *_filtering.* All filtering procedures inside selenoprofiles are written in python code and use the variable *x* to indicate the prediction to which the filtering procedure is applied. For advanced filtering, you should see the advanced usage section to understand and be able to use its syntax. There are three types of filtering: *blast_filtering* (applied to all blast hits to decide which ones will be considered), *p2g_filtering* and *p2g_refiltering* (both applied as a final filter to decide which predictions will be output).
Filters represent the most important non-sequence information layer of a profile. As a rule of thumb, when you use a new profile you may leave the filters as defaults and run selenoprofiles a first time. Then, inspect the results and change them to calibrate your profiles, then rerun selenoprofiles (removing output file and using step option *-F*). You will learn how to create filters suitable to your protein family in subsequent sections.
There are more elements that can appear in a profile configuration file. These will be treated later during this manual as their use is explained: *max_blast_hits_number, clustering_seq_id, max_column_gaps_for_blast_query, tag_db, gi2go_db, tags, go_terms, neutral_tags*.

## Configuration file vs command line

The configuration file contains all the settings of selenoprofiles, and it can used for a deep customization of its behavior. In selenoprofiles, all options can be specified in the configuration file or in the command line, with the latter overriding the former default values.
Options in the configuration file have the form

```
option_name = value
```

while in the command line they have the usual form

```
-option_name value
```
[2]

Next, we list the system settings options in the main configuration file:

- *temp* = folder

This will be used for the temporary files produced during the workflow. Actually, a subfolder with a random name is used, and deleted at the end of the computation. You should choose a temporary folder with free space at least of the size of the target file.

- *save_chromosomes= 1 / 0*

When active, subfolders are created in the temp folder to unpack the multifasta target files into single fasta files. Only the necessary chromosomes (or contigs) are extracted. Following principles of lazy computation, these files are saved and reused when selenoprofiles is run again on the same target. If you turn this option off, the single fasta files will be instead written in the random name subfolder and deleted at the end.

- *profile* = profile_name / set_keyword / file

The keyword *profile* in main configuration file denotes the default set of profiles searched, defined as described here. The default value is *eukaryotic*, which is a keyword for all eukaryotic built-in profiles.

---

[2] To catch option values of multiple words in command line, use double-quotes to delimit them:
`-blast_options " -a 4 "`

- *profiles_folder*= folder

As said, you can provide the profiles list to be searched using directly paths to alignment files, keywords for set of families, or family names. When you use family names, this is the folder where the alignment files named after them are searched for. If you want to use a set of custom profiles, you should create a folder for them and set this option to point to it.

The main configuration file is the place where keywords are defined. Keywords can be used for the categories presented in the last chapter, for profile specific parameters and procedures. There's an additional element that use a keyword logic: the set of families.

```
families_set.machinery  = sps,sbp2,pstk,secp43,SecS,eEFsec
```

This line in the configuration file allows to use the word *machinery* as a *-profile* option. This will be unpacked into the list of families on runtime. For very large sets of input profiles, we recommend to use option *-fam_list* that overrides *-p* (or *-profile*) option.

Other options found in the configuration file are:

```
three_prime_length=3000
```

This is the length of the sequence cut when the method three_prime is called. When used for selenoprotein families, if SECISearch3 is installed, this is the width of the region downstream the coding sequence where it is used. The option *five_prime_length* is not present in the default configuration file, but it can be set by the user on runtime or written in the configuration file.  This is necessary only if the output five_prime is active.

```
blast_opt     = -a 7
exonerate_opt =
genewise_opt  =
```

The *_opt* program options are concatenated to the command line when using slave programs are run, exactly as *_options* program options in the profile configuration. The difference between *_opt* and *_options* is that the former are always used, while the latter can be set for every profile. In the example, the option *-a* for blast specifies the maximum number of CPUs to be used for computation. This will be used for all psitblastn searches.

```
exonerate_extension = 200000
genewise_extension  = 100
genewise_tbs_extension = 10000
```

The extension parameters (expressed in nucleotides) are used in exonerate or genewise routines, and measure how much the blast alignment are expanded to search for full gene structures (as described in the next section).

```
species_library   = /somepath/names.dmp
GO_obo_file       = /somepath/gene_ontology_ext.obo
```

These two options tell the system where the reference file for the species names and the GO annotation file is located. The first is compulsory present on your system, the second is not.
Some lines in the configuration file start with *ACTION*:

```
ACTION.pre_choose._improve1 = if x.prediction_program()=='blast':  x.remove_internal_introns()
```

This defines an action. Actions are operations that are run on every prediction. They may serve different functions. Actions are performed at a certain point during the workflow, defined by their category (in this case *pre_choose*). Some actions are active by default to improve the predictions and are covered in the improving prediction chapter of the next section. You can learn more on actions (including how to write them) in a later chapter.
There are many more options, some of which will be mentioned later. The full list of options can be obtained by running selenoprofiles with *--help full*

## The results folder

The results folder contains all files produced by selenoprofiles. A single folder can store the output data for multiple targets. For each one, a subfolder for target is created concatenating with a dot the species and target names (e.g. Homo_sapiens.genome*)*. Think to the results folder as a working environment for a project that include searching multiple profiles in several species, or also in several targets for the same species (for example, genome and transcriptome).
The content of each target folder will vary depending not only on the results of the search, but also on the options specified by the user.
In its most complete form, the target folder will contain the file:

- results.sqlite database storing all filtered results on this target

and the folders:

- output contains the output files of selenoprofiles
- blast contains the psitblastn output files
- exonerate contains the exonerate output files
- genewise contains the genewise output files
- prediction_choice contains the output files for the prediction choice/labelling step
- filtering contains the output files for the filtering step
- tag_blast contains the output files of the tag blast, if used (see tag blast)

Inside these folders, files are named with a prefix for the profile name. Exonerate and genewise each produce a file for each blast hit satisfying the filtering conditions. Here, the file names are composed adding to the profile name a index linked to a blast hit (example: *fam.1.exonerate*). Additionally, these files are contained in subfolders of the exonerate folder named as each profile, to avoid having too many files in single folders when tons of hits are found by loose profiles. In the output folder, files names contain also the label assigned to each result, followed by the file format (example: *fam.1.selenocysteine.gff*)

Example: files produced searching *SelM* (profile name) in the *genome* (target name) of *Macaca_mulatta* (species name).

```
results_folder/Macaca_mulatta.genome/results.sqlite
results_folder/Macaca_mulatta.genome/link_target.fa
results_folder/Macaca_mulatta.genome/blast/SelM/SelM.psitblastn.1
results_folder/Macaca_mulatta.genome/exonerate/SelM/SelM.1.exonerate
results_folder/Macaca_mulatta.genome/genewise/SelM/SelM.1.genewise
results_folder/Macaca_mulatta.genome/prediction_choice/SelM.tab
results_folder/Macaca_mulatta.genome/filtering/SelM.tab
results_folder/Macaca_mulatta.genome/output/SelM.ali
results_folder/Macaca_mulatta.genome/output/SelM.1.selenocysteine.p2g
```

If you plan to run selenoprofiles massively, you may want to delete the intermediate files that it produces to avoid an excessive use of disk space. All subfolders listed above can be deleted; as long as results have already been stored in the results database, selenoprofiles will be able to retrieve the desired predictions and produce output files. When run with option *-clean*, selenoprofiles will delete all such subfolders (apart from *output/*) at the end of the computation.

## Output options

As you see in the above example list, an alignment file (`SelM.ali`) is produced as output. This fasta formatted alignment contains the sequences of all results found in this target

along with all the profile sequences. This is useful to inspect all results found a certain target, and compare their conservation and spanning respect to the profile. The alignment is computed by mapping each pairwise alignment constituting a prediction (protein-to-genome, or p2g) into the profile alignment. The program mafft is used to realign only certain columns of the alignment which deteriorate when adding many predictions in this way.
In the file, the fasta headers of the results start with the “output id” of the prediction (“family.index.label”, for example *SelM.1.selenocysteine*) and contain also other essential information.
The rest of the output files are named after the output id of the prediction plus the format. The available output formats are:

- p2g default output format (explained later in the visualizing results section)
- fasta protein sequence
- gff genomic coordinates in GFF
- gtf genomic coordinates in GTF
- cds coding sequence in fasta
- dna the full gene sequence, including introns, in fasta
- three_prime the sequence downstream of the prediction
- five_prime the sequence upstream of the prediction (must specify *-five_prime_length*)
- introns the sequence of all introns split in a multi-fasta file

The desired output formats are read from the options in the command line or the configuration file starting with *output_*: for example if option *-output_fasta* is active, the fasta files of all results will be produced, and so on. For all these formats, it is possible alternatively to produce a single file containing all results on a target, by adding *_file* to the option and providing an argument. If for example you want to produce a single GTF with all predictions, use

```
Selenoprofiles [...] -output_gtf_file  all_results.gtf
```

In the main configuration file you can see what file formats are produced by default. Out-of-the-box, the only active output options are *output_ali* (for the alignment of results along with the profile) and *output_p2g*. Sometimes, you may also want to use a different output folder: this can be chosen with *-outfolder*.
You can define your own output format by writing a method in python, and add it to selenoprofiles using the *-add* option (see later option -add).

## The results database

At the end of the pipeline, before outputing, results are stored in SQLite database called *results.sqlite,* placed inside the subfolder for this target in the results folder. It is possible to browse through results opening the database files with an SQLite browser, although normally you will not need to. The script *selenoprofiles_database.py* can be used to query or modify the database for most common operations.

# Inspecting results: *.p2g* format

Selenoprofiles native output format is the following: *.p2g*

```
FILE: /results_folder/Gallus_gallus.genome/output/Ahpc_1.4.pseudo.p2g
--
Output_id:  AhpC.3.pseudo
----------  -------------
-Species        Gallus gallus                                   -Taxid 9031
-Target         /db/Genomes/Gallus_gallus/genome.fa
-Chromosome (-) Z
-Program        exonerate
-Query name     Anolis_carolinensis
-Query range    34-226     length:226   coverage: 0.85
-Profile range  58-289     length:303   coverage: 0.77    sec_position: [99]
-ASI:           0.2521     (ignoring gaps: 0.2708)
-AWSIc:         0.4486     Z-score: 1.06
-AWSIw:         0.4561     Z-score: 1.145
-State          kept

------- alignment -------
Query   AAQCPLLDAAGEKTPFGTLFRDRKAIVVFVR <---Intron---> HFLUYTCKEYVEDLAKIPKKYLE <---Intron---> DANVRLVVIGQSSP
        || | //|| | / ||| |/| /|||||||| <   435nt    > /|| |||||||||||/|/ ||/ <   1167nt   > /|||||/||||||
Target  AAYCLVVDADGSRIPFGALYRRQKAIVVFVR               NFLCYTCKEYVEDLAKVPRSYLQ               EANVRLIVIGQSSY
        ggtttgggggaaactggttaccagaggtgc                 attttatagtgggcgagcaattc                ggagacagagcttt
        ccagtttacagggtctgctaggaacttttg   gt     ag    attgacgaaataatcatcggata   gt     ag    acatgttttgacca
        ccccgggcgcgtgcccgcgcggggccggtgg                tcgtcctggtaacgaaccgttaa               aatggtattagatt
                                                          *

Query   DHIK <---Intron---> PFCHLTGYSHEIYVDPGREIYKILGMKNGETADTPV <---Intron---> QSPHVKSSFLSGHIKSIWRAVFSPAFDF
         ||| <   409nt    > ||| ||||/||/|||| |||||/|||| ||  |  | <   197nt    > |||||||| | | |/|/|||/ ||||||
Target  HHIK                PFCSLTGYTHEMYVDPQREIYKMLGMKRGEGNDVSV                QSPHVKSSMLLGSIRSMWRAMTSPAFDF
        ccaa                 cttatagtacgatggccagataacgaaagggaggtg                 caccgataactgaaaaatagaaacgtgt
        aata   gt     ag      ctggtcgacaatatacagataattgtaggagaatc    gt     ag   tagcatacgtttggtggtggctcgcctat
        ttcg                 ctctatgtatagtataagattagtcgaatattcaa                 ggcttaaacgcgcttatggaagtcattcc


Query   QGDPTQQGGALILGPG <---Intron---> NQVHFVHLDKNRLDHVPINTVLQLA ! FRAME ! GVQTVNFTQRSQIIDV
        |||| |||| |||||| <   553nt    > |/|||/| |/|||||||||/||||| ! SHIFT ! ||  |||| / |||||
Target  QGDPAQQGGTLILGPG                 NEVHFLHHDRNRLDHVPINSVLQLA    1nt    GVNPVNFTNKPQIIDV
        cggcgccggatatgcg                  aggcttccgaaatgcgcaatgtccg           ggacgataaaccaagg
        agaccaaggctttgc    gt     ag   gaatattaaagagtaatctacttatc     c     gtactatcaacattat
        aacttagaatgcaca                 ttatttgtttacagtttcttatggga           atcaatcacacgttta

------- positions -------
Exon 1    41768514     41768606
Exon 2    41768010     41768078
Exon 3    41766789     41766842
Exon 4    41766274     41766379
Exon 5    41765945     41766076
Exon 6    41765315     41765391
Exon 7    41765266     41765313

-------- features -------
None
--------- 3' seq --------
Total sequence length available downstream >= 6000
Sequence until first stop codon:
TGA
 *
```

The header of the file contains the basic information about this gene prediction, and is pretty self-explanatory. Some numbers are reported: the ASI is the average of the sequence identities computed comparing the candidate sequence with each one of the profile sequences, and gives an idea of how much it *fits* in the profile. AWSIc and AWSIw are analog similarity scores (but more sophisticated), and are detailed later (see AWSI score). Their linked Z-score is obtained by comparing the score of this candidate sequence with the distribution of scores of the sequences in the profiles, comparing each one to all others. The default refiltering requires the AWSIc Z-score to be greater than -3.
Next in the output file, there is a line indicating the attribute *State*. Normally this is *kept*, unless you forced output of predictions normally filtered out (using the *-state* option as explained later).
Then, the query-target pairwise alignment constituting the gene structure prediction is shown. Between the amino acids, bars are used to show the identity “|” or the similarity “/“ of the aligned residues. Predicted in-frame stop codons (absent in the example) and selenocysteine columns in the input alignment are marked below with X and * respectively.
An insertion in the target producing a frameshift is present near the end of the example prediction. When analyzing low-quality genomes, frameshifts and stop codons should be not trusted, and checked with sequence data from the same organism by a different source, if available. In this example, the gene structure looks well conserved except for the insertion. The presence of introns and good splice sites also suggest that this is not a pseudogene. Thus, this result should be considered a valid gene despite its label *pseudo*. This is the reason why by default selenoprofiles does not filter out potential pseudogenes. When working with high quality target sequences, one can decide to filter out results with this label, as shown later.
Next in the file, the genomic positions of the exons are reported. The first nucleotide of a chromosome (or scaffold) is indexed as 1. The frameshift is considered as a short intron, dividing the real exon in two.
In the next section, all features found belonging to this predictions are shown. Features are objects linked to a p2g result, which the user can manipulate to add layers of analysis to the pipeline, and get custom output here in the *.p2g* file (as explained later).
Finally, the sequence at the three prime of the gene structure prediction is reported, until the first stop codon. In this example a TGA is found right downstream, indicating that the coding sequence prediction is complete at the 3’.

## Searching multiple targets

Selenoprofiles is meant to search for one or more protein families of interest in many species and compare results. We suggest to use a certain structure for the file paths in this case. The genome sequences of all investigated species should be in subfolders named after the species, with spaces replaced by underscores. The file name of the genome fasta sequence file (or a link to it) should be *genome.fa*. Example:

```
/home/genome_links/Drosophila_melanogaster/genome.fa
/home/genome_links/Homo_sapiens/genome.fa
/home/genome_links/Mus_musculus/genome.fa
/home/genome_links/Pan_troglodytes/genome.fa
```

When selenoprofiles is run on a target, it will format the sequence database file creating files such as *genome.index, genome.lengths* in the same species subfolder. Also, an advantage of this structure is that selenoprofiles will detect the species name from the target path, thus option *-s* is not strictly needed.
After the pipeline has been run, the results of a profile in many targets should be inspected all together. The program *selenoprofiles_join_alignments.py* searches for the *.ali*

alignments in the results folder and joins those of the same family into new alignments, which will contain the results in all targets along with the profile sequences. In the new alignment, the title identifiers corresponding to the predictions look like this:

*>family.id.label.species_name.target_name*

They are different from those in the previous *.ali* files, in that they contain the species and the target name as part of the first word, to make each title identifier unique. For more information on *selenoprofiles_join_alignments.py*, run it with option *--help.*
Every prediction consists of a pairwise alignment between a profile protein query and a nucleotide target. The new, joined alignments are produced by mapping all pairwise alignments to the profile. A procedure is used to detect columns that are misaligned by the process (for example when a insertion is present in many targets, but absent from all queries), and mafft is used to realign them.
Such procedure of alignment mapping is used to ensure the consistency of the alignment between the profile sequences, no matter how many predictions are present in the same alignment. Anyway, you may want to realign your results using a more sophisticated tool, such as T-coffee (http://www.tcoffee.org/).
The resulting alignment of your results can be inspected using a number of programs (http://en.wikipedia.org/wiki/List_of_alignment_visualization_software).
The joined alignments are also the input to the program *selenoprofiles_tree_drawer.py*, for visualizing the results of (potentially) multiple profiles in (potentially) multiple species with known phylogenetic relationship. The programs requires the installation of the ete2 tree python environment (see http://ete.cgenomics.org/), and loads a tree of the investigated species in newick or phylip format: round parenthesis such as “(“ and “)” are used to group lineages that cluster together. With few species, one can manually write such a file. For example the tree for human, chimp, mouse in simple newick would be:

```
((Homo sapiens,Pan troglodytes),Mus musculus);
```

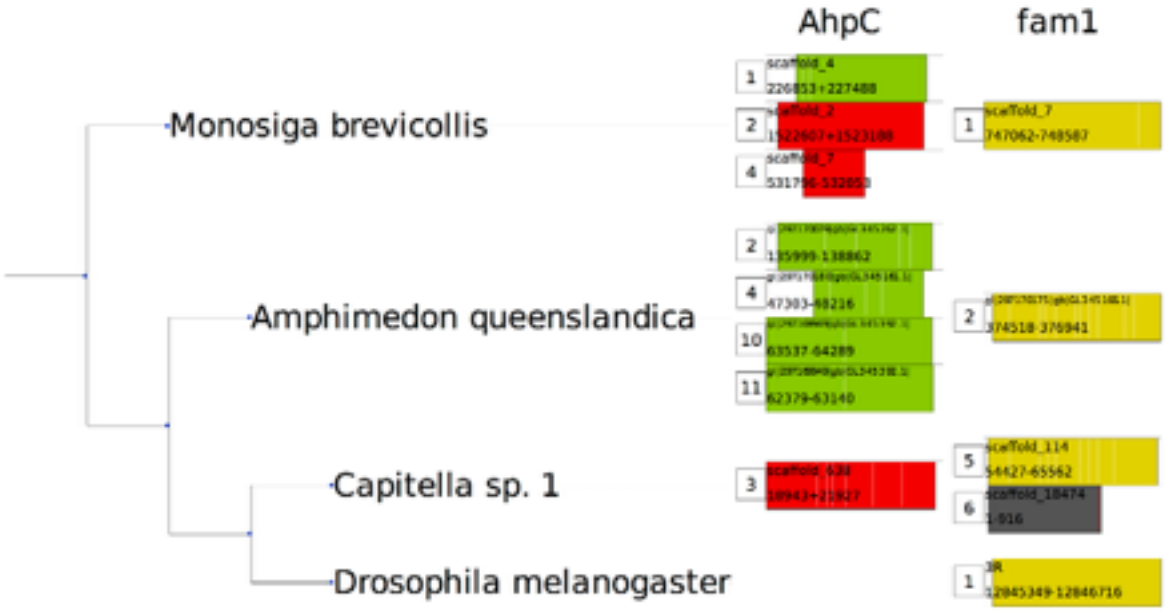
If we add rat and fruit fly, we have:

```
(((Homo sapiens,Pan troglodytes),(Mus musculus,Rattus norvegicus)), Drosophila melanogaster);
```

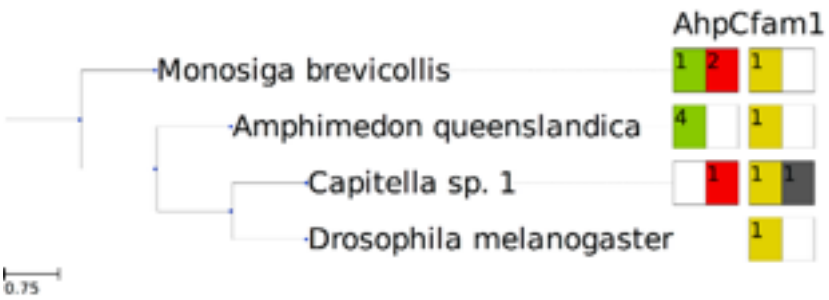
For searches on wide range of species, it may be useful to derive their rough tree from the NCBI taxonomy database. This can be done directly at its portal at http://www.ncbi.nlm.nih.gov/Taxonomy/CommonTree/wwwcmt.cgi, or with more automated tools such as http://github.com/jhcepas/ncbi_taxonomy.
Once you have your joined alignments of results, for example for profiles AhpC and fam1) and a species tree containing (at least) your species of interest, you can run:

```
selenoprofiles_tree_drawer.py AhpC.ali fam1.ali  -t species_tree.nw
```

This will open the ete2 graphical environment, showing something like this:




The species tree is indicated on the left. It contains only the species with at least one prediction. The results for different profiles are shown as different columns, on the right. Multiple results for a profile in a species are shown as adjacent rows.
Each result is shown as a colored rectangle. A numeric tag at its left indicates its selenoprofiles numeric id. The color depends on its label, with an hard-coded dictionary for selenoprotein families: green for selenoproteins, red for cysteine homologues, (...).
For standard, non-selenocysteine containing families (such as fam1 in the example) the only labels are *homologue* (yellow) and *pseudo* (dark grey). The dictionary of colors can be edited by the user directly inside the script *selenoprofiles_tree_drawer.py* (see *label_to_color* declaration). The rectangle width and position indicates the prediction coverage and horizontal span when mapped in the profile alignment. You fill find some additional information printed inside each rectangle: the id of the chromosome (or contig), and the genomic coordinate boundaries, separated with “+” for results on the plus strand, and “-” for results on the minus strand. Finally, the intron positions as relative to the protein alignment are shown as vertical white lines. When frameshifts are present, they are shown as vertical red lines. s*elenoprofiles_tree_drawer* can be used to produce images or pdf files summarizing even large sets of results, and has many options for customization (see *selenoprofiles_tree_drawer.py --help*). When a very high number of results have to be visualized, certain options can be used to reduce the amount of information per result shown. The option *-a* in particular allow to compress the number of results by label:



## Searching RNA sequences or bacterial genomes

Selenoprofiles was originally designed to search eukaryotic genomes. Here, coding sequences are usually split in different exons, adding the complication of splice site prediction. However, the pipeline can be used also to search RNA sequences, or bacterial sequences; in both these cases, splicing is not expected to occur.
To perform this kind of searches, it is convenient to use some options to modify selenoprofiles workflow. The main change is introduced using adding *-no_splice* to the command line. This option will cause a few modifications. First, each blast hit is always considered singularly: the procedure of merging blast hits by co-linearity (described briefly later) does not occur. Secondly, exonerate is used in "protein2dna" mode, instead of "protein2genome" mode; this is faster and avoids prediction of splice sites. Lastly, genewise is turned off, since it is not worth running for this kind of search.
Selenoprofiles normally performs a few modifications to improve the gene structure predictions by the slave programs, as described later. When searching RNA sequences or bacterial genomes, some of these procedures do not make sense, and it is worth turning them off. To do so, edit the main configuration file and comment (using "*#*") the following lines:

```
# ACTION.pre_choose._improve1 = if x.prediction_program()=='blast': \
                                        x.remove_internal_introns(min_length=18)
# ACTION.pre_choose._improve2   = x.clean_inframe_stop_codons(max_codons_removed=10)
# ACTION.pre_choose._improve3   = x.exclude_large_introns(max_intron_length=140000)
```

In the standard workflow, there are two additional modifications aimed at completing the gene prediction at the 5' and 3', searching for the starting methionine and stop codon respectively. Since selenoprofiles is meant to be an homology-based tool, normally the use of these procedures is very limited: only a short stretch of coding sequence can be added (*max_extension* parameter), and only when a narrow portion of the profile sequence is left unaligned at the relevant side (*max_query_unaligned* parameter). Also, completion at 5' normally stops at the first methionine found upstream (*full* parameter, default value is *False*). When searching high quality RNA sequences or bacterial genomes, you may want to increase the extent of these methods. To do so, edit the relevant lines in the main configuration file. Here below, we show an example with increased parameters.

```
ACTION.post_filtering._improve4 =  if x.filtered == 'kept':          \
                    x.complete_at_three_prime(max_extension=100, max_query_unaligned=300)
ACTION.post_filtering._improve5 =  if x.filtered == 'kept':          \
         x.complete_at_five_prime( max_extension=100, max_query_unaligned=300, full=True)
```

## Trading off speed and accuracy

Selenoprofiles workflow can be easily adjusted to the user requirements. Typically, the most common modifications concern the total run time and the accuracy of results. By default, the program is adjusted for maximum sensitivity and specificity at gene level, aimed to the accurate characterization of a limited number of protein families. Sensitivity is achieved by keeping loose filters for blast. Specificity is achieved through the p2g filters at the last step, when every candidate sequence is evaluated against its profile.
We describe here a few common methods to alter the default trade-off. All options are written as if they are to be used in the command line; however, you can make them part of your default workflow by adding them instead to the configuration file, following the guidelines explained above.
Normally, most of gene structures will have a correct prediction of splice sites. However, this may not be true for those predicted by blast alone. If the prediction of correct splice

boundaries is very important to you, use option -*no_blast*. This forces gene structures by exonerate or genewise to be chosen during the prediction choice step, sacrificing a little sensitivity (particularly for most distant homologues) for an improvement in splice site accuracy.

Run time can be improved in many ways. We report here our suggestions in order; each of these can be used in combination with the previous ones, and sacrifices some sensitivity.

Using " *-genewise_to_be_sure 0 "*, genewise will be executed only when exonerate produced a gene prediction for a certain blast hit. This saves a large fraction of computational time with little sensitivity cost.

A more extreme strategy is to completely turn off genewise, using option *-dont_genewise*. Exonerate is much faster than genewise, and in the majority of cases it is just as accurate. We generally use this configuration when running selenoprofiles when using a thousands of profiles to annotate complete genomes.

Another way to improve speed is to alter blast filtering, which makes sense only if a lot of candidates are processed in your searches. You can either decrease the maximum number of blast hits considered (*max_blast_hits.DEFAULT* in the configuration file, normally extremely high: 2500), or set a stricter evalue for blast filtering (default is 1e-2 — see *blast_filtering.DEFAULT*). Note that when using the standard final filter, the candidate sequence is compared to each profile sequence. For this reason, a profile with lots of sequences is likely to cause a longer run time. For this reason, we suggest to trim profiles to have a suitable number of sequences, when they are too many (see Appendix 1).

# The Selenoprofiles pipeline

## Psitblastn

Selenoprofiles uses psitblastn from the NCBI blastall package. This program can be considered an extension of tblastn. Instead of using only a single sequence as query, it considers also a Position Specific Scoring Matrix (PSSM). This allows to utilize the relative proportions of allowed residues at each profile position. Normally, its more famous relative psiblast (extension of blastp) is used iteratively against a sequence database, building a PSSM with the matches it finds. In our use of psitblastn, no iteration at all is performed, since the profile alignment is already provided as input and the PSSM can readily be derived.

• Pre-clustering

We experienced that when a profile is very broad (i.e., contains sequences quite dissimilar to each other), the psitblastn search is not very sensitive. For this reason, selenoprofiles implements a procedure that analyzes the input profile alignment in terms of its variability, and clusters its sequences based on their sequence identity. If the profile has a high variability, then this procedure will produce more than one cluster.
Then, a psitblastn search for each cluster is performed: one PSSM is built from the sequences of each cluster. Consequently, often there are overlapping blast hits coming from the searches of different clusters. Those are merged, keeping only the best one for each overlapping set. The sequence identity threshold for the profile clustering procedure can be defined for each profile (*clustering_seqid* parameter), or goes to the default value defined in the main configuration file.

• Consensus blast query

Psitblastn build a PSSM along the positions of a certain sequence of the profile, elected as the blast query. In our experience, the choice of the blast query has a big effect on the results of the search. The blast query is built for each search, as a “consensus”. Its sequence is given by the most present amino acid at each position of the alignment (or of the cluster, if more than one is present). There are two exceptions to this. For selenoproteins, in the positions where at least a Sec is detected, the blast query always bears a U. Then, the positions featuring a lot of gaps in the alignment are skipped. The maximum percentage of gaps for a column depends on the option *max_column_gaps_for_blast_query*, either specified in the profile configuration or set to the default in the main configuration file.
For technical reasons, all blast hits loaded in selenoprofiles are transformed so that their alignments are between the target and a unique query sequence, named the master blast query. This allows to have a more homogenous kind of data for subsequent computation: otherwise, blast hits coming from different clusters searches would have different sequences as query.

• Merging exons by co-linearity

After the overlapping hits from the various cluster searches are removed, blast hits are once again analyzed, and those likely to be exons of the same gene are joined: they are merged by co-linearity. This means that if a blast hit is downstream of another one, and also the correspondent portions of the aligned query sequences are one downstream of the other in the same direction, the blast hits will be merged into a single object (if they are not too far away). This procedure is done to minimize redundant computation.

• Blast filtering

Blast hits are filtered according to criteria that may be specified for each profile. In our experience, different protein families need very distinct criteria. Some families typically match a lot of spurious hits, while some others need loose filters to find all results. All filtering procedures in selenoprofiles are written in python and can be customized by the user, utilizing a set of methods that are already provided or can be created by the user. Filtering is detailed in a later section. Blast filtering is performed actually before removing redundancy across cluster searches, and also before merging by co-linearity. This is because merging blast hits requires loading them all into memory, sorting them and parsing them -- which sometimes would take very long if all blast hits in a output file are considered.

If for some reason you want to inspect manually the blast hits passing the filter, you can use option *-filtered_blast_file* and provide a file as argument, which will be created. Blast hits within this file have not been subject to inter-cluster or co-linearity merging.

• Maximum number of blast hits

In selenoprofiles, the computation is largely dependent on the number of blast hits passing filtering. For this reason, there is a fixed maximum number of blast hits which can be considered. The default value is extremely loose: 2500. When the limit is passed for a family, a warning is printed on screen and the workflow follows keeping only the blast hits found so far. Blast hits are read in the order they are in the blast output file. Blast sorts the hits according to the chromosomes (or contigs) they are located on, ordering the chromosomes according to the e-value of the best HSP found on them. This way of sorting is not strictly best-to-worse but it is similar, therefore most likely you won't lose any bona-fide gene because you reached the maximum limit of blast hits.

Also, the blast outputs produced searching the different clusters are read in order, with the cluster containing the highest number of sequences being first. Therefore, the first blast output read should be the most representative.

In an older version of selenoprofiles, the computation would simply stop if the max number of blast hits is reached. This behavior can be restored by setting off the relevant option, with *-blast_filtering_warning 0.*

## Exonerate

Each alignment coming from the blast phase is used as a seed to run exonerate in the corresponding genomic region, using a proper extension procedure.

• Reading and joining exonerate predictions

Exonerate is run on a chromosomic region in which a blast hit was found, and typically it will give a single gene structure prediction in output. Nonetheless, this is not always the case. For this reason, selenoprofiles considers only the exonerate prediction which, among those in its output file, overlaps with the blast hit used as seed. If more than one overlapping prediction is present (very rarely), the best scoring one is taken.

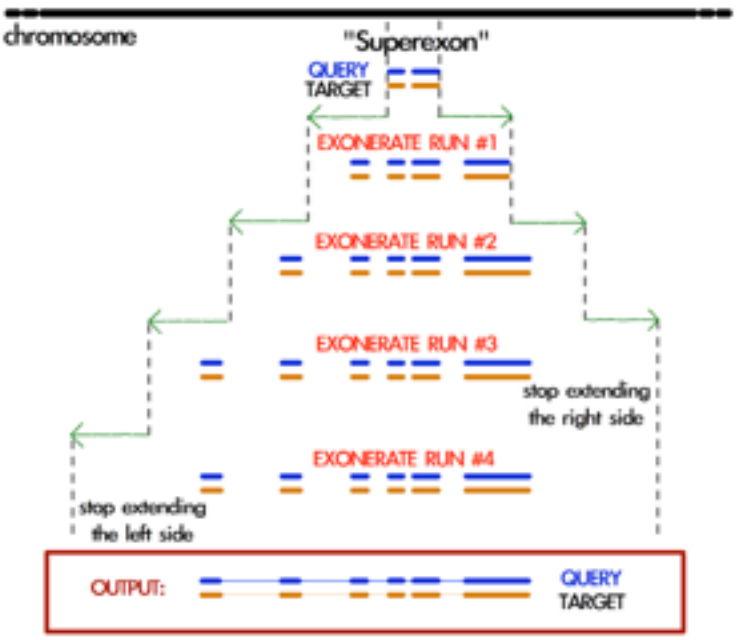
Also, exonerate generally joins the exons belonging the same gene, including the prediction of splice sites. Nonetheless, often no good scoring splice sites are found and such predictions may be found separated. Selenoprofiles attempts to merge the "main" exonerate prediction with the others in the same file, using the co-linearity concept previously mentioned for blast hits. This behavior is turned off when selenoprofiles option *no_splice* is active.

• Cyclic exonerate

Exonerate is run through a peculiar routine called cyclic exonerate (see figure below; see also selenoprofiles paper). This procedure comes in response to the following problem: if we want to run exonerate on a certain genomic region where a blast alignment gave us the hint of an homology match, we need to decide the boundaries of the region searched by exonerate. Of course the region provided by blast needs to be extended, but by how much? Gene sizes are incredibly variable. Taking the biggest size ever observed would result in a huge amount of useless computation, while on the other side taking an average would obviously be inappropriate for a fraction of cases.

This routine solves this problem by running exonerate more than once, increasing progressively the genomic space searched on both sides by a fixed parameter. The cycle stops when a run predicts the same coding sequence of the previous one. If the extension parameter is chosen greater than the largest expect intron, the procedure ensures that the widest prediction possible is achieved.

The cyclic routine runs exonerate on average less than 3 times. Given the high speed of exonerate, this is more than acceptable timewise, also considering that this step is not the most computationally intensive in selenoprofiles. Also, if the chromosome (contig) is comparable in size to the extension parameter, the cyclic routine is not performed and the whole chromosome is used as target. The default *exonerate_extension* is 200.000 bases.




*Schema of the cyclic exonerate routine, from selenoprofiles paper (see references). A "superexon" represents either a blast hit, or more than one merged by co-linearity.*

• Choosing the best query from the profile

Exonerate accepts a single sequence query, but in the pipeline the information of a whole profile of sequences is available. Thus, selenoprofiles chooses the best query sequence in the profile for each candidate gene, by searching the query which is most similar to the sequence predicted in the target. To do so, the current predicted sequence is mapped to the profile alignment exploiting the query, which is in common between the prediction alignment and the profile alignment. This is done at every cycle, before running exonerate. At the first run, the predicted sequence in the target is given by the blast prediction, and for

each subsequent run is given by the previous exonerate prediction. Before closing the cyclic routine, it is checked that the best query is still the one that was lastly chosen, otherwise one more cycle is run.

• Modifying exonerate behavior for selenocysteine sites

Selenoprofiles was created to predict genes belonging to selenoprotein families. It is able to do so by using special scoring schemes with exonerate and genewise (blast is used with a neutral score at these sites).
When dealing with Sec families, a particular scoring matrix derived from BLOSUM62 is used: the alignment of a "*" character to a stop codon in the target is scored positively. When the query is chosen from the alignment, its sequence is modified before it is used by exonerate: all the positions which contains at least one Sec in the profile are changed to "*", favoring *de facto* the alignment of Sec positions to UGA codons[3].

• Removing redundant exonerate hits

Often, blast hits representing exons of the same genes pass through the co-linearity merge procedure previously described, mainly because this is kept with loose parameters to avoid joining accidentally similar, close genes. When this happens, such blast hits are used to seed cyclic exonerate runs which end up in identical gene structure predictions.
After all exonerate runs are computed, their predictions are analyzed and the redundant ones are dropped, to save computational time in the genewise phase.

## Genewise

Generally, genewise represents the most computationally expensive step in selenoprofiles, together with blast. Genewise performs a tblastn-like alignment complemented with prediction of splice sites, basically just like exonerate, which. Nonetheless, genewise does not use heuristics and its running time is considerably higher. When you need to maximize speed, you can skip the genewise step using option -*dont_genewise*[4].
Genewise is generally run on genomic regions defined by an exonerate prediction, attempting to refine them. Such genomic regions are extended by a parameter, *genewise_extension*, which is only 100 bases by default. Unlike exonerate, genewise is run only once.

• Genewise "to be sure" routine

In many cases exonerate does not produce any prediction in output. This happens particularly for very low scoring blast hits, which cannot be reproduced by exonerate. In these cases, selenoprofiles performs a genewise routine called "to be sure", in which a blast hit (instead of an exonerate prediction) is used as seed of a genewise run. In our experience this rescues many predictions, but it is very computationally expensive. The extension of genomic region in the blast hit is defined by the *genewise_tbs_extension* parameter, which is 10.000 bases by default. One can avoid running this routine using option *-genewise_to_be_sure 0*.

• The query in genewise

As for exonerate, a single query sequence needs to be chosen to be run with genewise. In a standard run, the same query used by exonerate is chosen, as this is already the most similar to the target sequence. When a blast hit is used in the genewise "to be sure"

---

[3] The alignment of Sec positions to other stop codons is also favored. This is collateral, as no way was found for exonerate to favor the alignment only to UGA codons. Anyway, predictions in which a non-UGA stop codon is present in-frame would then be labelled as pseudogenes.

[4] The option -*dont_exonerate* is also available, but not recommended. If used, this has always to be coupled with -*dont_genewise*

routine, the best sequence is chosen from the profile by maximizing identity with the target, in the same way it is done in the first cycle of an exonerate routine.

• Modifying genewise behavior for selenocysteine sites

For genewise, a trick similar to the one described for exonerate is used when searching for selenoprotein families. Each query used is modified to bear a selenocysteine ("U") corresponding to every column of the alignment which possesses at least one. Then, the translation table normally used by genewise is changed, using one in which UGA is translated as "U". The scoring matrix given to genewise is then a modified BLOSUM62, in which a "U" in the target is score positively only to a "U" in the query.

## Improving predictions

In selenoprofiles a few steps are dedicated to the processing of the predicted gene structures, in order to correct them. All of them are implemented as methods of the superclass *p2ghit*, which comprises the classes for blast, exonerate or genewise predictions (see later p2ghit class). These methods are run through actions (see later actions) specified in the main configuration file. You can turn off the improvements methods by removing, or commenting  (with #), the corresponding lines in the main configuration file.

The first improvement is called *remove_internal_introns* and is performed only on blast hits. This method is useful because often blast joins two or more coding exons in a single hit, when the exons are on the same frame and the resulting stretch of unaligned amino acids in the target is acceptable in terms of scoring. A typical blast hit containing an evident intron is shown here:

```
 Score =  100 bits (249), Expect = 4e-20
 Identities = 49/93 (52%), Positives = 59/93 (63%), Gaps = 26/93 (27%)
 Frame = +2

Query: 12     LEPYMDENFITRAFAKMGENPVSVKLIRNKMTG-------------------------E 45
               LEPYMDENFI+RAFA MGE  +SVK+IRN++TG
Sbjct: 103916 LEPYMDENFISRAFATMGELVLSVKIIRNRLTGYV*SLFVFYHIPNFGVHLHTLFSLSRI 104095

Query: 46     PAGYCFVEFADEASAERAMHKLNGKPIPGANPP 78
               PAGYCFVEFAD A+AE+ +HK+NGKP+PGA P
Sbjct: 104096 PAGYCFVEFADLATAEKCLHKINGKPLPGATPV 104194
```

The portion `YV*SLFVFYHIPNFGVHLHTLFSLSRI` is the translation of an intron. It has no correspondence in the query, and it also contains a stop codon (it is normal as introns have no coding constraint). The *remove_internal_introns* method detects these cases by searching the sequence in the target for stretches of at least 18 bp (6 amino acids) not aligned to the query, and removes them from the prediction.

The second improvement is performed by function *clean_inframe_stop_codons*. This is applied to predictions by all programs, and comes from the observation that often these programs include stop codons that should be avoided. This would cause these predictions to be mislabelled as pseudogenes. This method is simple in principle: it checks for the presence of stop codons close to exon boundaries (default maximum: 10 codons). If it finds any, it removes the stop codons and also the portion which links it to the closest exon boundary.

The third improvement is *exclude_large_introns*. This is particularly useful on exonerate predictions, which sometimes possess extremely large introns, due only to spurious similarity with far away regions, and to the presence of decent splice sites just by random. This function detects each such large intron (default >= 140000 nt), and removes all exons (typically just one) at one side of that intron, the side with the smallest coding sequence.

While all described methods are applied before prediction choice, the fourth and fifth improvements are performed at the end of pipeline, only on the predictions passing the filter; nonetheless, they are described here below.
The functions *complete_at_five_prime* and *complete_at_three_prime* are attempts to complete the coding sequence predictions looking for an upstream ATG and a downstream stop codons. Let's see the corresponding lines in the *selenoprofiles.config* file (expanded for readability):

```
ACTION.post_filtering._improve4=
 \\ if x.filtered=='kept':
 \\   x.complete_at_three_prime(max_extension=10, max_query_unaligned=30)

ACTION.post_filtering._improve5=
 \\ if x.filtered=='kept':
 \\   x.complete_at_five_prime(max_extension=15, max_query_unaligned=30, full=False)
```

The completion at 5' is performed only if a ATG is found before a stop codon, and if at most 15 codons would be added. Also, two other conditions must be met: no non-standard characters must be find in the 5' extension, and the profile query of this prediction must have an unaligned portion at N-terminal not bigger than 30 amino acids. This is to avoid completing partial hits, whose upstream ATG are not likely to be the real starts, as other large portion of coding sequence are expected upstream.
Also, normally the function stops when the first methionine is found upstream -- if the first codon is already a AUG, no extension is performed. When *full=True* is provided, it attempts instead to extend to the furthest possible methionine, when coupled with high values of *max_extension*.
The completion at the 3' is performed only if the profile query has an unaligned portion at C-terminal not bigger than 30 amino acids, if the extension is at most 10 codons, and if no strange characters are found in the candidate extension.
The use of these two methods is very limited by default, because selenoprofiles is meant to kept its nature of homology-based tool. However, their extent can easily be altered by the user through the main configuration file, as shown earlier in "searching RNA sequences or bacterial genomes".
Selenoprofiles can be customized to perform additional improvements. The user has to write a function accepting a *p2ghit* as input, and modify the main configuration file to run the function at the right step, using actions.

## Prediction program choice

After the genewise step, three predictions are available for every candidate: one by blast, one by exonerate, and one by genewise. The predictions are analyzed and only one is taken to represent this candidate gene to the filtering phase, and possibly to output. The function *choose_prediction* is used to decide among any number of candidates. This same function is used during all steps in which genes are merged to remove redundancy, to decide which one to keep. The following conditions are checked in order: if at any point only one of the predictions shows to be better than all others for a criteria, the function stops and that prediction is returned.
The first condition checked is the presence of frameshifts. If a prediction possesses frameshifts while another doesn't, the latter is taken[5].

[5] Nonetheless, blast predictions are automatically discarded if any other prediction contains frameshifts. This is necessary because blast does not predict frameshifts. Thus, when a real pseudogene with frameshifts is analyzed, the prediction choice routine would inevitably take to the blast prediction, since the others have frameshifts and blast does not.

Then, if the predictions come from a selenoprotein family, the number of aligned Sec positions is considered: if one possess more than the others, it is chosen.
The number of in-frame stop codons (others than SecTGAs) is then checked: if one possess less than the others (for example one has none, while the others have), it is chosen.
After, the length of the predicted coding sequence is determinant: the prediction featuring the longest sequence is chosen.
If at this point the choice has not been made yet, the prediction whose program has highest priority is chosen, given these priorities in descending order: genewise, exonerate, blast.
Option *-no_blast* forces selenoprofiles to choose the exonerate or genewise prediction. This is useful only if an accurate splice sites prediction is important for you. It comes at the cost that, when only the blast prediction is available (for example because exonerate produced an empty output, and genewise an invalid alignment), the candidate is always discarded.

## Labeling

After a single prediction per candidate is chosen, this is analyzed and labelled.
For standard families, there are only two possible labels: *homologue* (a regular prediction) and *pseudo* (with any in-frame stop codon or frameshift). It is possible for the user to define its own labeling procedure: this is shortly described in the option -add chapter.
For selenoprotein families, labeling is used to characterize the amino acid aligned to the Sec position. Generally there's a single Sec in selenoproteins. If there's more than one, the label assigned by selenoprofiles depends on the most-left aligned Sec position. The possible labels are *selenocysteine*, *cysteine* or any other amino acid (only rarely found at these positions though). If the prediction does not span any Sec position, it is labelled as *unaligned*. If it contains frameshifts or in-frame stop codons (apart from Sec-TGA), then it is labeled as *pseudo*. An additional label, *uga_containing*, is assigned to those predictions whose only pseudogene feature is one or more in frame UGAs (of course not aligned to Sec positions). This label is useful because very rarely the scoring schemes used for selenoprotein families allow the alignment over a non-Sec UGA, and we don't want to filter those out as if it were pseudos. Also, the label may be useful to discover new Sec positions in known selenoprotein families.

## Final filtering

After labeling, predictions are evaluated through the final filter before output. This filter, exactly as the blast filter, can be specific for each family and be written using the methods provided in selenoprofiles classes. The filter outcome is summed up in a filtering label, hereafter called "filtering state" (or just state) to differentiate it from the label assigned in the previous step. The final filter actually consists of two separate filters, called *p2g_filtering* and *p2g_refiltering* in the configuration files. A prediction excluded by the first one will be assigned a state of *filtered*. A prediction excluded by the second one will be assigned a state of *refiltered.*
Just before the predictions enter the final filter, there is an additional redundancy check: the predictions overlapping each other are compared and only the best one is kept. Predictions discarded this way are assigned a state of *redundant*.
Those predictions which passed all the redundancy check and the two steps of the final filter without being discarded are assigned a state of *kept* and represent the normal output of selenoprofiles.

Nonetheless, the user may decide to output the predictions with a different state, using the *-state* option, optionally with multiple arguments, comma separated with no space within. If for example you want to output all *filtered* and *refiltered* predicted, add to your command line:

```
-state  filtered,refiltered
```

The *-state* option can accept the following arguments: *kept*, *filtered*, *refiltered*, *redundant* or *overlapping* (see below). There is a way to have even more control on what prediction are output: the *-output_filter* option. This accepts a procedure with the same syntax of filters and actions, which is evaluated for every prediction: those for which this evaluates to *True* will be output. If for example you want to output only predictions on the positive strand, you can use:

```
-output_filter  "x.strand==‘+'"
```

To do this, you need to know a bit about the classes used in selenoprofiles, described in the advanced usage section. After filtering, results are stored in the sqlite database, ready for the output phase.

## Removing inter-family redundancy

Selenoprofiles scans for multiple profiles in a single run. The output is produced only when all families have been searched. This is because results from different profiles may overlap, especially when some of them share a certain degree of sequence similarity. So after all results are stored in the database, this is parsed and every prediction is compared with all others on the same chromosome (or contig). When two such predictions overlap, the function *choose_among_overlapping_p2gs_interfamily* is used to decide which one to keep. The other is assigned a state of *overlapping.* These predictions will not be output by default. Note that this operation is performed directly on the database: the intermediate text files written in the filtering phase will display the state previously assigned.
Another important note: the inter-family redundancy check is performed every time an output phase is run, and depends on the results present in the database at that moment. For this reason, searching several profiles in distinct selenoprofiles runs will lead to more (or the same number of) output files than searching all of them in a single run. The results database at the end will be identical, but as when every profile reached its output phase, the predictions of all other profiles were not available, the inter-family redundancy cannot be checked properly.
If you searched different profiles on separates runs, the best thing to do is just delete all output files and rerun selenoprofiles with all these profiles using *-D* flag to re-run database storage. No heavy computation will be repeated, and only the output files for the non-overlapping predictions will be produced.

## Running selenoprofiles in parallel

Selenoprofiles can be easily parallelized to be run on a large number of targets. Since the computation is independent for each target, such selenoprofiles jobs (optionally scanning for multiple profiles) can be freely split and submitted to different nodes of a computer cluster. But selenoprofiles allows also to split the computation on a single target, which is necessary if you are using it to completely annotate a genome with a comprehensive collection of protein profiles. In this case, the potential overlap of results by different profiles is a hurdle to parallelization. Thus, the strategy is not to proceed to output until results from all profiles are available. This can be accomplished by option *-stop*, or by

option -*no_db*. With -*stop*, the program will halt after having filtered and stored the results in the sqlite database. So, you can parallelize the search for each profile, using -*stop* in each such command line. Following the first example shown in this manual:

```
Selenoprofiles results_folder  -t /db/genome.fasta  -s "Macaca_mulatta" -p family1 -stop
Selenoprofiles results_folder  -t /db/genome.fasta  -s "Macaca_mulatta" -p family2 -stop
Selenoprofiles results_folder  -t /db/genome.fasta  -s "Macaca_mulatta" -p family3 -stop
Selenoprofiles results_folder  -t /db/genome.fasta  -s "Macaca_mulatta" -p family4 -stop
...
Selenoprofiles results_folder  -t /db/genome.fasta  -s "Macaca_mulatta" -p familyN -stop
```

Each of the commands above can be sent to a different node in a computer cluster. When all of them are finished, you can then run:

```
Selenoprofiles results_folder  -t /db/genome.fasta  -s "Macaca_mulatta" -p fam_all -merge
```

Assuming that the keyword *fam_all* is defined in the main configuration file as the list of all profiles, this will make selenoprofiles load all results previously computed from the database, remove inter-family overlaps, and proceed to output for all profiles.
This strategy works only if all selenoprofiles instances in the parallelized phase work until completion. If for any reason any job crashes, this may leave the sqlite database in a state that compromises the other jobs as well. If you experience database errors, you may need to cleanse the *results.sqlite* file using script *selenoprofiles_database.py,* and rerun. In the worst case, you can delete the sqlite file. As all intermediates files by slave programs are kept, the great majority of computation is never repeated anyway (unless you activated option -*clean*, which should be avoided in these cases).
Option -*no_db* provides a more robust alternative to -*stop*. When -*no_db* is active, the sqlite database is not used at all by selenoprofiles, and execution is stopped after the final filtering step. Therefore, you can parallelize the jobs as before:

```
Selenoprofiles results_folder  -t /db/genome.fasta  -s "Macaca_mulatta" -p family1 -no_db
Selenoprofiles results_folder  -t /db/genome.fasta  -s "Macaca_mulatta" -p family2 -no_db
...
Selenoprofiles results_folder  -t /db/genome.fasta  -s "Macaca_mulatta" -p familyN -no_db
```

and finally compute overlaps and output with this:

```
Selenoprofiles results_folder  -t /db/genome.fasta  -s "Macaca_mulatta" -p fam_all -merge
```

In this case, the computational time required for the last run is significantly increased, since all intermediate files need to be parsed again, and all actions have to be rerun to populate the database. Normally though, this is acceptable time-wise.

# Advanced usage

Selenoprofiles was designed to be as customizable as possible. It offers to the user the possibility of writing python code which will be integrated and run. The code can be provided mainly through the configuration file of each profile, and through the main selenoprofiles configuration file. Additionally, custom modules can be loaded using option *-add*, as we will see later.
In the simplest use of custom code, the user can set profile specific procedures, exploiting the built-in methods for filtering:

```
### fam1.fa.config
blast_filtering =  x.evalue < 1e-15
p2g_filtering =    x.awsi_filter (awsi=0.3)
p2g_refiltering =  x.coverage() > 0.5
```

With more experience, it is possible to add custom information to output, or even annotate motifs or secondary structures in the predictions:

```
### selenoprofiles config
(...)
ACTION.pre_output.see_cys=  write(x.output_id()+  "  Cys:"  +(  join([str(i)  for  i,  aa  in
enumerate(x.protein()) if aa== "C"]  or "None"  ), 1)


### output
fam1,1.homologue Cys:14,17,64,189,192
fam1,5.homologue Cys:18,21,194,197
fam1,11.pseudo Cys:60,63
fam1,19.pseudo Cys:None
```

## The *p2ghit* class

To learn how to use custom code, you need to be familiar with some variables and classes in selenoprofiles, as these are the objects that your code will be manipulating. To do this, you should have already some experience with python code and classes. The *p2ghit* class is the key of user customization. It represents a prediction of selenoprofiles, coming from any source among blast, exonerate or genewise. It contains the alignment of a query against a target, and the genomic coordinates of such alignments. Let's see its mostly used attributes and methods (for a full list, read script *selenoprofiles.py* at *class p2ghit*):

## *p2ghit class*

| Attribute or method | Description |
|---|---|
| id | The numeric id of the prediction (string). It is unique for that family and target |
| chromosome | The first word of the fasta header of the chromosome or scaffold where this prediction resides |
| strand | The strand of the prediction ( + or - ) |
| label | The label assigned to this prediction in the labeling phase |
| filtered | The filtered state assigned by the filtering phase (<br>After inter-family overlaps are computed, the state |
| output_id() | The prediction name displayed in output (profile name.index id.label). Example: |
| prediction_program() | The program that generated this prediction ( |
| query_full_name() | The full name of the query, as it appears in the profile alignment |
| coverage() | A float value, indicating how much profile is spanned by the prediction (max is 1.0) |
| protein() | Protein sequence, with * for stop codons, U for Sec |
| cds() | Nucleotide coding sequence, as ATGC characters |
| positions_summary() | A string with the positions of all exons. Examples:<br>`24-40,70-100` |
| exons | A list (array) containing the exons. Each exon is a list of 2 elements (integers), the position of start and the position of end of the coding sequence, both 1-based and included. Each prediction has at least one exon. |
| header() | A string used as default fasta header. Contains lots of non-sequence information. Example:<br>`SBP2.1.homologue chromosome:scaffold1 strand:+ positions:869-881,1163-1417 species:Polysphondylium_pallidum_PN500 target:genomes/P.pallidum/genome.fa prediction_program:exonerate` |
| dna() | Full nucleotide gene sequence, including introns and frameshifts if present. |
| splice_site_sequences() | A list of 4 letter strings, with the first two and last two nucleotides of each intron in the prediction. |
| subsequence(self, start, length) | Generic function to return any nucleotide subsequence of a prediction, using lazy computing. It can be used with negative start or large length to get the sequence around the genomic interval. Normally the indexes are relative to the predicted coding sequence, but you can use *include_introns=True* |
| alignment | Pairwise protein alignment between a profile query and the target, as an instance of the *alignment* class in |

There are plenty more of methods. Many are actually inherited from the *p2ghit* parent class, called *gene*, defined in the library *MMlib.py*.

## Custom output: option *-fasta_add*

The *-fasta_add* option represents an elegant and fast way to add information to output. A python written procedure with the same style of actions and filters must be provided as argument. The procedure is evaluated to a string which is inserted in the fasta headers of the files in output. All the fasta files in output will contain the add-on, as they all call the same function to determine the fasta header. Files with extension *fasta*, *cds*, *dna*, *three_prime*, *five_prime* and also *ali* will have it. Let's see an example. Normally the fasta headers contain the following information:

```
>GPx.6.selenocysteine chromosome:chr3 strand:- positions:
49395460-49395711,49394824-49395180 species:"Homo sapiens" target:/Genomes/
Homo_sapiens/genome.fa prediction_program:genewise
```

Let's say that you want to add the length of the protein to the header. You could add this to your command line:

```
-fasta_add '"seq_length:" +str( len(x.protein()) )'
```

Now if you run selenoprofiles with this (forcing the replacement of the old output with *-O* or specifying another output folder), you will have:

```
>GPx.6.selenocysteine chromosome:chr3 strand:- positions:
49395460-49395711,49394824-49395180 species:"Homo sapiens" target:/Genomes/
Homo_sapiens/genome.fa prediction_program:genewise seq_length:203
```

## Actions

The actions are performed during the workflow on each prediction coming from the prediction choice/labelling step. The action is provided as python code that is directly executed in the selenoprofiles environment. In a classical *for* loop, the variable *x* in the code is replaced by each *p2ghit* instance and executed. The keyword *ACTION* in the main configuration file denotes the active actions. Actions can be specified also in the command line. From now on, we will display the examples with the configuration file syntax:

```
ACTION.pre_filtering.echo = print 'hello world', x.id, x.label
```

Separating the left side with dots, the first field is the keyword *ACTION*, the second field is the category of the action and the third is the name of the action. The category determines the time point of the actions, while the name is used only to order the actions in the same category. In this example, the user will just see something like this appearing in the output of selenoprofiles:

```
...
CHOOSE: choosing among available predictions, assigning label --> selenoprofiles_results/
Polysphondylium_pallidum_PN500.genome/prediction_choice/SelI.tab (just loading file)
SelI.1    : exonerate  longest CDS predicted       unaligned
SelI.3    : blast      longest CDS predicted       unaligned
SelI.4    : exonerate  longest CDS predicted       unaligned
SelI.7    : blast      SecTGA aligned              pseudo
hello world 1 unaligned
hello world 3 unaligned
hello world 4 unaligned
hello world 7 pseudo
...
```

Each action is performed on all available prediction at a certain step of the pipeline, determined by his category. There are many possible categories of actions:

*post_blast_filter, post_blast, post_blast_merge, pre_choose, pre_filtering, post_filtering, pre_output*.

The categories names are pretty self-explanatory, but see Appendix 2 for their precise mapping in the workflow. The actions *post_blast* and *post_blast_merge* are performed on blast hits, while the others are performed on blast hits or exonerate/genewise predictions.
You will have to choose the category of your actions depending on what operation you want to perform. Actions executed during *pre_filtering* can be used to improve the predictions, but remember that their attribute .filtered is not set yet. *post_filtering* actions can access the .filtered attribute and are performed before storing results on the database. *pre_output* actions can add useful information to the log output.
Let's see an example which uses an if statement to execute operations only on a certain subset of the available predictions. Typically, the attributes that you want to check are the .label and the .filtered attributes. Let's say for example that we want to check the chromosomes and strands where the prediction with label "unaligned" rely:

```
ACTION.post_filtering.test = "if x.label=='unaligned': print x.output_id(), ' CHROMOSOME
', x.chromosome, x.strand "
```

This adds something like this in the standard output of selenoprofiles:

```
...
SelI.1.unaligned  CHROMOSOME  gi|284795330|gb|GL290990.1| +
SelI.3.unaligned  CHROMOSOME  gi|284795323|gb|GL290997.1| +
SelI.4.unaligned  CHROMOSOME  gi|284795338|gb|GL290984.1| -
...
```

The next action is for giving a quick look to the protein sequence of all discarded predictions. Below is the output added.

```
ACTION.post_filtering.check_ali = "if x.filtered != 'kept': print x.output_id(),
x.protein()"

...
SelI.4.unaligned ITLVGLFCNIAMYLIVYFQCPGLTEPAPRWCYFLIAFLIFAYQTLDNLDGKQARRTKSSSPLGELFDHCCDA
SelI.7.pseudo VTATGFVCNFIALFLMSSYMRPVNDGQEPV
...
```

After the *post_filtering* actions are performed, the results are stored in the selenoprofiles database. Remember that if selenoprofiles finds the results in the database, it does not perform the steps up to filtering. Therefore beware that if you specify actions of category pre or post filtering (or any of the categories before them) on a second run of selenoprofiles, it won't perform them unless you force the proper routine, for example with option -F to force the filtering routine. *pre_output* actions, on the contrary, are performed both if in the current run results are produced or loaded from the database, but only on the results which are output (determined by the *-state* option).
Later, we will see how actions can be used to correct gene structures, or to add custom genomic features to the predictions.

## Blast filtering

There are 3 layers of filtering in selenoprofiles, all regulated by procedures defined in the profile. We have already seen them: blast filtering, p2g filtering and refiltering. The same grammar applies to all of them. For blast filtering, the most common attribute checked is the *evalue*, an attribute specific of blast hits. The blast hit is a subclass of *p2ghit* and has

the same methods. Let's see a simple blast filtering procedure as written in a profile configuration file; this accepts only the blast hits with *evalue* minor (better) than 1e-5:

```
blast_filtering =  x.evalue < 1e-5
```

Selenoprofiles offers also more sophisticated tools, which map the prediction back to profile alignment to use what we know from the profile alignment. For example many families possess N-terminal regions of disordered or repetitive sequence, which hits spuriously many regions in the genome. The resulting blast hits span only the initial portion of the profile.
You may want to exclude those, using function *is_contained_in_profile_range*:

```
blast_filtering =  x.evalue < 1e-5 and not x.is_contained_in_profile_range(1, 35)
```

The similar function *spans_profile_range* asks whether the predictions spans certain columns of the alignment, useful when you want only proteins with a certain conserved domain.

```
blast_filtering =  x.evalue < 1e-5 and x.spans_profile_range(50, 60)
```

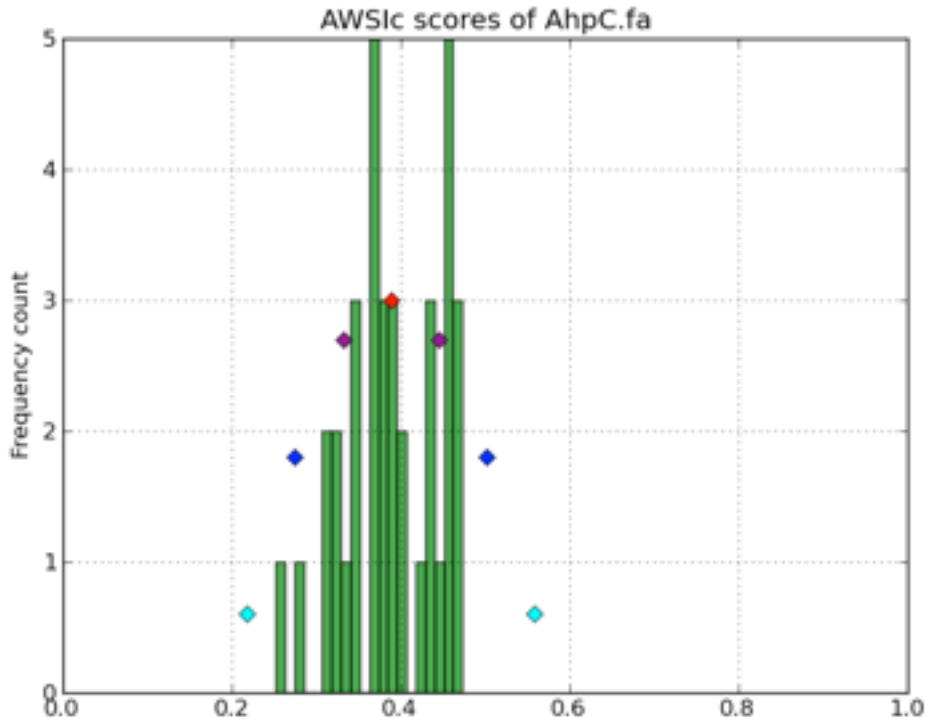
The function *show_conservation_in_profile_range* is useful when dealing with blast filtering of profiles with regions of low information. It checks the number of pairwise similarities (defined as positive scores in the BLOSUM62 matrix) between the amino acids in the query and in the target in the prediction along a certain profile range. In the example below, predictions are required to have 3 conserved amino acids in the region from positions 1 to 50.

```
blast_filtering  = x.show_conservation_in_profile_range(1, 50, 3)
```

## AWSI Z-score based filtering

We developed various method to score how much a sequence "fits" in a protein profile. We called the best performing one Average Weighted Sequence Identity (AWSI).
It is based on the Weighted Sequence Identity (WSI), a scoring method for comparison of two sequences, with one of the two belonging to a profile alignment.
The WSI score is computed as an average of sequence identities with different weights on the different columns of the profiles. In the pairwise comparison between the profile sequence and the candidate sequence, the weight is given by the representation of the amino acid in this profile sequence and column across all the profile. More conserved columns are given more weight thus more importance. This weight is also multiplied by the column coverage, that is to say, the total number of characters which are not gaps divided by the total number of profile sequences. In this way, the alignment regions present only in a small subset of sequences have less importance.
When the term AWSI is used in this manual, we refer to the variant AWSIc, computed as just explained. There is another variant (AWSIw), which is computed in the same way, but the weight is not multiplied by the column coverage.
When comparing a candidate sequence against a profile, a WSI for each profile sequence is calculated. Each one ranges from 0 to 1, as it is normalized to the sum of weights in that WSI. Now the AWSI of the candidate sequence is just the average of all computed WSI.
Although the range of AWSI is also between 0 and 1, the maximum value it can assume is constrained by the profile characteristics. In a profile with very dissimilar sequences, no candidate sequence can reach high scores (as if it matches a sequence of the profile, it cannot match the different ones at the same time). Thus, it is useful to adjust the AWSI threshold for each profile.

For this purpose, profile alignments are analyzed when used for the first time, and AWSI values for all sequences are computed. For each profile sequence, we compute its AWSI as explained above, considering this sequence as a candidate, and the rest of sequences as the profile to compare against.
The distribution of these AWSI scores is used to decide the similarity threshold when fitting a sequence into this particular profile. The AWSI score of the target sequence is fit in a Gaussian distribution with the profile average and standard deviation, and a Z-score is computed. In the default p2g refiltering procedure (*awsi_filter*), the Z-score must be greater than -3.
The script *selenoprofiles_build_profile.py* can be used to display the distribution of the AWSI scores with option *-d*, as shown here above (pylab must be installed). The frequencies of the computed AWSI values are shown as green columns, while colored dots are used to display the approximated gaussian distribution: the red dot is the average, while the purple, blue and cyan dots correspond to the average minus 1, 2, 3 standard deviation respectively. The default cut-off point is thus indicated by the leftmost cyan dot.




The methods of the *p2ghit* class relevant to AWSI scores are:

- awsi() with no arguments, it returns the AWSIc value for this candidate. Used as *awsi(with_coverage=False)*, returns AWSIw instead

- awsi_z_score() returns the z-score compute comparing the AWSI of this candidate with the profile distribution. This function also accepts the *with_coverage=False* switch to return AWSIw instead.

- awsi_filter() returns *True* if the prediction would pass the default AWSI-based filtering, *False* otherwise. This function also accepts the *with_coverage=False* switch to return AWSIw instead. This is normally computed just as *awsi_z_score()>-3*, with two possible exceptions. For extremely conserved profiles, the cut-off threshold would be generally too strict. Thus, if the candidate has an extremely high AWSI (>0.9), it is accepted

regardless of the profile characteristics. The second exception is for profiles with few sequences (<3). In these case, the computed AWSI standard deviation is always zero or extremely close to it, and this would also result in filtering too strict. Thus, for these profiles the filter just checks that *awsi()>=0.3*

One can easily alter the filter behavior using any of these arguments to the *awsi_filter* function: *z_score*, *awsi*, *few_sequences_awsi.* For example awsi_filter(awsi=0.5) accepts any candidate scoring a AWSI with the profile of 0.5 or greater (or a *z_score* >-3).

## Other filtering functions

Here's some other methods useful for blast or p2g filtering of specific families.
The function *seq_in_profile_pos* provides the amino acid predicted in the target at a certain position of the profile alignment (may be - for unaligned). It can be used to check that certain domains are complete (e.g. redox boxes CXXC).

```
p2g_refiltering = x.seq_in_profile_pos(31)== 'C' and x.seq_in_profile_pos(34)== 'C'
```

The function *sequence_identity_with_profile* computes a quantitative measure of how much the prediction fits in the profile: it computes the sequence identity of the prediction with every profile sequence, and average them. It is a simplification of the AWSI score. With no arguments, internal (but not terminal) gaps are counted as mismatches. The choice of the threshold in this case depends largely on the profile.

```
p2g_refiltering = x.label!='pseudo' and x.sequence_identity_with_profile()>=0.25
```

The more useful function *sequence_identity_in_range* is analogous the previous one, but computes the average sequence identity only on a certain range of the profile. Predictions not spanning this region are given 0.

```
p2g_refiltering = x.label!='pseudo' and x.sequence_identity_in_range(40, 80)>=0.35
```

For a full list of the methods of the *p2ghit* class, run *selenoprofiles_3.py* with *-help full* or inspect the script inside your installation directory.

## Tag blast filtering

Tag blast is an implemented form of filtering. This consists in searching the protein sequence predicted in the target against a comprehensive protein database (typically nr - non redundant proteins at NCBI). The output generally provides a good annotation of the protein in question. Note that your profile may match sequences in the genome that are real genes, but do not belong to your family and are hit because of their sequence similarity. These predictions usually have blast hits against proteins in nr which are not in your protein family. Tag blast utilizes a set of profile-defined tags to scan the titles in the blast output and assign a score to the prediction. A predicted sequence that resembles proteins not belonging to the family are likely to be spurious, and will be assigned a negative tag score. To use tag blast, you must first set the list of tags for your profile in its configuration file. Tags are strings which are interpreted as perl regular expressions. In the configuration file of the profile, the tags are written as a python list of strings:

```
tags = ['SecS ', '(Sec|selenocysteine|tRNA).* selenium transferase']
```

Tags should be carefully designed in order to recognize all sequences of the profile and those with similar names. For each blast hit appearing in the blast file, the tags are tested and a score is assigned to the title. Its absolute value is the negative logarithm of the

*evalue*: a blast hit with *evalue* 1e-5 gets 5 points. The final tag score assigned to prediction is the sum of all the titles. If the title matches any profile tag, its score will be positive. If it matches any neutral tag, its score will be zero. If a title does not match any profile or neutral tag, its score will be negative. The neutral tags are used to skip all the blast hits with uninformative titles and those based only on computational prediction. The neutral tags are defined in your main configuration file, with a decent default value. For filtering, we check whether the final tag score assigned to predictions is positive:

```
p2g_refiltering =       x.label!='pseudo'  and x.tag_score() > 0
```

If you want to use the tag score in a filter, we suggest you to inspect manually the results and check their tag score first. For example with this action (paste it in the main configuration file):

```
ACTION.post_filtering.check_score = print “Tag score of”, x.output_id()+” filtered: “+x.filtered+”\n”+str(x.tag_score(verbose=1))
```
[6]

The verbose mode will allow you to check the titles of all proteins present in the blast output and the score assigned to them. This will allow you to build and improve useful tags for your family.
When the method *tag_score* is run for the first time on a *p2ghit*, blastp is run against the database defined in the profile or in the main configuration file (under the keyword *tag_db*). The output file is kept in the *tag_blast* subfolder inside the folder dedicated to this target. A tag blast run takes a few minutes, so take care of avoid doing it on a lot of hits. If you put the *tag_score* evaluation on the right side of an *and* construct, the tag blast will not be performed unless all conditions to his left are true:

```
p2g_refiltering = x.coverage()>0.4 and x.label!=‘pseudo’ and x.tag_score()>0
```

## GO score filtering

Similarly to the tag score, the GO score utilizes the same blast search against nr, but in this case it is the GO terms associated to the proteins found which are evaluated. A list of the positive GO terms is to be provided in the profile configuration file:

```
go_terms = ["GO:08028", "GO:08030"]
```

A score is assigned to each blast hit depending on the *evalue*, as in the tag score. The GO terms are searched considering their hierarchy: if for a certain title in the blast output, a GO term is found which is a child of a GO term defined in the profile configuration, this will count as positive. Blast hit with no annotated GO are scored neutral. Only molecular functions GO terms are checked.

```
p2g_refiltering = x.label!='pseudo' and x.go_score()>0
```

## Integrate your own code: option *-add*

With the *-add* option, you can provide a python add-on file that will be loaded in selenoprofiles. This will allow you to define functions can then be used in any procedure, for example for filtering or output. The code inside the file provided is read line by line and executed in selenoprofiles when all variables are already loaded and everything is ready to run.

---

[6] the str( ) function is necessary to convert the integer returned by tag_score into a string that can be concatenated and printed

User defined functions are useful for filtering, labeling or outputing. Let's see how to create a simple output function. Create a file called *extension.py* where you define function which accept a *p2ghit*:

```
def my_name_is(z):
       """This functions accepts a p2ghit and returns its output id """
       return z.output_id()
```

If you now you provide this file with the option *-add*, the function *my_name_is* will be available in selenoprofiles. Running selenoprofiles with:

```
-add extension.py -ACTION.pre_output.test "print my_name_is(x)"
```

you will have something like this in the output:

```
...
SelI.1.unaligned
SelI.3.unaligned
SelI.4.unaligned
...
```

Let's see a more relevant example. Assume that for some reason you are interested only in the non-pseudo, single-exon predictions. You could then write this function in your *extension.py*:

```
def has_no_introns(z):
       “”” This functions accepts a p2ghit and returns True if it has no introns “””
       return len(z.exons)==1
```

You may then use this function for filtering, adding something like this in your profile *.config* file:

```
p2g_refiltering =  x.label!= "pseudo" and has_no_introns(x)
```

Adding functions may be useful for several purposes. It is possible to write procedures to improve the predictions, as those previously presented, or for filtering, as shown above. It can also be used to perform one-time operations (for example to load custom data), or override some functions or attributes used in selenoprofiles. For example, the user may want to customize the labeling procedure used in selenoprofiles. The easiest way to do this is writing a new labeling procedure in the *extension.py* file, which redefines the *.label* attribute of the input *p2ghit*, and use it in a *pre_filtering* action. In this example, we define a procedure to label the predictions as short or long, checking their predicted protein length:

```
def  custom_labelling(z):
       if len(z.protein()) >= 50:   z.label=‘long’
       else:                        z.label=‘short’
```

We activate this by adding this action in the main configuration file:

```
ACTION.pre_filtering.labelling =  custom_labelling(x)
```

Note that when the new function is called, the standard labeling procedure has been already called, so a *.label* attribute is available, and you can check it (or use it) to define the new label. Example:

```
def  custom_labelling(z):
       original_label=z.label
       if len(z.protein()) >= 50:   z.label=‘long_’+original_label
       else:                        z.label=‘short_’+original_label
```

The label is then typically used for filtering:

```
p2g_refiltering =  x.label.startswith("long")
```

There are a few global functions in selenoprofiles that user may be interested in altering. In various steps of the workflow, the program must decide which gene structure prediction is best among 2 or more candidates. The first such function is named *choose_prediction*. This is used in the prediction choice step, when a single prediction among blast, exonerate and genewise is chosen. It accepts a list of *p2ghit,* with variable length (1-3). It returns a tuple like (*p*, *s*) where *p* is the chosen *p2ghit* and *s* is a string with a reason why (it will be printed and stored in a file). The native function is the quite complex, and takes into account the presence of frameshifts, presence of stop codons, aligned Sec position (for selenoprotein families), length of coding sequence (you can inspect the code at *def choose_prediction* in *selenoprofiles.py*). Let's see an example in which this function is replaced by a simple hierarchal function, choosing predictions by genewise over those by exonerate, over those by blast (note that it is still possible that even blast is chosen in this way, if for a given hit the exonerate and genewise predictions are empty or non-valid). Put this into your *extension.py* file provided to option *-add*:

```
global choose_prediction
def choose_prediction(candidates):
	for c in candidates:
		if c.prediction_program()=='genewise': return ( c, 'genewise is available')
	for c in candidates:
		if c.prediction_program()=='exonerate': return ( c, 'exonerate is 2nd best')
	return (candidates[0], 'only blast available')
```

When writing a new *choose_prediction* function, you may still want to call internally the old function, which you can refer to as *choose_prediction_selenoprofiles.* In this example, the new function keeps the behavior of the old one, except for blast predictions which are forced to be never chosen. This is accomplished by returning an *empty_p2g*() object when only blast is available.

```
global choose_prediction
def choose_prediction(candidates):
	if all( [ c.prediction_program()=='blast'    for c in candidates ] ):
		return empty_p2g(), 'excluding blast'
	else:
		return choose_prediction_selenoprofiles(candidates)
```

The second such function is named *choose_among_overlapping_p2gs_intrafamily* and is used when removing intrafamily redundancy. This accepts two *p2ghit* that were found overlapping and returns the best one, which is kept. The default function calls internally *choose_prediction.* In its code, this is named *choose_prediction_selenoprofiles*, so if you override the *choose_prediction*, the *choose_among_overlapping_p2gs_intrafamily* function will still run the original, built-in procedure.
If you want to remove intrafamily redundancy using an overridden *choose_prediction* function, it is necessary to override *choose_among_overlapping_p2gs_intrafamily* too. You can search its code in *selenoprofiles_3.py* as a template.
The third and last function is named *choose_among_overlapping_p2gs_interfamily* and is used when removing redundancy between gene predictions by various profiles. This also accepts two *p2ghit* and returns one. The default function considers the AWSI score of the candidate with the 2 profiles, and their filtered attribute (a prediction kept by a profile is never masked by an overlapping prediction filtered by another profile). Let's see how to

replace it with a function which always keeps the prediction with longer protein sequence. Create an *extension.py* file like this:

```
global choose_among_overlapping_p2gs_rem_red
def choose_among_overlapping_p2gs_rem_red(p2g_hit_A, p2g_hit_B):
       if len(p2g_hit_A.protein()) >  len(p2g_hit_B.protein()): return p2g_hit_A
       elif len(p2g_hit_A.protein()) <  len(p2g_hit_B.protein()): return p2g_hit_B
       else: return  p2g_hit_A
```

If you believe that your own function may be useful to other users, or if you need help building your own function, feel free to contact me (see email on the cover page).

## Custom prediction features

Selenoprofiles offers the possibility to annotate and manipulate custom features linked to gene predictions. Such annotations (*p2g_features*) can be used for example for protein motifs or domains, or signal sequences, or secondary structures, present in all or some gene predictions. Within selenoprofiles, SECIS elements are implemented as *p2g_features*. Technically, *p2g_feature* is a python class, thought to be generic so the user can created a child-class (subclass) to adapt it to his specific purpose.
Selenoprofiles includes a built-in example to show the capabilities of *p2g_features*: the class *protein_motif*. This is thought to annotate a short motif within the protein sequence, the redox box, expressed as the perl-like regular expression *C..C* (*C* stands for cysteine, and *.* means any character). The class *protein_motif* allows to detect these motifs and easily integrate them in the p2g or gff output.
For any custom *p2g_feature,* the user has to define at least the following procedures: how to search and assign these features, how to dump them in the sqlite database, how to load them back. Then, optionally one can define how to output them to the gff and/or p2g file, and also how to reload the features if gene structure predictions are modified. The *protein_motif* includes examples of all these procedures.

All the code relevant to the *protein_motif* is here below, copied from *selenoprofiles_3.py*.

```
def annotate_protein_motif(p, silent=False):
  """p is a p2ghit. This is an example of method to annotate the p2g_feature protein_motif. To use,
add this to the main configuration file:
  ACTION.post_filtering.annotate_motif =    "if x.filtered == 'kept':  annotate_protein_motif(x)"
  """
  s= protein_motif.motif.search(  p.protein() )   ##using search method of re.RegexObject  --
protein_motif.motif is such an object
  while s:
    protein_motif_instance=         protein_motif()
    protein_motif_instance.start=   s.start()+1   #making 1 based
    protein_motif_instance.end=     s.end()       #making 1 based and included, so it'd be +1-1
    protein_motif_instance.sequence=     \\
                p.protein() [ protein_motif_instance.start-1 : protein_motif_instance.end ]
    p.features.append(protein_motif_instance)     ## adding feature to p2g object
    if not silent:  printerr('annotate_protein_motif found a motif: ' \\
                             +protein_motif_instance.output()+' in prediction: '+p.output_id(), 1)
    s=protein_motif.motif.search(  p.protein(), pos= s.start()+1 ) ## searching again, starting from
just right of the previously found position



class protein_motif(p2g_feature):
  """ protein motif is an example of a p2g_feature, to annotate the positions of a certain motif
defined as a perl-style regexp.  The motif is defined in the line following this, as a class
attribute. In the example, the redox box (CXXC) is the motif.
      Attributes:
      - start      start of the protein motif in the protein sequence (1-based, included)
      - end        end of protein motif in the protein sequence (1-based, included)
      - sequence   motif sequence
  """
  motif=re.compile( 'C..C' )
  included_in_output=True
  included_in_gff=   True

  def dump_text(self):
    """ Returns a string with all the information for this feature. This string is stored in the
sqlite database. """
    return str(self.start)+':'+str(self.end)+':'+self.sequence

  def load_dumped_text(self, txt):
    """ Reverse the dump_text method: gets a string as input, and loads the self object with the
information found in that string. """
    start, end, sequence= txt.split(':')
    self.start= int(start);    self.end=int(end);    self.sequence=sequence

  def output(self):
    """ Returns a string. This will be added to the p2g output of the prediction to which this
feature is linked -- if class attribute included_in_output is True"""
    return 'Motif: '+self.sequence+' Start: '+str(self.start)+' End: '+str(self.end)

  def gff(self, **keyargs):
    """This must return a gff-like tab-separated string. In this case, we are exploiting and
overriding the gff method of the gene class, which is a parent class for p2g_feature"""
    ## getting a gene object with the genomic coordinates of the protein motif. we use the gene
method subseq, which returns a subsequence of the parent gene. Indexes are adjusted for protein-
nucleotide conversion
    motif_gene_object= self.parent.subseq( start_subseq= (self.start-1)*3 +1,    \\
                                           length_subseq=(self.end-self.start+1)*3, minimal=True )
    #now motif_gene_object has a .exons attributes with the genomic coordinates of the protein
motif. now we can use the native gff method of the obtained gene object
    return gene.gff(motif_gene_object,    **keyargs)

  def reset(self):
    """ This method is called when the linked prediction is modified, to allow to recompute some or
all attributes of the feature. In this case, we are removing all features of this class, and
annotating them again with the same method used to add them in first place:
annotate_protein_motif"""
    ##removing instances of this class
    for index_to_remove in \\
     [i  for i, f in enumerate(self.parent.features) if f.__class__ == protein_motif ]  [::-1]: \\
        self.parent.features.pop(index_to_remove)
    #reannotating
    annotate_protein_motif( self.parent, silent=True )
```

The code contains the definition of a class (*protein_motif*, including 5 methods), and the function *annotate_protein_motif*. This function takes as input a *p2ghit* instance, analyzes it, and if any protein motif is found, it populates its *.features* attribute with one *protein_motif* instance for each motif found.
If this function is never run, the *protein_motif* class is unused. As mentioned within the code, to activate it you should add this line to the main configuration file:

```
ACTION.post_filtering.annotate_motif =   if x.filtered == 'kept': annotate_protein_motif(x)
```

In this way, the *annotate_protein_motif* will be run on every prediction that passed filtering. The protein motif *C..C* is defined as the class attribute *motif*, which is of type *RegexObject* from the pattern matching module *re*. Inside the *annotate_protein_motif* function, it is searched in the predicted protein sequence its dedicated method *search*. For each motif found, a *protein_motif* instance is created, and the start and end positions of the match are stored within this object; the protein sequence of the motif is also derived and stored. Once the *protein_motif* instance is ready, it is appended to the *.features* list attribute of the input *p2ghit*. Shortly after, this *p2ghit* reaches the database step, and its information is stored as a sqlite entry. All the features associated to it are also stored in the database. For this reason, the method *dump_text* is called on every feature instance. This method must return a string containing all the information sufficient to then load it back. The method *load_dumped_text* is its reverse, and is used during the output phase to load the dumped information from the database into an empty *protein_motif* instance. An annotating function (in this case *annotation_protein_motif*), and the *p2g_feature* class methods *dump_text* and *load_dumped_text* are the minimal set of definitions to make a functional feature. Other attributes and methods can be used to output the features.To output features to the native selenoprofiles format (*.p2g*, previously illustrated), the class attribute *included_in_output* must be *True*, and the *output* method has to be defined. Features can be used for gff output too, if the class attribute *included_in_gff* is set to *True*. In this case, it makes sense to take advantage of the *gene* class, the parent of both classes *p2ghit* and *p2g_feature.* The *gene* object is designed to represent a genomic interval, optionally composed by multiple exons, on a certain chromosome (or scaffold) of a target file. It provides plenty of methods such as for fasta fetching, cutting subsequences, computing overlaps, merging gene structures and so on. Its native *gff* method returns one line for each exon in the object, reporting its coordinates and optionally other attributes. In the example above, the *protein_motif* class is not really used as a *gene* object, but just as a data container for the attributes *start*, *end*, *sequence*: its attributes *chromosome*, *strand*, *exons* are not used. Instead, the correct genomic coordinates of the protein motif are derived dynamically, and added to output by overriding the native *gff* method of the class gene. For each motif instance, its start and end positions relative to the full protein sequence are available. Thus, the *gene* method *subseq* is used to derive the global genomic coordinates of the motif. This function accepts as input a *gene* (self) object, a start position and a region length, and returns another *gene* object, which contains a subset of the genomic intervals in the self object. If the desired region spans any exon boundary, the returned object contains multiple exons. In the code, the indexes are adjusted for converting protein-based to nucleotide-based positions. Once the appropriate gene object containing the global genomic coordinates for the motif is ready (*motif_gene_object*), the native *gene* class *gff* method can be called.
Lastly, the method *reset* can be defined for custom features that have to be recomputed when the predictions are modified, by actions such as those explained in improving predictions. In the example, the *protein_motif* instances are searched and expelled from the *features* list of the *p2ghit* object for which the *reset* function is run. Then, the annotating function *annotate_protein_motif* is run again.

# Appendix 1: guide to profile building

Building good profiles is of key importance for the accuracy of predictions. Their sensitivity and specificity mostly depends on their sequence variation (many representatives for a family are better than few), and on the filters used. The best way to build good profiles is to progressively tune them by inspecting results. If you plan to search a large number of genomes, it is a good routine to begin with just a few of them to get the profile right. First thing on the checklist is the number of processed blast hits. If there are thousands, you should tighten up the blast filtering procedure. Then, ideally the genes in output should be inspected, to see if they fit your expectations.
You can parse log files for *OK* tags, indicating an output gene, or *DROPPED* tags, that denotes predictions discarded by the filter, as well as for *WARNING* or *ERROR* tags to see if everything went fine. Then, the programs *selenoprofiles_join_alignments* and *selenoprofiles_tree_drawer* constitute useful tools to collect and visualize results.
If there are too many genes in output, or too few, try and change the filtering procedures. By default, the stringency of a profile depends on the distribution of the AWSI scores of its sequences, which measure how similar its sequences are among themselves. For each candidate result, a AWSI score is computed and compared with the profile distribution, computing a Z-score which must be greater than -3 to pass the filter. A simple way to control the stringency of a profile is to alter the minimum Z-score of its filtering procedure:

```
p2g_refiltering =   x.awsi_filter(z_score=-5)
```

Using the AWSI Z-score, profiles with very similar sequences accept only results which are also very similar, while broader profiles are more loosely filtered. Thus, a good profile should possess an amount of sequence variation which is not too low, nor too high. As a rule of thumb, profiles should contain more than ten sequences, but no more than a hundred. The script *selenoprofiles_build_profiles* can be used with option *-r* to remove redundancy in an input alignment, in order to trim large profiles to an acceptable number of sequences. The same script can be used with option *-d* to inspect the AWSI distribution of a profile. Generally the profiles with AWSI cut-offs between 0.2 and 0.6 work reasonably well. If the cut-off is higher, it means that the profile is extremely conserved, and thus will output only extremely similar candidates. In this case stringency can be lowered by setting manually a AWSI cut-off independent of the Z-score. The same *awsi_filter* function can be used, as it accepts also a AWSI threshold: a candidate is accepted if either the AWSI or the Z-score are higher than the respective thresholds.

```
p2g_refiltering =   x.awsi_filter(awsi=0.5)
```

If the default AWSI cut-off is very low, it means that the profile is too broad, containing sequences too dissimilar to each other. If large, the best strategy is generally to split the input alignment into two or more profile alignments. Alternatively, one can try to keep the profile as it is, and set an efficient filter using the tools explained in this manual.
A useful filtering tool is the coverage: the prediction is mapped into the profile, and the distance between its projected boundaries, divided by the profile alignment length gives the coverage. A strict coverage filter excludes partial protein predictions:

```
p2g_refiltering = x.coverage()>0.75
```

When you are searching for protein families containing of common domains, you may want to exclude the hits limited to these protein regions, using again the positions of the prediction mapped to the profile:

```
p2g_refiltering = not x.is_contained_in_profile_range(1, 60) and not
                                        \\     x.is_contained_in_profile_range(100, 160)
```

The tag and GO score are powerful tools to allow to discriminate even between similar protein families. Both tag and GO score procedures require a run of blastp against nr, and thus are quite computationally expensive. For this reason, they should be used only for the most difficult profiles, for which the AWSI score is not enough to differentiate bona-fide genes and spurious hits. Even then, it is worth to additionally limit the number of results for which this is run, for example checking AWSI. In this example, all results with AWSI greater than 0.6 automatically pass the filter, while for those with AWSI between 0.2 and 0.6 the *go_score* is evaluated.

```
p2g_refiltering = x.awsi()>0.6 or ( x.awsi()>0.2 and x.go_score()>0 )
```

The tags should be written by searching the results with blastp against nr and looking at protein titles. For GO scoring, the script *selenoprofiles_build_profiles* provides a utility to find suitable terms, if the input profile sequences contain gi codes from NCBI nr. The GO annotations for all profile sequences are fetched, and their number is compared with the total number of proteins for each GO term.

# Appendix 2: full list of operations

Load variables and functions:
    Read configuration file
    Read command line
    Check presence of target file and profiles
    Check/convert species name
    Initialize results database if necessary
    Read active actions
    Read parameters
Load file provided with -add option
Load/compute length of all chromosomes in the target file
For each input profile:
    Load/compute clusters of profile alignment
    Check if results are already in database. If so, skipping all these steps:
        For each cluster:
            Run/load psitblastn
            For each blast hit in the blast output for this cluster:
                Transform it to have it relative to the master blast query
                Replace "*" with U in the target sequence if a UGA is aligned to a Sec position
                Perform pre_blast_filter actions
                Evaluate if blast hit passes blast filtering. If it doesn't, discard it
                Perform post_blast actions
        If more than one cluster:  merge overlapping blast hits from the different cluster searches
        Merge blast hits by colinearity
        For each blast hit:  Perform post_blast_merge actions
        For each blast hit:  Run/load exonerate using blast hit as seed
        Discard duplicated exonerate hits and the blast hits associated to them
        For each blast hit:
            If an exonerate hit is available: run/load genewise using it as seed
            Else: run/load genewise using the blast hit as seed (genewise_to_be_sure routine)
        For each blast hit:
            For each non-empty prediction among blast, exonerate, genewise:
                Perform pre_choose actions
        Check if the choose prediction output file is already present. If not:
                Choose a prediction among the available ones: blast, exonerate, genewise
                Assign label to the chosen prediction
            Write choose prediction output file
        For each prediction:  Perform pre_filtering actions
        Check if the filtering predictions output file is already present. If not:
            Determining the overlap between predictions
            For each prediction:
                If the prediction overlaps an identical or smaller prediction, filter it as "redundant"
                Else, evaluating p2g_filtering. If it doesn't pass, filter prediction as "filtered"
                Else, evaluating p2g_refiltering. If it doesn't pass, filter prediction as "refiltered"
                Else: filter prediction as "kept"
            Writing filtering predictions output file
        For each prediction:  Perform post_filtering actions
        Write predictions (including their filtered state) in the database
Checking if results from different families overlap each other. Filtering those as "overlapping"
For each input profile:
    Computing list of predictions to be output (based on output states / output filter)
    For each prediction to be output:
        Perform pre_output actions
        For each active output format:
            If the output file is not already present: write output file
    Write alignment output (with all predictions to be output along with profile sequences)

# Appendix 3: links and references

**Selenoprofiles:**
Mariotti M, Guigó R. Selenoprofiles: profile-based scanning of eukaryotic genome sequences for selenoprotein genes. Bioinformatics. 2010 Nov 1;26(21):2656-63
website: http://big.crg.cat/services/selenoprofiles

**Blast:**
Altschul SF, Madden TL, Schäffer AA, Zhang J, Zhang Z, Miller W, Lipman DJ. Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. Nucleic Acids Res. 1997 Sep 1;25(17):3389-402. Review.
installation: ftp://ftp.ncbi.nlm.nih.gov/blast/executables/release/LATEST/

**Exonerate:**
Slater GS, Birney E. Automated generation of heuristics for biological sequence comparison. BMC Bioinformatics. 2005 Feb 15;6:31.
website: http://www.ebi.ac.uk/~guy/exonerate/

**Genewise:**
Birney E, Clamp M, Durbin R. GeneWise and Genomewise. Genome Res. 2004 May; 14(5):988-95.
website: http://www.ebi.ac.uk/Tools/Wise2/
installation: ftp://ftp.ebi.ac.uk/pub/software/unix/wise2/wise2.2.0.tar.gz

**NCBI protein databases:**
search: http://www.ncbi.nlm.nih.gov/sites/entrez?db=protein&itool=toolbar
download: ftp://ftp.ncbi.nih.gov/blast/db/FASTA/nr.gz

**Gene ontology:**
website; http://www.geneontology.org/
The python code to query the gene ontology used in selenoprofiles is partially from:
http://gitorious.org/annotation/annotation/trees/master.
which is an adaptation by François Serra of the code by Nepusz Tamás (**thanks** to both!)
https://github.com/ntamas/biopython

**MAFFT alignment program:**
Katoh K, Asimenos G, Toh H. Multiple alignment of DNA sequences with MAFFT.
Methods Mol Biol. 2009;537:39-64.
website: http://mafft.cbrc.jp/alignment/software/

**ETE2 tree visualization:**
Huerta-Cepas J, Dopazo J, Gabaldón T. ETE: a python Environment for Tree Exploration. BMC Bioinformatics 2010, 11:24.
website: http://ete.cgenomics.org/;

**Pylab graph visualization:**
website: http://www.scipy.org/PyLab

**SECISearch3:**
Mariotti M, Lobanov AV, Guigó R, Gladyshev VN. SECISearch3 and Seblastian: new tools for prediction of SECIS elements and selenoproteins. Nucleic Acids Res. 2013; manuscript in publication.
website: http://seblastian.crg.es/ or http://gladyshevlab.org/SelenoproteinPredictionServer/

# Appendix 4: troubleshooting

Here's some errors that I experienced often installing selenoprofiles and the required slave programs in different systems. If you have selenoprofiles errors which are not reported here, contact me (see email address in the cover page).

## Blast error

Selenoprofiles runs the *blastpgp* binary (to build a PSSM for each profile) through symbolic links in its installation directory. In some systems this may cause this error:

```
[blastpgp] WARNING: Unable to open BLOSUM62
[blastpgp] WARNING: BlastScoreBlkMatFill returned non-zero status
[blastpgp] WARNING: SetUpBlastSearch failed.
```

Blast cannot find the BLOSUM62 matrix, that is to say, its installation data folder. To fix the problem, edit (or create) the file ~/.ncbirc and add something like this to its content:

```
[NCBI]
data=/path_to_blast_installation/blast-2.2.2x/data
```

To know what is you blast installation folder, use the *which* command in bash (e.g. *which blastpgp*) and follow possible symbolic links until you have something like:

```
/path_to_blast_installation/ncbi_blast-2.2.2x/bin/blastpgp
```

The data folder to insert in ~/.ncbirc is then the one shown above.

## Genewise errors

Genewise is part of the wise2 package that can be found here (newer versions may exist):
ftp://ftp.ebi.ac.uk/pub/software/unix/wise2/wise2.2.0.tar.gz
In some systems, an error appears as you build the program with *make*:

```
sqio.c:232: error: conflicting types for 'getline'
/usr/include/stdio.h:653: note: previous declaration of 'getline' was here
make[1]: *** [sqio.o] Error 1
make[1]: Leaving directory `/PATH/src/HMMer2'
make: *** [realall] Error 2
```

The problem is in a function declaration (*getline*) in the file HMMer2/sqio.c, since this function is already declared in many compilers. To solve it, type:

```
cd wise2.2.0/src/HMMer2/
sed 's/getline/getline_new/' sqio.c  > a &&  mv a sqio.c
```

Now get back to wise2.2.0/src/ and type *make all*. Take care of the final message it shows: you need to set the environmental variable *WISECONFIGDIR* to point to right place for genewise to work. If you do not, you may have the following error:

```
Warning Error    Could not open human.gf as a genefrequency file
Warning Error    Could not read a GeneFrequency file in human.gf
Fatal Error    Could not build objects!
```

To take care of this, add to your bash configuration file ~/.bashrc something like this:

```
export WISECONFIGDIR=/path_to_installation/wise2.2.0/wisecfg/
```

so this will be executed for every bash instance you will run from now on.