`fur` 3.1 : Find Unique Genome Regions

https://github.com/haubold/fur

Bernhard Haubold

June 9, 2023

Contents

List of Programs

Chapter 1

Make `fur` Database, `makeFurDb`

1.1 Introduction

The program `fur`[1] requires a database to run, which is computed with `makeFurDb`. `MakeFurDb` takes as input a directory containing the target genomes and a directory containing the neighbor genomes. It generates a directory containing the `macle` index and the BLAST database required by `fur`.

1.2 Implementation

The program consists of an include section, function declarations and definitions, and the main function.

Program 1.1 (`makeFurDb`)**.**

1a ⟨*makeFurDb.c* 1a⟩≡
⟨*Include headers, P. 1.1* 2b⟩
⟨*Function declarations, P. 1.1* 3c⟩
⟨*Function definitions, P. 1.1* 3d⟩
⟨*Main function, P. 1.1* 1b⟩

The main function interacts with the user, reads the input data, writes the database, and frees any memory still allocated.

1b ⟨*Main function, P. 1.1* 1b⟩≡ (1a)
```
int main(int argc, char **argv) {
  ⟨Interact with user, P. 1.1 2a⟩
  fprintf(stderr, "# Reading data...");
  ⟨Read data, P. 1.1 2h⟩
  fprintf(stderr, "done.\n");
  ⟨Write database, P. 1.1 4d⟩
  ⟨Free memory, P. 1.1 2e⟩
}
```

[1] `https://github.com/haubold/fur`

1.2.1 User Interaction

The most fundamental user interactions are error messages, which require the name of the program sending the message. This is set for future reference.

2a ⟨*Interact with user, P. 1.1* 2a⟩≡ (1b) 2c▷
```
setprogname(argv[0]);
```

The function setprogname is defined in `bsd/stdlib.h`.

2b ⟨*Include headers, P. 1.1* 2b⟩≡ (1a) 2d▷
```
#include <bsd/stdlib.h>
```

The user interacts with the program via a set of options and their arguments.

2c ⟨*Interact with user, P. 1.1* 2a⟩+≡ (1b) ◁2a 2f▷
```
Args *args = getArgs(argc, argv);
```

The `Args` data structure and the functions for handling it are defined in `mfdbI.h`.

2d ⟨*Include headers, P. 1.1* 2b⟩+≡ (1a) ◁2b 2i▷
```
#include "mfdbI.h"
```

The arguments container is freed at the end.

2e ⟨*Free memory, P. 1.1* 2e⟩≡ (1b) 4b▷
```
freeArgs(args);
```

If the user asks for help or an error has occurred, a usage message—also defined in `mfdbI.h`—is printed and the program exits.

2f ⟨*Interact with user, P. 1.1* 2a⟩+≡ (1b) ◁2c 2g▷
```
if (args->h || args->err)
  printUsage();
```

Similarly, the user might like to know the program version, in response to which a (small) splash is made before exiting.

2g ⟨*Interact with user, P. 1.1* 2a⟩+≡ (1b) ◁2f
```
if (args->v)
  printSplash(args);
```

The interaction with the user is now finished and the program on its way.

1.2.2 Construct Database

Database construction begins by reading the targets and the neighbors. Occasionally a user may duplicate a target among the neighbors or the other way round. This is always an error and we catch it.

2h ⟨*Read data, P. 1.1* 2h⟩≡ (1b)
```
SeqArr *ta, *ne;
⟨Read targets, P. 1.1 3a⟩
⟨Read neighbors, P. 1.1 4a⟩
⟨Check targets and neighbors don't intersect, P. 1.1 4c⟩
```

Sequences and sequence arrays are defined in `seq.h`. This header also defines the functions for manipulating these data structures.

2i ⟨*Include headers, P. 1.1* 2b⟩+≡ (1a) ◁2d 3b▷
```
#include "seq.h"
```

The targets are read from a directory passed by the user. Every entry in that directory except for "." and ".." is assumed to be a sequence file.

3a ⟨*Read targets, P. 1.1* 3a⟩≡ (2h)
```
DIR *d;
struct dirent *dir;
ta = newSeqArr();
d = eopendir(args->t);
while ((dir = readdir(d)) != NULL)
  if (strcmp(dir->d_name, ".")  != 0 &&
        strcmp(dir->d_name, "..") != 0)
    readSeq(ta, args->t, dir->d_name);
closedir(d);
```

The previous code chunk refers to a number of preexisting objects, including the directory, `DIR`, and its entries, `dirent`, both declared in `dirent.h`. The function `eopendir` is an error-aware version of `opendir` declared in `error.h`. The function `readdir` is again declared in `dirent.h`, and `strcmp` in `string.h`.

3b ⟨*Include headers, P. 1.1* 2b⟩+≡ (1a) ◁2i 3e▷
```
#include <dirent.h>
#include <sys/types.h>
#include "error.h"
#include <string.h>
```

Now `readSeq` still needs to be declared. It is a function of the sequence array to be added to, the directory path, and the name of the sequence file.

3c ⟨*Function declarations, P. 1.1* 3c⟩≡ (1a)
```
void readSeq(SeqArr *sa, char *dir, char *file);
```

Its main work is to concatenate the directory path and the file name into the file path that serves as the argument to `getJoinedSeq`.

3d ⟨*Function definitions, P. 1.1* 3d⟩≡ (1a)
```
void readSeq(SeqArr *sa, char *dir, char *file) {
  char *path = emalloc(strlen(dir) + strlen(file) + 2);
  path[0] = '\0';
  strcat(path, dir);
  if (path[strlen(path)-1] != '/')
    strcat(path, "/");
  strcat(path, file);
  seqArrAdd(sa, getJoinedSeq(path));
  free(path);
}
```

The only function called in `readSeq` not yet declared is `free`, which is part of in `strlib.h`.

3e ⟨*Include headers, P. 1.1* 2b⟩+≡ (1a) ◁3b
```
#include <stdlib.h>
```

Reading the neighbors is similar to reading the targets.

4a ⟨*Read neighbors, P. 1.1* 4a⟩≡ (2h)
```
ne = newSeqArr();
d = eopendir(args->n);
while ((dir = readdir(d)) != NULL)
  if (strcmp(dir->d_name, ".")  != 0 &&
        strcmp(dir->d_name, "..") != 0)
    readSeq(ne, args->n, dir->d_name);
closedir(d);
```

The targets and neighbors are freed at the end.

4b ⟨*Free memory, P. 1.1* 2e⟩+≡ (1b) ◁2e
```
freeSeqArr(ta);
freeSeqArr(ne);
```

We make sure no sequences were duplicated between targets and neighbors.

4c ⟨*Check targets and neighbors don't intersect, P. 1.1* 4c⟩≡ (2h)
```
for (int i = 0; i < ta->n; i++) {
  char *n1 = strrchr(ta->arr[i]->name, '/') + 1;
  for (int j = 0; j < ne->n; j++) {
    char *n2 = strrchr(ne->arr[j]->name, '/') + 1;
    if (strcmp(n1, n2) == 0)
        error("%s is neighbor and target\n", n1);
  }
}
```

The data just read is now converted into the fur database. The database is a directory, which is constructed first. It contains two kinds of files, the `macle` index, and the BLAST database.

4d ⟨*Write database, P. 1.1* 4d⟩≡ (1b)
```
⟨Create database directory, P. 1.1 4e⟩
⟨Write macle index, P. 1.1 5b⟩
⟨Write BLAST database, P. 1.1 6b⟩
```

Creation of the database directory depends on whether the directory already exists or not.

4e ⟨*Create database directory, P. 1.1* 4e⟩≡ (4d)
```
struct stat sb;
if (stat(args->d, &sb) != -1) {
  ⟨Directory exists, P. 1.1 4f⟩
} else {
  ⟨Directory does not exist, P. 1.1 5a⟩
}
```

If the directory already exists and the user allows it to be overwritten by using option `-o`, the directory is simply left unchanged. Without overwriting, an error is thrown.

4f ⟨*Directory exists, P. 1.1* 4f⟩≡ (4e)
```
if (!args->o)
  error("%s already exists.\n", args->d);
```

If the directory doesn't exist, it is created.

5a ⟨*Directory does not exist, P. 1.1* 5a⟩≡ (4e)
```
char cmd[1024];
sprintf(cmd, "mkdir %s", args->d);
if (system(cmd) < 0)
  error("couldn't run system command %s\n", cmd);
```

The `macle` index consists of a representative target and the neighbors. These are passed to `macle`[2] using the pipe mechanism. Since their names are mainly relevant for internal usage, the representative is called ti, where i is its index in the target array, and the neighbors are called ni.

5b ⟨*Write macle index, P. 1.1* 5b⟩≡ (4d)
```
int r = 0;
⟨Find representative target, P. 1.1 5c⟩
char *tmpl = "macle -s > %s/macle.idx", cmd[1024];
sprintf(cmd, tmpl, args->d);
FILE *pp = epopen(cmd, "w");
fprintf(stderr, "# Making macle index with target representative \"%s\"...",
          ta->arr[r]->name);
fprintf(pp, ">t%d\n%s\n", r, ta->arr[r]->data);
for (int i = 0; i < ne->n; i++)
  fprintf(pp, ">n%d\n%s\n", i, ne->arr[i]->data);
pclose(pp);
fprintf(stderr, "done.\n");
```

If the name of the representative target is given by the user, this is converted to the index in the target sequence array. Otherwise the longest sequence is picked as the representative.

5c ⟨*Find representative target, P. 1.1* 5c⟩≡ (5b)
```
if (args->r) {
  ⟨Convert representative name to index, P. 1.1 5d⟩
} else {
  ⟨Find longest target, P. 1.1 6a⟩
}
```

When searching the names of the targets for the representative, a partial match suffices. Multiple or no matches are an error.

5d ⟨*Convert representative name to index, P. 1.1* 5d⟩≡ (5c)
```
r = -1;
for (int i = 0; i < ta->n; i++)
  if (strstr(ta->arr[i]->name, args->r)) {
    if (r == -1)
        r = i;
    else
        error("%s is ambiguous.\n", args->r);
  }
if (r == -1)
  error("couldn't find %s.\n", args->r);
```

[2] https://github.com/evolbioinf/macle

6a ⟨*Find longest target, P. 1.1* 6a⟩≡ (5c)

```
int max = -1;
for (int i = 0; i < ta->n; i++)
  if (max < ta->arr[i]->l) {
    max = ta->arr[i]->l;
    r = i;
  }
```

The BLAST database consists of the targets and neighbors, named ti and ni, respectively. The program `makeblastdb` computes the database, its option `parse_seqids` allows later retrieval of the representative target by `fur`.

6b ⟨*Write BLAST database, P. 1.1* 6b⟩≡ (4d)

```
fprintf(stderr, "# Making BLAST database...");
tmpl = "makeblastdb -parse_seqids -out %s/blastdb "
  "-dbtype nucl -title db > /dev/null";
sprintf(cmd, tmpl, args->d);
pp = epopen(cmd, "w");
for (int i = 0; i < ta->n; i++)
  fprintf(pp, ">t%d\n%s\n", i, ta->arr[i]->data);
for (int i = 0; i < ne->n; i++)
  fprintf(pp, ">n%d\n%s\n", i, ne->arr[i]->data);
pclose(pp);
fprintf(stderr, "done.\n");
```

Chapter 2

Find Unique Regions, `fur`

2.1 Introduction

The design of diagnostic PCR primers is often hampered by an excess of candidates that also amplify off-target regions. To minimize the chance of cross-amplification, primers should be designed from template sequences that are unique to the target strain. The program `fur` *finds unique regions* by comparing the genomes of a sample of target strains to the genomes of the closest relatives the targets are to be distinguished from. The underlying heuristic is that any region that distinguishes a target from its closest relatives, also distinguishes it from all other sequences out there.

Consider, for example, *Escherichia coli* ST131, a multi-drug resistant strain that causes urinary tract and blood infections in humans [5]. *E. coli* ST131 belongs to the B2 phylogenetic subgroup, which corresponds to serotype O25b:H4. Figure 2.1 shows the phylogeny of 105 *E. coli* B2 strains. The clade marked ST131 comprises 95 strains newly sequenced by [5], plus three STS131 reference genomes, SE15, NA114, and EC958. This clade defines the *targets* marked $\mathcal{T}$ in Figure 2.1. The seven remaining *E. coli* strains are the *neighbors*, $\mathcal{N}$. They also belong to the B2 group, but not to ST131 [5]. The aim is to find regions specific to ST131. In Section 8.1 a tutorial-style analysis of this data set shows how to do this using `fur`.

The program takes as input a database computed using `makeFurDb`[1] from two directories of sequence files, the first contains one or more target genomes, the second one or more neighbor genomes. `fur` uses `macle` [6] to identify candidate regions that are unique to a representative target when compared to all neighbors. These candidate regions are then checked for presence in all targets using `phylonium` [4] and absence from all neighbors using BLAST [1]. The resulting templates are finally printed to screen. They are now ready for submission to a primer design program like `primer3` [7].

2.2 Implementation

The program is based on arrays of sequences and arrays of intervals on those sequences. Arrays of sequences are defined in `seq.h`, while intervals and their arrays are still to be defined. Apart from data structures for intervals and their arrays, the program consists of the usual include section, declarations and definitions of functions, and finally the `main` function.

Program 2.1 (`fur`)**.**

8 ⟨*fur.c* 8⟩≡
```
#include "seq.h"
⟨Include headers, P. 2.1 10e⟩
⟨Data structures, P. 2.1 10a⟩
⟨Function declarations, P. 2.1 10c⟩
⟨Function definitions, P. 2.1 10d⟩
⟨Main function, P. 2.1 11d⟩
```

[1] `https://github.com/haubold/makeFurDb/`

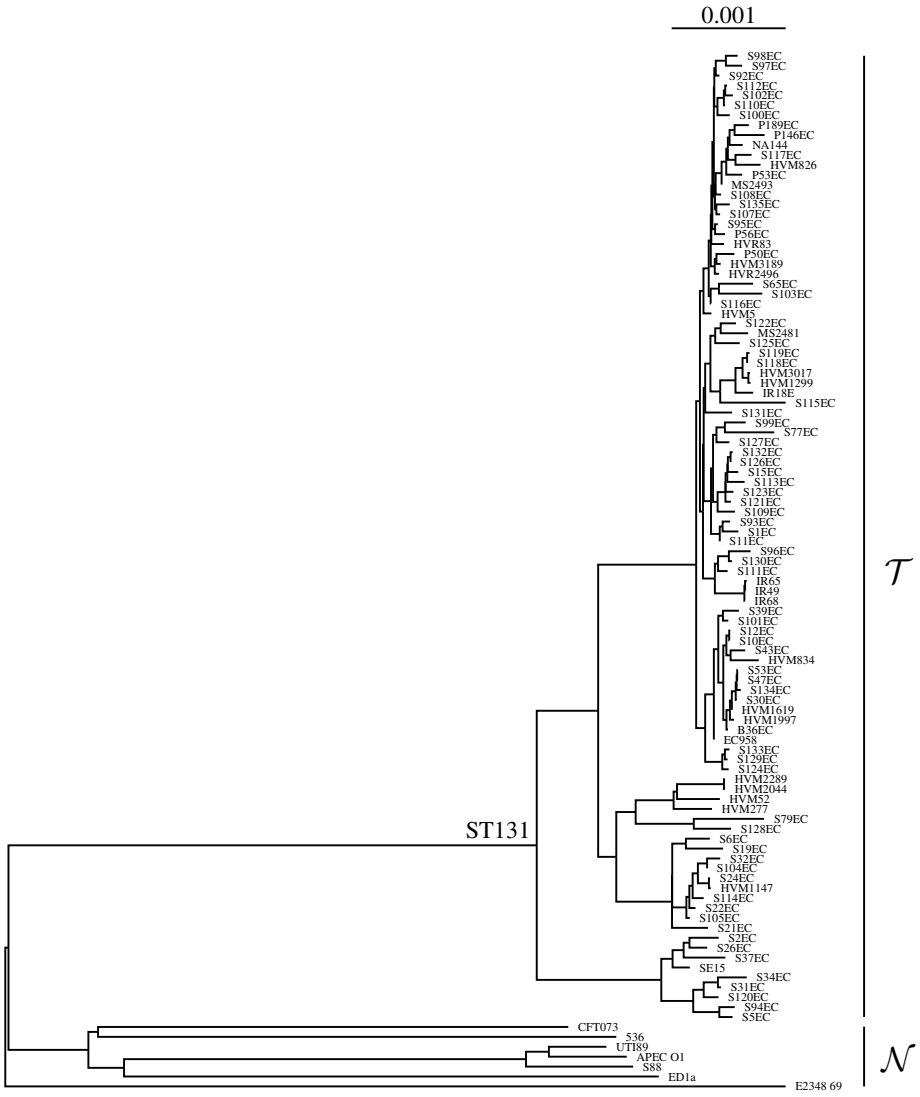

Figure 2.1: Phylogeny of 105 strains of *Eschericia coli* computed from whole genome sequences using `andi` [2]. The scale bar is the number of substitutions per site. The clade marked ST131 contains the pathogenic targets ($\mathcal{T}$), the remaining seven strains are the neighbors ($\mathcal{N}$).

2.2.1 Arrays of Intervals

Intervals and their arrays are the basic building blocks of `fur` still undefined, so they are defined first. Intervals have a start and an end.

10a ⟨*Data structures, P. 2.1* 10a⟩≡ (8) 10b▷
```
typedef struct intv {
  int s, e;
} Intv;
```

An arbitrary number of n intervals is stored in an *interval array*.

10b ⟨*Data structures, P. 2.1* 10a⟩+≡ (8) ◁10a
```
typedef struct intvArr {
  Intv **arr;
  int n;
} IntvArr;
```

Interval arrays require functions for construction, freeing, and addition. Construction is declared with start and end positions supplied as parameters.

10c ⟨*Function declarations, P. 2.1* 10c⟩≡ (8) 10f▷
```
Intv *newIntv(int s, int e);
```

These positions are saved once space has been allocated for them.

10d ⟨*Function definitions, P. 2.1* 10d⟩≡ (8) 10g▷
```
Intv *newIntv(int s, int e) {
  Intv *i = (Intv *)emalloc(sizeof(Intv));
  i->s = s;
  i->e = e;
  return i;
}
```

The function `emalloc` is declared in `error.h`.

10e ⟨*Include headers, P. 2.1* 10e⟩≡ (8) 11f▷
```
#include "error.h"
```

Next, the construction of an interval array is declared.

10f ⟨*Function declarations, P. 2.1* 10c⟩+≡ (8) ◁10c 10h▷
```
IntvArr *newIntvArr();
```

Its definition returns an empty array of intervals.

10g ⟨*Function definitions, P. 2.1* 10d⟩+≡ (8) ◁10d 11a▷
```
IntvArr *newIntvArr() {
  IntvArr *ia = (IntvArr *)emalloc(sizeof(IntvArr));
  ia->arr = NULL;
  ia->n = 0;
  return ia;
}
```

Freeing of an interval array is declared.

10h ⟨*Function declarations, P. 2.1* 10c⟩+≡ (8) ◁10f 11b▷
```
void freeIntvArr(IntvArr *ia);
```

In its definition each interval is freed in turn before the interval array itself is freed.

11a ⟨*Function definitions, P. 2.1* 10d⟩+≡ (8) ◁10g 11c▷
```
void freeIntvArr(IntvArr *ia) {
  for (int i = 0; i < ia->n; i++)
    free(ia->arr[i]);
  free(ia->arr);
  free(ia);
}
```

Declare the addition of an interval to an existing interval array.

11b ⟨*Function declarations, P. 2.1* 10c⟩+≡ (8) ◁10h
```
void intvArrAdd(IntvArr *ia, Intv *i);
```

The definition makes space for the newly arrived interval and then adds it.

11c ⟨*Function definitions, P. 2.1* 10d⟩+≡ (8) ◁11a
```
void intvArrAdd(IntvArr *ia, Intv *i) {
  ia->arr = (Intv **)
    erealloc(ia->arr, (ia->n + 1) * sizeof(Intv *));
  ia->arr[ia->n++] = i;
}
```

Interval arrays are now ready to be used. This is done in the `main` function, which first interacts with the user, then analyzes the targets and neighbors, and finally prints the desired templates. At the end of the program, any memory still allocated is freed.

11d ⟨*Main function, P. 2.1* 11d⟩≡ (8)
```
int main(int argc, char **argv) {
  ⟨Interact with user, P. 2.1 11e⟩
  ⟨Analyze sequences, P. 2.1 12e⟩
  ⟨Print templates, P. 2.1 24d⟩
  ⟨Free memory, P. 2.1 12a⟩
}
```

2.2.2 User Interaction

Whenever the program interacts with the user, it identifies itself, so its name is set.

11e ⟨*Interact with user, P. 2.1* 11e⟩≡ (11d) 11g▷
```
setprogname(argv[0]);
```

The function `setprogname` is declared in the standard part of the BSD library.

11f ⟨*Include headers, P. 2.1* 10e⟩+≡ (8) ◁10e 11h▷
```
#include <bsd/stdlib.h>
```

The user interaction is mediated via a container holding the options and their arguments.

11g ⟨*Interact with user, P. 2.1* 11e⟩+≡ (11d) ◁11e 12b▷
```
Args *args = getArgs(argc, argv);
```

The `Args` data structure and the `getArgs` function are declared in `interface.h`.

11h ⟨*Include headers, P. 2.1* 10e⟩+≡ (8) ◁11f 15a▷
```
#include "interface.h"
```

The argument container is freed at the end.

12a ⟨*Free memory, P. 2.1* 12a⟩≡ (11d) 13d▷
```
freeArgs(args);
```

The options passed via `args` might include a request for help, or indicate an error. In that case, `printUsage`, which is also declared in `interface.h`, emits a usage message before exiting.

12b ⟨*Interact with user, P. 2.1* 11e⟩+≡ (11d) ◁11g 12c▷
```
if (args->h || args->err)
  printUsage();
```

Alternatively, the user might request information about the program, whereupon it makes a modest splash and exits.

12c ⟨*Interact with user, P. 2.1* 11e⟩+≡ (11d) ◁12b 12d▷
```
if (args->v)
  printSplash(args);
```

The last option we check is `-p`, which might indicate a P-value that's not between 0 and 1. In that case we write a friendly message and print the usage.

12d ⟨*Interact with user, P. 2.1* 11e⟩+≡ (11d) ◁12c
```
if (args->p < 0.0 || args->p > 1.0) {
  fprintf(stderr, "Please use a P-value between 0 and 1.\n");
  printUsage();
}
```

2.2.3 Find Unique Templates

Analysis of the targets and neighbors proceeds in three steps:

1. Identify unique regions, $\mathcal{U}_1$, by comparing one representative target to all neighbors.
2. Intersect $\mathcal{U}_1$ with the targets to get unique regions present in all targets, $\mathcal{U}_2$.
3. Subtract the neighbors from $\mathcal{U}_2$ to get regions truly unique to the targets, $\mathcal{U}_3$. In theory, all regions in $\mathcal{U}_2$ should be unique with respect to the neighbors, so $\mathcal{U}_2 = \mathcal{U}_3$. However, the construction of $\mathcal{U}_1$ is less sensitive than the subtraction step. So in practice we have $\mathcal{U}_2 \supset \mathcal{U}_3$.

To summarize, a set of unique regions is created (step 1) and then reduced to ensure its sensitivity (step 2) and specificity (step 3) as markers of the targets.

12e ⟨*Analyze sequences, P. 2.1* 12e⟩≡ (11d)
⟨*Identify unique regions, P. 2.1* 12f⟩
⟨*Intersect with targets, P. 2.1* 18b⟩
⟨*Subtract neighbors, P. 2.1* 21e⟩

Unique regions are identified using the external program `macle`[2] [6]. This operates by traversing a pre-computed index. The index is part of the `fur` database and contains the neighbors augmented by the representative target. This index is used to compute local complexity values for identifying unique intervals.

12f ⟨*Identify unique regions, P. 2.1* 12f⟩≡ (12e)
⟨*Get representative target, P. 2.1* 13a⟩
⟨*Construct unique intervals, P. 2.1* 13e⟩

[2] `https://github.com/evolbioinf/macle`

To obtain the representative target, its name is needed, which allows retrieval of its sequence.

13a ⟨*Get representative target, P. 2.1* 13a⟩≡ (12f)
```
char rn[256];
Seq *rs = NULL;
⟨Get representative name, P. 2.1 13b⟩
⟨Get representative sequence, P. 2.1 13c⟩
```

The representative name is obtained from the `macle` index, where it tops the name list, an ordering is ensured by `makeFurDb`.

13b ⟨*Get representative name, P. 2.1* 13b⟩≡ (13a)
```
char *tmpl = "macle -l %s/macle.idx | "
  "head -n 6 | tail -n 1 | "
  "awk '{print $6 }'";
char cmd[1024];
sprintf(cmd, tmpl, args->d);
FILE *pp = epopen(cmd, "r");
if (fscanf(pp, "%s", rn) == EOF)
  error("couldn't run %s\n", cmd);
pclose(pp);
```

With the name as handle, the corresponding sequence is extracted from the BLAST database.

13c ⟨*Get representative sequence, P. 2.1* 13c⟩≡ (13a)
```
tmpl = "blastdbcmd -entry %s -db %s/blastdb";
sprintf(cmd, tmpl, rn, args->d);
pp = epopen(cmd, "r");
Seq *sp;
while ((sp = getSeq(pp)) != NULL)
  rs = sp;
pclose(pp);
```

The representative target is freed at the end of the program.

13d ⟨*Free memory, P. 2.1* 12a⟩+≡ (11d) ◁12a
```
freeSeq(rs);
```

To construct unique intervals, the complexity threshold indicating uniqueness is computed as preparation for the sliding window analysis of local complexity. Figure 2.2 shows a cartoon of a sliding window analysis. The overlapping windows returned by `macle` are characterized by their mid-points (dots) and are either unique (lightgray) or not (black). Unique windows are summarized into unique intervals (dashed). The user is told about the size of the preliminary template set, and the array of unique intervals is eventually converted to an array of unique sequences, the template candidates.

13e ⟨*Construct unique intervals, P. 2.1* 13e⟩≡ (12f)
```
double mc, gc = 0.;
long len = 0;
IntvArr *ia;
⟨Compute complexity threshold, P. 2.1 14⟩
⟨Sliding window analysis, P. 2.1 15b⟩
⟨Report result of sliding window analysis, P. 2.1 17a⟩
⟨Prepare array of unique sequences, P. 2.1 17c⟩
```

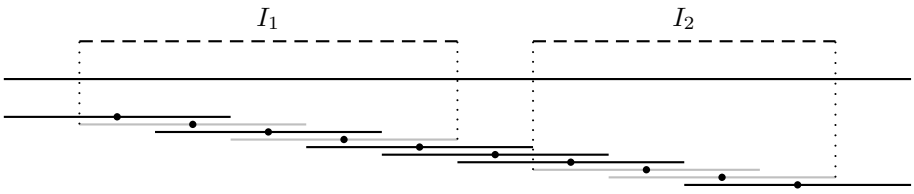

Figure 2.2: Sliding window analysis of a genome sequence. The overlapping windows are centered on their mid-points (dots) and their complexity is either greater than the threshold, which makes them unique (lightgray), or not (black). Unique windows are summarized into the unique intervals I_1 and I_2 (dashed).

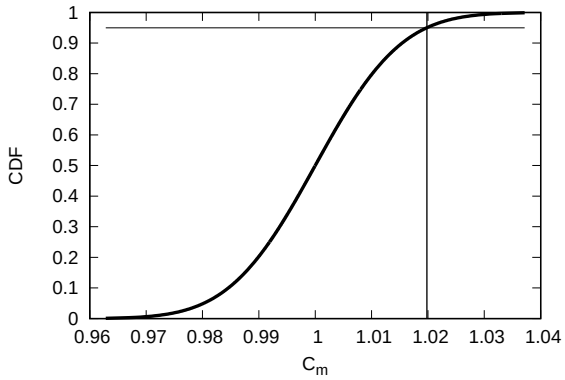

Figure 2.3: Cumulative density function (CDF) of the match complexity (C_{m}) in 500 bp windows over a 35.5 Mb data set with GC-content 0.5 [6]. The parameter choice corresponds to the neighbors depicted in Figure 2.1. The vertical line indicates the complexity threshold for a cumulative value of 0.95.

The complexity threshold is a function of aggregate sequence length, GC-content, window length, and the inverse of the cumulative density function (CDF) of the match length null distribution. Figure 2.3 shows this function and how choosing a particular CDF-value on the y-axis, 0.95 in the example, corresponds to a complexity-threshold on the x-axis, 0.019. Sequence length and GC content are looked up in the `macle` index, window length and probability supplied by the user.

14 ⟨*Compute complexity threshold, P. 2.1* 14⟩≡ (13e)

```
tmpl = "macle -l %s/macle.idx | "
  "tail -n +2 | "
  "awk '{print $2}'";
sprintf(cmd, tmpl, args->d);
pp = epopen(cmd, "r");
if (fscanf(pp, "%ld", &len) == EOF)
  error("couldn't run %s\n", cmd);
if (fscanf(pp, "%lf", &gc) == EOF)
  error("couldn't run %s\n", cmd);
```

```
mc = quantCm(len, gc, args->w, args->p);
pclose(pp);
```

The function `quantCm` is part of the `matchLen`[3] library.

15a ⟨*Include headers, P. 2.1* 10e⟩+≡ (8) ◁11h 20e▷

```
#include "matchLen.h"
```

A sliding window analysis by `macle` returns pairs of values, (m, C_m), where m is the window midpoint and C_m its complexity. We determine the start and end of the current window and decide whether or not it is inside a unique interval. A unique interval is closed by the first non-unique window that starts to the right of the interval, see I_1 in Figure 2.2. However, after iterating across all windows, the start of the last window might still be inside the last interval, I_2 in Figure 2.2 is an example. So we check for this and store the last interval if needed.

15b ⟨*Sliding window analysis, P. 2.1* 15b⟩≡ (13e)

```
⟨Prepare sliding window analysis, P. 2.1 15c⟩
int in = 0;
while (fscanf(pp, "%f %f", &m, &c) != EOF) {
  ⟨Determine window start and end, P. 2.1 16a⟩
  ⟨Window inside unique interval?, P. 2.1 16b⟩
}
pclose(pp);
if (in) {
  ⟨Store last unique interval P. 2.1 16e⟩
}
```

The sliding window analysis requires the opening of a pipe for reading `macle` output. The pipe command consists of three steps. The first calls `macle`, the second cuts the (m, C_m) pairs from the output, and the third removes windows without reliable sequence data, where $C_\mathrm{m} = -1$. In addition, the sliding window analysis requires variables for holding the current midpoint and complexity values, the interval array, and the start and end points of windows and intervals.

15c ⟨*Prepare sliding window analysis, P. 2.1* 15c⟩≡ (15b)

```
tmpl =
  "macle -i %s/macle.idx -n %s -w %d -k %d | "
  "cut -f 2,3 | "
  "awk '$2 > -1'";
sprintf(cmd, tmpl, args->d, rn, args->w, args->k);
pp = epopen(cmd, "r");
float m, c;
ia = newIntvArr();
int is, ie, ws, we;
```

[3] https://github.com/evolbioinf/matchLen

The start and end points of a window are calculated roughly as $m \pm w/2$, where w is the window length. To get the borders exactly right, consider a sequence of length 100, for which `macle` prints a mid-point of 50. To recover the correct start and end positions of 1 and 100 from this, compute

$$\begin{aligned} \text{start} &= m - w/2 + 1 \\ \text{end} &= m + w/2 \end{aligned}$$

Since positions in strings are zero-based, while `macle` output is one-based, the final start and end values are shifted by one position to the left.

16a ⟨*Determine window start and end, P. 2.1* 16a⟩≡ (15b)
```
ws = m - args->w / 2;
we = m + args->w / 2 - 1;
```

We decide whether the current window is inside a unique interval or not.

16b ⟨*Window inside unique interval?, P. 2.1* 16b⟩≡ (15b)
```
if (in) {
  ⟨Inside unique interval, P. 2.1 16c⟩
} else {
  ⟨Outside unique interval, P. 2.1 16d⟩
}
```

If a unique *window* overlaps an existing unique *interval*, the interval is extended to the right (Figure 2.2). If the unique window lies beyond the existing interval, the interval is "closed" at the endpoint found in the last extension and added to the interval array. Note that the interval is *not* closed as soon as it cannot be extended. Such a rule would break up I_1 in Figure 2.2 into two overlapping and hence redundant intervals.

16c ⟨*Inside unique interval, P. 2.1* 16c⟩≡ (16b)
```
if (ws <= ie && c >= mc)
  ie = we;
else if (ws > ie) {
  in = 0;
  intvArrAdd(ia, newIntv(is, ie));
}
```

If a unique window is found outside a unique interval, a new unique interval is created.

16d ⟨*Outside unique interval, P. 2.1* 16d⟩≡ (16b)
```
if (c >= mc) {
  in = 1;
  is = m - args->w / 2;
  ie = m + args->w / 2 - 1;
}
```

We store the last unique interval.

16e ⟨*Store last unique interval P. 2.1* 16e⟩≡ (15b)
```
intvArrAdd(ia, newIntv(is, ie));
```

The result of the sliding window analysis is reported.

17a ⟨*Report result of sliding window analysis, P. 2.1* 17a⟩≡ (13e)

```
int nn = 0, nm = 0;
⟨Parse result of sliding window analysis, P. 2.1 17b⟩
char *h1 = "# Step                    Sequences  Nucleotides  "
  "Mutations (N)";
char *h2 = "# ------------------------------------------------"
  "-------------";
fprintf(stderr, "%s\n%s\n", h1, h2);
tmpl = "# Sliding window             %6d     %8d         %6d\n";
fprintf(stderr, tmpl, ia->n, nn, nm);
```

The result of the sliding window analysis is parsed by looking at residue to count Ns and everything else.

17b ⟨*Parse result of sliding window analysis, P. 2.1* 17b⟩≡ (17a)

```
for (int i = 0; i < ia->n; i++)
  for (int j = ia->arr[i]->s; j <= ia->arr[i]->e; j++)
    if (rs->data[j] == 'N')
        nm++;
    else
        nn++;
```

The array of unique intervals is now converted to the corresponding array of sequences. The templates are numbered and the fragment coordinates are included in the headers. Once the templates have been written, the interval array is freed. For debugging purposes, the program can also print the unique sequences.

17c ⟨*Prepare array of unique sequences, P. 2.1* 17c⟩≡ (13e)

```
SeqArr *sa = newSeqArr();
char name[1024];
for (int i = 0; i < ia->n; i++) {
  Intv *iv = ia->arr[i];
  sprintf(name, "template_%d %d-%d\n", i + 1, iv->s + 1,
            iv->e + 1);
  Seq *s = newSeq(name);
  ⟨Copy sequence data, P. 2.1 18a⟩
  seqArrAdd(sa, s);
}
freeIntvArr(ia);
⟨Print unique sequences? P. 2.1 17d⟩
```

After printing the unique sequences, the program exits.

17d ⟨*Print unique sequences? P. 2.1* 17d⟩≡ (17c)

```
if (args->u) {
  for (int i = 0; i < sa->n; i++)
    printSeq(stdout, sa->arr[i], -1);
  exit(0);
}
```

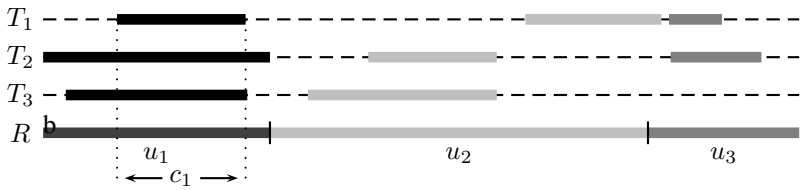

Figure 2.4: Intersect unique regions and targets. The three unique regions, $\{u_1, u_2, u_3\}$ are concatenated to form the reference sequence, R. The target sequences, $\{T_1, T_2, T_3\}$ are aligned to R and the gapped positions removed to leave the candidate templates. In this cartoon there is only one such candidate, c_1.

To copy the sequence data, memory is allocated, each nucleotide copied, and the sequence string terminated by the null character.

18a ⟨*Copy sequence data, P. 2.1* 18a⟩≡ (17c)

```
s->data = emalloc(iv->e - iv->s + 2);
for (int j = iv->s; j <= iv->e; j++)
  s->data[s->l++] = rs->data[j];
s->data[s->l] = '\0';
```

The intervals in hand are candidates for template sequences. But before they are printed, they are reduced to those regions present in all targets and absent from all neighbors.

2.3 Intersect with Targets

At this point the template candidates come from a single target sequence, the representative. To ensure they also occur in all other targets, the templates are intersected with the remaining targets using a second external program, `phylonium` [4]. `Phylonium` takes as input a set of sequences, one of which is designated the reference. In the context of `fur`, the reference is made up of the template candidates just identified. All contigs of the reference are concatenated. For example, in Figure 2.4 the reference, R, consists of three unique candidate regions, u_1, u_2, and u_3. The remaining targets—T_1, T_2, and T_3 in the example—get aligned to R. Region u_1 now has overlapping matches from all three targets, region u_2 has matches from the three targets, but only those from T_2 and T_3 overlap, and region u_3 has no match from T_3. The intersection between R and T_1–T_3 is formed by removing all positions with gaps, resulting in one template candidate, c_1.

18b ⟨*Intersect with targets, P. 2.1* 18b⟩≡ (12e)

```
⟨Write templates to file, P. 2.1 19a⟩
⟨Write targets to files, P. 2.1 19b⟩
⟨Run phylonium, P. 2.1 19d⟩
⟨Delete template and target files, P. 2.1 21b⟩
⟨Report result of intersection, P. 2.1 21c⟩
⟨Print ubiquitous templates and exit? P. 2.1 21d⟩
```

The templates are written to the file `r.fasta` inside the database directory by iterating across the template array and printing headers and sequences.

19a ⟨*Write templates to file, P. 2.1* 19a⟩≡ (18b)
```
tmpl = "%s/r.fasta";
sprintf(name, tmpl, args->d);
FILE *fp = efopen(name, "w");
for (int i = 0; i < sa->n; i++)
  printSeq(fp, sa->arr[i], -1);
fclose(fp);
```

The remaining targets are read from the BLAST database and written to individual files inside the database array. The program `blastdbcmd`, which is part of the BLAST package, allows access to BLAST databases. This adds a blank to the end of each header, which we remove again to ensure accurate identification later.

19b ⟨*Write targets to files, P. 2.1* 19b⟩≡ (18b)
```
in = 0;
tmpl = "blastdbcmd -entry all -db %s/blastdb | sed 's/ $//'";
sprintf(cmd, tmpl, args->d);
pp = epopen(cmd, "r");
⟨Iterate across BLAST database, P. 2.1 19c⟩
pclose(pp);
```

When iterating across the BLAST database, we avoid neighbors and the target representative.

19c ⟨*Iterate across BLAST database, P. 2.1* 19c⟩≡ (19b)
```
while ((sp = getSeq(pp)) != NULL) {
  if (sp->name[0] == 't' && strcmp(sp->name, rn) != 0)  {
    sprintf(name, "%s/t%d.fasta", args->d, ++in);
    fp = efopen(name, "w");
    printSeq(fp , sp, -1);
    fclose(fp);
  }
  freeSeq(sp);
}
```

If there was at least one target in addition to the target representative, we run `phylonium`. To do so, we construct and execute the appropriate command, then save the results. These consist of a set of FASTA entries with headers containing information about mutations in the fragments. The mutations are marked in the sequences to inform primer construction later.

19d ⟨*Run* `phylonium`*, P. 2.1* 19d⟩≡ (18b)
```
if (in > 0) {
  ⟨Construct and execute phylonium command, P. 2.1 20a⟩
  ⟨Save phylonium results, P. 2.1 20b⟩
  ⟨Mark mutations, P. 2.1 20c⟩
}
```

Phylonium is applied to the target files just constructed. It writes the intersection to the file p.fasta. All output to stdout or stderr is discarded.

20a ⟨*Construct and execute* phylonium *command, P. 2.1* 20a⟩≡ (19d)
```
tmpl = "phylonium -p %s/p.fasta -r %s/r.fasta %s/*.fasta "
  "> /dev/null 2> /dev/null";
sprintf(cmd, tmpl, args->d, args->d, args->d);
if (system(cmd) < 0)
  error("couldn't run system call %s\n", cmd);
```

The intersecting sequence fragments in p.fasta are saved if long enough.

20b ⟨*Save* phylonium *results, P. 2.1* 20b⟩≡ (19d)
```
sprintf(name, "%s/p.fasta", args->d);
fp = efopen(name, "r");
freeSeqArr(sa);
sa = newSeqArr();
while ((sp = getSeq(fp)) != NULL)
  if (sp->l >= args->n)
    seqArrAdd(sa, sp);
  else
    freeSeq(sp);
fclose(fp);
```

Phylonium returns entries of the form

$$\texttt{>part}_i\ (s..e)\ n\ p_1\ p_2\ ...\ p_n$$

where n is the number of mutations found at positions $p_1, p_2, ..., p_n$. These positions are set to the unknown nucleotide, N, so they can later be avoided when designing primers.

20c ⟨*Mark mutations, P. 2.1* 20c⟩≡ (19d)
```
for (int i = 0; i < sa->n; i++) {
  ⟨Determine the number of mutations, P. 2.1 20d⟩
  ⟨Iterate across mutations, P. 2.1 21a⟩
}
```

The number of mutations is found by looking for the closing bracket of the fragment's interval. Using the -x option, the user can request only exact matches.

20d ⟨*Determine the number of mutations, P. 2.1* 20d⟩≡ (20c)
```
char *h = strstr(sa->arr[i]->name, ")");
h += 2;
int j = atoi(strtok(h, " "));
if (j == 0) {
  continue;
} else if (args->x) {
  freeSeq(sa->arr[i]);
  sa->arr[i] = NULL;
  continue;
}
```

The functions strstr for looking up the first occurrence of a character and strtok to iterate across string tokens are both declared in string.h.

20e ⟨*Include headers, P. 2.1* 10e⟩+≡ (8) ◁15a 23b▷
```
#include <string.h>
```

All integers following the number of mutations are one-based positions.

21a ⟨*Iterate across mutations, P. 2.1* 21a⟩≡ (20c)
```
char *t = strtok(NULL, " ");
while (t != NULL) {
  int p = atoi(t) - 1;
  sa->arr[i]->data[p] = 'N';
  t = strtok(NULL, " ");
}
```

The files used by `phylonium` are deleted.

21b ⟨*Delete template and target files, P. 2.1* 21b⟩≡ (18b)
```
tmpl = "rm %s/*.fasta";
sprintf(cmd, tmpl, args->d);
if (system(cmd) < 0)
  error("couldn't run system call %s\n", cmd);
```

The user is told about the number of sequences, nucleotides, and `Ns` in the targets.

21c ⟨*Report result of intersection, P. 2.1* 21c⟩≡ (18b)
```
int ns = 0;
nn = nm = 0;
for (int i = 0; i < sa->n; i++) {
  if (!sa->arr[i]) continue;
  ns++;
  for (int j = 0; j < sa->arr[i]->l; j++)
    if (sa->arr[i]->data[j] == 'N') nm++;
    else nn++;
}
tmpl = "# Presence in targets        %6d     %8ld         %6d\n";
fprintf(stderr, tmpl, ns, nn, nm);
```

The ubiquitous templates can be inspected.

21d ⟨*Print ubiquitous templates and exit? P. 2.1* 21d⟩≡ (18b)
```
if (args->U) {
  for (int i = 0; i < sa->n; i++)
    if (sa->arr[i])
        printSeq(stdout, sa->arr[i], -1);
  exit(0);
}
```

2.4 Subtract Neighbors

Any neighbor sequences still present among the remaining templates are removed (subtracted) using a third external program, `blastn` [1]. The candidate templates are searched in the BLAST database and the hits written to file. This file is read back into `fur`, and the regions with homologs among the neighbors are again set to `N`, unless the "exact" option, `-x`, is set.

21e ⟨*Subtract neighbors, P. 2.1* 21e⟩≡ (12e)
```
⟨Search neighbors, P. 2.1 22a⟩
⟨Mark regions found among neighbors, P. 2.1 22d⟩
⟨Report result of subtraction, P. 2.1 24a⟩
```

The neighbors are searched by constructing the `blastn` pipe and then running the neighbor sequences through it.

22a ⟨*Search neighbors, P. 2.1* 22a⟩≡ (21e)
⟨*Construct neighbor pipe, P. 2.1* 22b⟩
⟨*Write templates to neighbor pipe, P. 2.1* 22c⟩

In the neighbor pipe we write the subject accession and query coordinates to the output file, `o.txt`, inside the database directory. By default BLAST is run in the slower and more sensitive "blastn" mode, but the user can switch to the faster and less sensitive "megablast" mode.

22b ⟨*Construct neighbor pipe, P. 2.1* 22b⟩≡ (22a)
```
tmpl = "blastn -task %s -db %s/blastdb -num_threads %d "
  "-evalue %e -outfmt \"6 sacc qacc qstart qend\" "
  "| grep '^n' > "
  "%s/o.txt";
if(args->m)
  sprintf(cmd, tmpl, "megablast", args->d, args->t, args->e, args->d);
else
  sprintf(cmd, tmpl, "blastn", args->d, args->t, args->e, args->d);
pp = epopen(cmd, "w");
```

The template candidates are written to this pipe with their index numbers as identifiers.

22c ⟨*Write templates to neighbor pipe, P. 2.1* 22c⟩≡ (22a)
```
for (int i = 0; i < sa->n; i++)
  if (sa->arr[i])
    fprintf(pp, ">%d\n%s\n", i, sa->arr[i]->data);
pclose(pp);
```

BLAST may return overlapping regions. These are summarized before marking them.

22d ⟨*Mark regions found among neighbors, P. 2.1* 22d⟩≡ (21e)
```
tmpl = "%s/o.txt";
sprintf(name, tmpl, args->d);
fp = efopen(name, "r");
```
⟨*Summarize neighbor BLAST output, P. 2.1* 22e⟩
```
fclose(fp);
```
⟨*Set homologous neighbor regions to N, P. 2.1* 23d⟩
⟨*Free BLAST resources, P. 2.1* 23e⟩

To summarize the output of the BLAST search among neighbors, space for the results is created before the results themselves are scanned.

22e ⟨*Summarize neighbor BLAST output, P. 2.1* 22e⟩≡ (22d)
⟨*Allocate space for output of neighbor BLAST, P. 2.1* 23a⟩
⟨*Scan output of neighbor BLAST, P. 2.1* 23c⟩

We allocate space for the start and end positions of each homologous regions and initialize these to values that allow us to later summarize overlapping intervals.

23a ⟨*Allocate space for output of neighbor BLAST, P. 2.1* 23a⟩≡ (22e)
```
int *start = emalloc(sa->n * sizeof(int));
int *end   = emalloc(sa->n * sizeof(int));
for (int i = 0; i < sa->n; i++) {
  start[i] = INT_MAX;
  end[i]   = -1;
}
```

`INT_MAX` is the maximum value an integer may take and is defined in `limits.h`.

23b ⟨*Include headers, P. 2.1* 10e⟩+≡ (8) ◁20e
```
#include <limits.h>
```

During the scan of the BLAST output, intervals are extended to the left and the right.

23c ⟨*Scan output of neighbor BLAST, P. 2.1* 23c⟩≡ (22e)
```
int ii, qs, qe;
char s[32];
while (fscanf(fp, "%s %d %d %d", s, &ii, &qs, &qe) != EOF) {
  if (qs < start[ii])
    start[ii] = qs ;
  if (qe > end[ii])
    end[ii] = qe;
}
```

The regions with homology among the neighbors are set to `N`, bearing in mind that BLAST-coordinates are 1-based, character arrays 0-based. As the arrays with the start and end coordinates are not needed any more afterwards, they are freed.

23d ⟨*Set homologous neighbor regions to N, P. 2.1* 23d⟩≡ (22d)
```
for (int i = 0; i < sa->n; i++) {
  int l = end[i] - start[i] + 1;
  if (l > 0 && args->x) {
    freeSeq(sa->arr[i]);
    sa->arr[i] = NULL;
    continue;
  }
  for (int j = start[i] - 1; j < end[i]; j++)
    sa->arr[i]->data[j] = 'N';
}
```

The resources used up by the BLAST run, the output file and the arrays of start and end positions, are freed again.

23e ⟨*Free BLAST resources, P. 2.1* 23e⟩≡ (22d)
```
tmpl = "rm %s/o.txt";
sprintf(cmd, tmpl, args->d);
if (system(cmd) < 0) {
  fprintf(stderr, "couldn't run system call %s\n", cmd);
  exit(0);
}
free(start);
free(end);
```

In order to report the results of the subtraction step, we iterate over all residues in all sequences and count the number of mutations. At this point we can also classify the sequences into those fit for subsequent analysis and those that aren't.

24a ⟨*Report result of subtraction, P. 2.1* 24a⟩≡ (21e)

```
nn = nm = ns = 0;
for (int i = 0; i < sa->n; i++) {
  if (!sa->arr[i]) continue;
  ⟨Count mutations, P. 2.1 24b⟩
  ⟨Classify sequences, P. 2.1 24c⟩
}
tmpl = "# Absence from neighbors     %6d     %8ld         %6d\n";
fprintf(stderr, tmpl, ns, nn, nm);
```

The mutations are counted by again looking at every residue in the current result set. An `N` is counted as a mutation, everything else as a nucleotide.

24b ⟨*Count mutations, P. 2.1* 24b⟩≡ (24a)

```
int cn = 0, cm = 0;
for (int j = 0; j < sa->arr[i]->l; j++)
  if (sa->arr[i]->data[j] == 'N')
    cm++;
  else
    cn++;
```

Sequences are classified as fit for printing if they contain enough nucleotides.

24c ⟨*Classify sequences, P. 2.1* 24c⟩≡ (24a)

```
if (cn >= args->n) {
  ns++;
  nm += cm;
  nn += cn;
} else {
  freeSeq(sa->arr[i]);
  sa->arr[i] = NULL;
}
```

The last step in `fur` is to print the template sequences just identified.

24d ⟨*Print templates, P. 2.1* 24d⟩≡ (11d)

```
for (int i = 0; i < sa->n; i++)
  if (sa->arr[i])
    printSeq(stdout, sa->arr[i], -1);
freeSeqArr(sa);
```

To check the output of `fur` in the Tutorial (Section 8.1), we write the AWK script `count`. It counts the headers and sums the sequence lengths before reporting the number of templates and nucleotides.

Program 2.2 (`count`)**.**

25 ⟨*count* 25⟩≡

```
#!/usr/bin/awk -f
{
  if (/^>/)
    c++
  else {
    t += length($1)
    n += gsub("N", "")
  }
}
END {
  printf "# %s tmpl, %d nuc, %d N, %d total\n", c, t - n, n, t
}
```

`Fur` is now ready to be used.

Chapter 3

Clean Sequences, `cleanSeq`

3.1 Introduction

The marker candidates returned by `fur` may contain long internal runs of `N`, or prefixes or suffixes consisting of `N`s. The program `cleanSeq` cuts these runs and ensures that the resulting fragments are not too short.

3.2 Implementation

The outline of `cleanSeq` has hooks for imports, types, variables, functions, and the logic of the main function.

Program 3.1 (`cleanSeq`)**.**

28a ⟨*cleanSeq.go* 28a⟩≡

```
package main

import (
        ⟨Imports, Pr. 3.1 28d⟩
)
⟨Variables, Pr. 3.1 29c⟩
⟨Types, Pr. 3.1 30c⟩
⟨Functions, Pr. 3.1 29f⟩
func main() {
        ⟨Main function, Pr. 3.1 28b⟩
}
```

In the main function we declare the options, set the usage, parse the options, and iterate over the input.

28b ⟨*Main function, Pr. 3.1* 28b⟩≡ (28a)

```
⟨Declare options, Pr. 3.1 28c⟩
⟨Set usage, Pr. 3.1 28e⟩
⟨Parse options, Pr. 3.1 29b⟩
⟨Iterate over input, Pr. 3.1 29e⟩
```

`CleanSeq` requires a maximum length of internal runs and a minimum fragment length. The user can also use `-v` to request the version.

28c ⟨*Declare options, Pr. 3.1* 28c⟩≡ (28b)

```
var optL = flag.Int("l", 150, "maximum length of internal run of Ns")
var optM = flag.Int("m", 100, "minimum fragment length")
var optV = flag.Bool("v", false, "print version & " +
        "program information")
```

We import `flag`.

28d ⟨*Imports, Pr. 3.1* 28d⟩≡ (28a) 29a ▷

```
"flag"
```

The usage consists of the usage message proper, an explanation of the program's purpose, and an example command.

28e ⟨*Set usage, Pr. 3.1* 28e⟩≡ (28b)

```
u := "cleanSeq [-h] [option]... [file]..."
p := "Cut runs of N from the sequences returned by fur."
e := "cleanSeq foo.fasta"
clio.Usage(u, p, e);
```

We import `clio`.

29a ⟨*Imports, Pr. 3.1* 28d⟩+≡ (28a) ◁28d 29d▷
```
"github.com/evolbioinf/clio"
```

We parse the options and respond to `-v`.

29b ⟨*Parse options, Pr. 3.1* 29b⟩≡ (28b)
```
flag.Parse()
if *optV {
        a := "Bernhard Haubold"
        e := "haubold@evolbio.mpg.de"
        l := "GNU General Public License, " +
                "https://www.gnu.org/licenses/gpl.html"
        clio.PrintInfo("cleanSeq", version, date, a, e, l)
        os.Exit(0)
}
```

The variables `version` and `date` get injected at compile time.

29c ⟨*Variables, Pr. 3.1* 29c⟩≡ (28a)
```
var version, date string
```

We import `os`.

29d ⟨*Imports, Pr. 3.1* 28d⟩+≡ (28a) ◁29a 29g▷
```
"os"
```

We iterate over the input by calling `clio.PaseFiles` on the input files. This takes as argument the function `parse`, which in turn takes as arguments the parameters of the cleaning procedure.

29e ⟨*Iterate over input, Pr. 3.1* 29e⟩≡ (28b)
```
files := flag.Args()
clio.ParseFiles(files, parse, *optL, *optM)
```

In `parse` we retrieve the arguments just passed and iterate over the sequences in the input file to clean them.

29f ⟨*Functions, Pr. 3.1* 29f⟩≡ (28a)
```
func parse(r io.Reader, args ...interface{}) {
        ⟨Retrieve arguments, Pr. 3.1 29h⟩
        scanner := fasta.NewScanner(r)
        for scanner.ScanSequence() {
                sequence := scanner.Sequence()
                ⟨Clean sequence, Pr. 3.1 30a⟩
        }
}
```

We import `io` and `fasta`.

29g ⟨*Imports, Pr. 3.1* 28d⟩+≡ (28a) ◁29d 32a▷
```
"io"
"github.com/evolbioinf/fasta"
```

We retrieve the two arguments just passed through type assertion.

29h ⟨*Retrieve arguments, Pr. 3.1* 29h⟩≡ (29f)
```
maxRunLen := args[0].(int)
minFragLen := args[1].(int)
```

We clean a sequence by first identifying its relevant runs of N. Then we remove the runs and any short fragments thus produced. The remaining fragments are printed.

30a ⟨*Clean sequence, Pr. 3.1* 30a⟩≡ (29f)
```
⟨Identify runs of N, Pr. 3.1 30b⟩
⟨Remove runs of N, Pr. 3.1 30e⟩
⟨Remove short fragments, Pr. 3.1 31e⟩
⟨Print fragments, Pr. 3.1 32b⟩
```

We use a slice of intervals to store the start and end positions of runs runs after we checked them.

30b ⟨*Identify runs of* N*, Pr. 3.1* 30b⟩≡ (30a)
```
var runs []interval
var run interval
data := sequence.Data()
n := len(data)
for i := 0; i < n; i++ {
        j := 0
        for i+j < n && data[i+j] == 'N' { j++ }
        ⟨Check runs, Pr. 3.1 30d⟩
        i += j
}
```

We declare an `interval` with a `start` and an `end`.

30c ⟨*Types, Pr. 3.1* 30c⟩≡ (28a)
```
type interval struct {
        start, end int
}
```

A valid run is either a prefix, a suffix, or it has the minimum number of Ns set by the user.

30d ⟨*Check runs, Pr. 3.1* 30d⟩≡ (30b)
```
if (i == 0 && j > 0) || i + j == n || j >= maxRunLen {
        run.start = i
        run.end = i + j - 1
        runs = append(runs, run)
}
```

By removing runs of N, we split the sequence in fragments, which we store in a slice. We consider two cases, no runs and at least one run.

30e ⟨*Remove runs of* N*, Pr. 3.1* 30e⟩≡ (30a)
```
var fragments []*fasta.Sequence
if len(runs) == 0 {
        ⟨No run, Pr. 3.1 30f⟩
} else {
        ⟨At least one run, Pr. 3.1 31a⟩
}
```

If there weren't any runs, there is only one fragment, the original sequence, which we store.

30f ⟨*No run, Pr. 3.1* 30f⟩≡ (30e)
```
fragments = append(fragments, sequence)
```

If there is at least one run, we do three things. We ask whether we are dealing with a prefix, we iterate over the internal runs, and we extract the fragment to the left of the last run.

31a ⟨*At least one run, Pr. 3.1* 31a⟩≡ (30e)
⟨*Is the run a prefix? Pr. 3.1* 31b⟩
⟨*Deal with internal runs, Pr. 3.1* 31c⟩
⟨*Extract last fragment, Pr. 3.1* 31d⟩

Whether or not a run is a prefix only matters if there is more than one run. In that case the end of the previous run is adjusted and the prefix squeezed from the slice.

31b ⟨*Is the run a prefix? Pr. 3.1* 31b⟩≡ (31a)
```
prevEnd := -1
if runs[0].start == 0 && len(runs) > 1 {
        prevEnd = runs[0].end
        runs = runs[1:]
}
```

We iterate over the runs and store the fragments we get.

31c ⟨*Deal with internal runs, Pr. 3.1* 31c⟩≡ (31a)
```
header := sequence.Header()
for _, run = range runs {
        seq := fasta.NewSequence(header,
                data[prevEnd+1:run.start])
        fragments = append(fragments, seq)
        prevEnd = run.end
}
```

If the last run wasn't a suffix, we extract the fragments its removal generated.

31d ⟨*Extract last fragment, Pr. 3.1* 31d⟩≡ (31a)
```
if run.end < n {
        seq := fasta.NewSequence(header, data[run.end+1:])
        fragments = append(fragments, seq)
}
```

We squeeze short fragments out of the slice.

31e ⟨*Remove short fragments, Pr. 3.1* 31e⟩≡ (30a) 31f▷
```
i := 0
for _, f := range fragments {
        if len(f.Data()) >= minFragLen {
                fragments[i] = f
                i++
        }
}
fragments = fragments[0:i]
```

Any fragments produced at this stage have the same header. If there are more than one, this is confusing, so we number them.

31f ⟨*Remove short fragments, Pr. 3.1* 31e⟩+≡ (30a) ◁31e
```
if len(fragments) > 1 {
        for i, f := range fragments {
                f.AppendToHeader(" - F" + strconv.Itoa(i+1))
        }
}
```

We import `strconv`.

32a ⟨*Imports, Pr. 3.1* 28d⟩+≡ (28a) ◁29g 32c▷

```
"strconv"
```

We print the fragments using the print mechanism for sequences.

32b ⟨*Print fragments, Pr. 3.1* 32b⟩≡ (30a)

```
for _, f := range fragments {
        fmt.Println(f)
}
```

We import `fmt`.

32c ⟨*Imports, Pr. 3.1* 28d⟩+≡ (28a) ◁32a

```
"fmt"
```

Chapter 4

Compute the Sensitivity and Specificity of `fur`, `senSpec`

4.1 Introduction

Given a set of templates proposed by `fur`, we'd like to know how many of the nucleotides were found among the targets compared to how many should have been found. This is called the *sensitivity* [3, p. 121f]:

$$S_{\mathrm{n}} = \frac{t_{\mathrm{p}}}{t_{\mathrm{p}} + f_{\mathrm{n}}}, \tag{4.1}$$

where t_{p} is the number of true positives—the number of nucleotides hit—and f_{n} the number of targets that should have been but weren't.

We'd also like to compare the number of nucleotides in target hits to the number of nucleotides in neighbor hits, which is called the *specificity*:

$$S_{\mathrm{p}} = \frac{t_{\mathrm{p}}}{t_{\mathrm{p}} + f_{\mathrm{p}}}, \tag{4.2}$$

where f_{p} is the number of nucleotides in neighbor hits, the false positives.

S_{n} and S_{p} are bounded by 0 and 1, the greater they both are, the better. However, it is easy to maximize just one of them at the expense of the other, so their correlation is often used to measure classification accuracy [3, p. 122]:

$$C = \frac{t_{\mathrm{p}} t_{\mathrm{n}} - f_{\mathrm{p}} f_{\mathrm{n}}}{\sqrt{(t_{\mathrm{p}} + f_{\mathrm{p}})(t_{\mathrm{n}} + f_{\mathrm{n}})(t_{\mathrm{n}} + f_{\mathrm{p}})(t_{\mathrm{p}} + f_{\mathrm{n}})}}. \tag{4.3}$$

C ranges between -1 and 1, with 1 indicating perfect classification—all template nucleotides are found in all targets and none of them are found among the neighbors—0 indicates no discrimination, and -1 perfect anti-classification, where all neighbors are hit and no targets. When comparing the results of a `fur` run to the underlying database, the sensitivity, the specificity, and their correlation should be 1.

4.2 Implementation

The program `senSpec` takes as input a query consisting of the set of template sequences computed by `fur` and a `fur` database. It returns the specificity, sensitivity, and correlation of that `fur` run.

Initially the user is prompted for input, before the query lengths are saved. Then the number of targets and neighbors is computed. This is followed by counting the true positive nucleotides, t_{p}, and their complement, the false negative nucleotides, f_{n}. Then we count the false positive nucleotides, f_{p}, and their complement, the true negative nucleotides, t_{n}. At the end, the sensitivity, S_{n}, the specificity, S_{p}, and their correlation, C, are printed.

Program 4.1 (`senSpec`)**.**

34 ⟨*senSpec* 34⟩≡

```
#!/usr/bin/awk -f
BEGIN {
  ⟨Interact with user, P. 4.1 35a⟩
  ⟨Save query lengths, P. 4.1 35b⟩
  ⟨Count targets, P. 4.1 35c⟩
  ⟨Count neighbors, P. 4.1 35d⟩
```

⟨*Compute t_p and f_n, P. 4.1* 35e⟩
⟨*Compute f_p and t_n, P. 4.1* 37b⟩
⟨*Print S_n, S_p, and C, P. 4.1* 37e⟩
}

The user is prompted for a set of query sequences and the name of a `fur` database.

35a ⟨*Interact with user, P. 4.1* 35a⟩≡ (34)
```
defEvalue = 1e-5
if (!query || !db) {
  print  "Usage: senSpec -v query=<query.fasta> -v db=<furDb>"
  printf "\t[-v evalue=<evalue>; default: %.1e]\n", defEvalue
  exit
}
if (!evalue)
  evalue = defEvalue
```

The query lengths are saved by traversing the query file.

35b ⟨*Save query lengths, P. 4.1* 35b⟩≡ (34)
```
cmd = "cat " query
while (cmd | getline) {
  if (/^>/) {
    h = $1;
    sub(">", "", h)
    ql[h] = 0
  } else
    ql[h] += length($0)
}
```

The targets are counted by filtering for them in the BLAST part of the `fur` database.

35c ⟨*Count targets, P. 4.1* 35c⟩≡ (34)
```
tmpl = "blastdbcmd -entry all -db %s/blastdb | grep -c '^>%s'"
cmd = sprintf(tmpl, db, "t")
cmd | getline
nt = $1
close(cmd)
```

The neighbors are counted in a similar way.

35d ⟨*Count neighbors, P. 4.1* 35d⟩≡ (34)
```
cmd = sprintf(tmpl, db, "n")
cmd | getline
nn = $1
close(cmd)
```

To count the true positive and false negative nucleotides, we run a BLAST and traverse its results before computing the individual statistics.

35e ⟨*Compute t_p and f_n, P. 4.1* 35e⟩≡ (34)
⟨*Construct BLAST command, P. 4.1* 36a⟩
⟨*Traverse BLAST results, P. 4.1* 36b⟩
⟨*Compute t_p, P. 4.1* 36d⟩
⟨*Compute f_n, P. 4.1* 37a⟩

The BLAST search returns four values:

1. `sacc`: Subject accession
2. `qacc`: Query accession
3. `qstart`: Query start in alignment
4. `qlen`: Query end in alignment

They are first filtered for *targets* and then sorted by subject, query, and query start, in that order.

36a ⟨*Construct BLAST command, P. 4.1* 36a⟩≡ (35e)
```
tmpl = "blastn -outfmt \"6 sacc qacc qstart qend\" "
tmpl = tmpl "-task blastn -query %s -db %s/blastdb -evalue %s "
tmpl = tmpl "| awk '$1 ~ /^%s/' "
tmpl = tmpl "| sort -k 1,1 -k 2,2 -k 3,3n"
cmd = sprintf(tmpl, query, db, evalue, "t")
```

During traversal of the BLAST results, we sum the number of nucleotides hit one or more times in the subject.

36b ⟨*Traverse BLAST results, P. 4.1* 36b⟩≡ (35e 37b)
```
s = 0
qstart = 0
qend = -1
while (cmd | getline) {
  ⟨Analyze BLAST hit, P. 4.1 36c⟩
}
close(cmd)
```

Each BLAST hit either extends an existing interval on the query or starts a new interval. Starting a new interval implies closure of a previous one, at which point the number of nucleotides contained in that interval is added to the sum.

36c ⟨*Analyze BLAST hit, P. 4.1* 36c⟩≡ (36b)
```
if (sacc == $1 && qacc == $2) {
  if ($3 <= qend && $4 > qend)
    qend = $4
} else {
  s += qend - qstart + 1
  sacc = $1
  qacc = $2
  qstart = $3
  qend = $4
}
```

To compute the final value of t_{p}, we take the sum so far and add the nucleotides from the last BLAST hit.

36d ⟨*Compute* t_{p}*, P. 4.1* 36d⟩≡ (35e)
```
tp = s + qend - qstart + 1
```

The count of false negatives nucleotides is the difference between the observed t_p and its maximum value.

37a ⟨*Compute f_n, P. 4.1* 37a⟩≡ (35e)
```
for (a in ql)
  m += ql[a] * nt
fn = m - tp
```

To compute the false positive and true negative nucleotides, we first construct the BLAST command to filter for *neighbors*, parse its results, and then compute the desired quantities.

37b ⟨*Compute f_p and t_n, P. 4.1* 37b⟩≡ (34)
```
cmd = sprintf(tmpl, query, db, evalue, "n")
⟨Traverse BLAST results, P. 4.1 36b⟩
⟨Compute f_p, P. 4.1 37c⟩
⟨Compute t_n, P. 4.1 37d⟩
```

The false positive nucleotides are the sum coming out of the traversal of the BLAST results plus the nucleotides in the last hit.

37c ⟨*Compute f_p, P. 4.1* 37c⟩≡ (37b)
```
fp = s + qend - qstart + 1
```

The true negatives are the difference to the maximum value f_p could take.

37d ⟨*Compute t_n, P. 4.1* 37d⟩≡ (37b)
```
m = 0
for (a in ql)
  m += ql[a] * nn
tn = m - fp
```

Now we use equation (4.1) to compute the sensitivity, equation (4.2) for the specificity, and equation (4.3) for their correlation.

37e ⟨*Print S_n, S_p, and C, P. 4.1* 37e⟩≡ (34)
```
sn = tp / (tp + fn)
sp = tp / (tp + fp)
d = tp * tn - fp * fn
n = (tp + fp) * (tn + fn) * (tn + fp) * (tp + fn)
n = sqrt(n)
c  = d / n
print "#S_n\tS_p\tC"
printf "%.3f\t%.3f\t%.3f\n", sn, sp, c
```

The computation of sensitivity and specificity works best if we clean the sequences first by removing runs of `N`s using `cleanSeq`.

Chapter 5

Convert Unique Regions to Input for `primer3`, `fur2prim`

5.1 Introduction

Primers are designed in several steps. First, `fur` identifies diagnostic regions in a template sequence. Then a program for designing primers, for example `primer3` is used to find primer pairs in the diagnostic regions. However, converting the output of `fur` to `primer3` input can be tricky and `fur2prim` is designed to automate this.

5.2 Implementation

`fur2prim` reads `fur` output and prints a text file for driving a `primer3` run. The program first prints a usage message, if so desired, and then the `primer3` input.

Program 5.1 (`fur2prim`)**.**

40a ⟨*fur2prim* 40a⟩≡

```
#!/usr/bin/awk -f
BEGIN {
  ⟨Print usage, P. 5.1 40b⟩
}
{
  ⟨Parse template sequence, P. 5.1 41a⟩
}
END {
  ⟨END action, P. 5.1 42d⟩
}
```

There is no mandatory input, but there are a number of parameters like oligo and product length, and melting temperature to be specified as we work through the implementation. These can be specified by the user or left in their default state.

40b ⟨*Print usage, P. 5.1* 40b⟩≡ (40a)

```
⟨Define default parameter values, P. 5.1 41e⟩
if (h || help) {
  print "fur2prim: Convert fur output to primer3 input"
  print "Usage: fur2prim furOutput.fasta"
  ⟨Query parameter values, P. 5.1 42a⟩
  ex = 1
  exit
}
⟨Assign parameter values, P. 5.1 42b⟩
```

The `fur` output consists of FASTA formatted unique regions extracted from the target representative. For each unique region a `primer3` entry is printed terminated by `=`.

41a ⟨*Parse template sequence, P. 5.1* 41a⟩≡ (40a)
```
if (/^>/) {
  if (n) {
    ⟨Print primer3 input, P. 5.1 41b⟩
    print "="
  }
  seq = ""
  n++
} else
  seq = seq $0
```

The input for `primer3` consists of two parts, one constant, the other variable.

41b ⟨*Print* `primer3` *input, P. 5.1* 41b⟩≡ (41a 42d)
```
⟨Print constant primer3 input, P. 5.1 41c⟩
⟨Print variable primer3 input, P. 5.1 41d⟩
```

In the constant input we request the construction of pairs of primers, each augmented by an internal oligo.

41c ⟨*Print constant* `primer3` *input, P. 5.1* 41c⟩≡ (41b)
```
print "PRIMER_TASK=generic"
print "PRIMER_PICK_LEFT_PRIMER=1"
print "PRIMER_PICK_RIGHT_PRIMER=1"
print "PRIMER_PICK_INTERNAL_OLIGO=1"
```

The variable input concerns first of all the primer and product size.

41d ⟨*Print variable* `primer3` *input, P. 5.1* 41d⟩≡ (41b) 42c ▷
```
printf "PRIMER_MIN_SIZE=%d\n", primMinSize
printf "PRIMER_MAX_SIZE=%d\n", primMaxSize
printf "PRIMER_PRODUCT_SIZE_RANGE=%d-%d\n", prodMinSize, prodMaxSize
printf "PRIMER_MIN_TM=%.1f\n", primMinTm
printf "PRIMER_MAX_TM=%.1f\n", primMaxTm
printf "PRIMER_INTERNAL_MIN_TM=%.1f\n", inMinTm
printf "PRIMER INTERNAL_MAX_TM=%.1f\n", inMaxTm
```

At the beginning of the program these parameters are given default values.

41e ⟨*Define default parameter values, P. 5.1* 41e⟩≡ (40b)
```
defPrimMinSize = 15
defPrimMaxSize = 25
defProdMinSize = 70
defProdMaxSize = 150
defPrimMinTm = 54
defPrimMaxTm = 58
defInMinTm = 43
defInMaxTm = 47
```

Later, they are queried.

42a ⟨*Query parameter values, P. 5.1* 42a⟩≡ (40b)
```
printf "\t[-v primMinSize=<S>; default: %d]\n",  defPrimMinSize
printf "\t[-v primMaxSize=<S>; default: %d]\n",  defPrimMaxSize
printf "\t[-v prodMinSize=<S>; default: %d]\n",  defProdMinSize
printf "\t[-v prodMaxSize=<S>; default: %d]\n",  defProdMaxSize
printf "\t[-v primMinTm=<T>; default: %.1f]\n", defPrimMinTm
printf "\t[-v primMaxTm=<T>; default: %.1f]]\n", defPrimMaxTm
printf "\t[-v inMinTm=<T>; default: %.1f]\n", defInMinTm
printf "\t[-v inMaxTm=<T>; default: %.1f]\n", defInMaxTm
```

Any as yet undefined parameter is assigned its default value.

42b ⟨*Assign parameter values, P. 5.1* 42b⟩≡ (40b)
```
if (!primMinSize) primMinSize = defPrimMinSize
if (!primMaxSize) primMaxSize = defPrimMaxSize
if (!prodMinSize) prodMinSize = defProdMinSize
if (!prodMaxSize) prodMaxSize = defProdMaxSize
if (!primMinTm) primMinTm = defPrimMinTm
if (!primMaxTm) primMaxTm = defPrimMaxTm
if (!inMinTm) inMinTm = defInMinTm
if (!inMaxTm) inMaxTm = defInMaxTm
```

As the last step in the construction of the input for `primer3`, the template sequence is appended.

42c ⟨*Print variable* `primer3` *input, P. 5.1* 41d⟩+≡ (41b) ◁41d
```
printf "SEQUENCE_TEMPLATE=%s\n", seq
```

When the program enters the END block, it might do so after an `exit` in the BEGIN block. In that case it exits again. Otherwise, the last entry is printed, unless there was no input.

42d ⟨*END action, P. 5.1* 42d⟩≡ (40a)
```
if (ex)
  exit
if (n) {
  ⟨Print primer3 input, P. 5.1 41b⟩
  print "="
}
```

Chapter 6

Extract Primers from `primer3` Output, `prim2fasta`

6.1 Introduction

The program `primer3` generates output in its own format. However, primer sequences are subsequently often checked using BLAST, which requires input in FASTA format. The program `prim2fasta` extracts primer pairs from `primer3` output and writes each pair in a separate file for subsequent checking.

6.2 Implementation

The program requests the base name of the output files, `b`, and then prints each primer-pair in a file called `b1.fasta`, `b2.fasta`, and so on.

Program 6.1 (`prim2fasta`)**.**

44a ⟨*prim2fasta* 44a⟩≡

```
#!/usr/bin/awk -f
BEGIN {
  ⟨Request base name, P. 6.1 44b⟩
}
⟨Extract forward primer, P. 6.1 44c⟩
⟨Extract reverse primer, P. 45⟩
```

If no base name is supplied, the user is prompted for one.

44b ⟨*Request base name, P. 6.1* 44b⟩≡ (44a)

```
if (!file) {
  print "prim2fasta: Extract primer sequences from primer3 output"
  print "Usage: prim2fasta -v file=<fileBaseName> primer3.out"
  exit
}
```

The forward primer is reported as, for example

```
PRIMER_LEFT_0_SEQUENCE=TTCTGTATCGTTTCTCCA
```

It is printed before the reverse primer, so encountering a forward primer opens a new file.

44c ⟨*Extract forward primer, P. 6.1* 44c⟩≡ (44a)

```
/PRIMER_LEFT_.*_SEQUENCE/ {
  n++
  f = file n ".fasta"
  print "Writing", f
  printf ">f%d\n", n > f
  split($1, a, "=")
  printf "%s\n", a[2] >> f
}
```

The reverse primer is extracted in a similar way, except that now the output file is closed rather than opened.

45 ⟨*Extract reverse primer, P. 6.1* 45⟩≡ (44a)

```
/PRIMER_RIGHT_.*_SEQUENCE/ {
  printf ">r%d\n", n >> f
  split($1, a, "=")
  printf "%s\n", a[2] >> f
  close(f)
}
```

Chapter 7

Check Primers, `checkPrim`

7.1 Introduction

PCR primers designed to amplify a specific region may also unintentionally amplify other regions in the same genome or in the genomes of other organisms. To guard against such off-target amplification, primers are compared to a suitable sequence database and all potential amplification products in a particular set of organisms reported. There already exists an excellent web-based program to do this, Primer-BLAST. While Primer-BLAST is intended foremost as a tool for designing primers from scratch, it also contains a module for checking primer specificity. However, running programs over the internet is usually less convenient than running them locally. Our aim is therefore to write a stand-alone version of this module, `checkPrim`.

7.2 Implementation

`checkPrim` takes as input a set of primers, a BLAST database, and an organism identified by one or more NCBI taxon-IDs. It returns the virtual PCR products, or amplicons, found in members of the taxa concerned. It can also do the opposite, return the amplicons found outside the members of the focal taxa.

`checkPrim` first switches the field separator from default white space to tab. It then interacts with the user, sets optional parameters to their default values, and constructs the BLAST command for looking up the primer matches. Finally, it analyzes the BLAST results to look for potential amplicons.

Program 7.1 (`checkPrim`)**.**

48 ⟨*checkPrim* 48⟩≡

```
#!/usr/bin/awk -f
BEGIN {
  FS = "\t"
  ⟨Interact with user, P. 7.1 49a⟩
  ⟨Set default values of optional parameters, P. 7.1 50a⟩
  ⟨Construct BLAST command, P. 7.1 50b⟩
  ⟨Save BLAST results, P. 7.1 51a⟩
  ⟨Analyze BLAST results, P. 7.1 52a⟩
}
```

The user is asked to supply three parameters: A file containing one or more primers (`query`), a BLAST database (`db`), and at least one taxon-ID, which is interpreted either as the target (`taxids`) or its complement (`negativeTaxids`). The taxon-IDs can also be supplied as files (`taxidList` and `negativeTaxidList`). If one of the three required parameters is not supplied, a usage message is printed prompting for complete input. In addition, optional parameters can be set.

49a ⟨*Interact with user, P. 7.1* 49a⟩≡ (48)

```
⟨Initialize default values of optional parameters, P. 7.1 49c⟩
if (!query || !db || !(taxids || negativeTaxids ||
                       taxidList || negativeTaxidList)) {
  print "checkPrim: Check the specificity of PCR primers using BLAST"
  print "Usage: checkPrim <options>"
  print "Options:"
  print  "\t-v query=<query>"
  print  "\t-v db=<db>"
  ⟨Print taxon-ID options, P. 7.1 49b⟩
  ⟨Query optional parameters, P. 7.1 49d⟩
  exit 0
}
```

The user can set taxon-IDs with comma-delimited lists or with files.

49b ⟨*Print taxon-ID options, P. 7.1* 49b⟩≡ (49a)

```
printf "\t-v taxids=<t1,t2...> || "
printf "-v negativeTaxids=<t1,t2...> || "
printf "-v taxidList=<foo.txt> || "
printf "-v negativeTaxidList=<foo.txt>\n"
```

There are four optional parameters: The maximum number of mismatches (`maxMism`), the maximum length of an amplicon (`maxLen`), the number of threads used by BLAST (`numThreads`), and the E-value (`evalue`). I took the default maximum number of mismatches and the maximum amplicon size from the Primer-BLAST website, and the E-value from the documentation of stand-alone `blastn`.

49c ⟨*Initialize default values of optional parameters, P. 7.1* 49c⟩≡ (49a)

```
defMaxMism = 5
defMaxLen = 4000
defNumThreads = 1
defEvalue = 10
```

The optional parameters are queried using the standard `-v` notation of AWK.

49d ⟨*Query optional parameters, P. 7.1* 49d⟩≡ (49a)

```
printf "\t[-v maxMism=<maxMism>; default: %d]\n", defMaxMism
printf "\t[-v maxLen=<maxLen>; default: %d]\n", defMaxLen
printf "\t[-v numThreads=<numThreads>; default: %d]\n",
  defNumThreads
printf "\t[-v evalue=<evalue>; default: %d]\n", defEvalue
```

Any undefined optinal parameters are set to their default values.

50a ⟨*Set default values of optional parameters, P. 7.1* 50a⟩≡ (48)

```
if (!maxMism)
  maxMism = defMaxMism
if (!maxLen)
  maxLen = defMaxLen
if (!numThreads)
  numThreads = defNumThreads
if (!evalue)
  evalue = defEvalue
```

The BLAST search is based on the `blastn-short` mode of the `blastn` program. This mode is optimized for sequences shorter than 50 nucleotides. We also set the BLAST database and deal with the taxon-IDs the user supplied. We return five aspects of each BLAST hit (`-ioutfmt`): The query and subject accessions, the number of mismatches, the start and end positions on the subject, and two items of taxonomic information on the subject: the taxon id and its scientific name. In addition, we set the `outfmt` option such that the lengths of the query and the alignment are printed. This allows us to filter for full-length matches, that is, we are looking for alignment that are global in the query and local in the subject. By default, BLAST results are sorted first by the input order of the query—all matches of the first query followed by all matches to the second, and so on—and then by their subject position. For identifying spurious amplicons, it is more convenient to group the results by subject and then sort by position within each subject.

50b ⟨*Construct BLAST command, P. 7.1* 50b⟩≡ (48)

```
tmpl = "blastn -task blastn-short -query %s -db %s "
⟨Deal with taxon-IDs, P. 7.1 50c⟩
tmpl = tmpl "-outfmt \"6 qacc sacc mismatch sstart send staxid "
tmpl = tmpl "qlen length ssciname\" -num_threads %d -evalue %d "
tmpl = tmpl "| awk '$3 <= %d && $7 == $8'"
tmpl = tmpl "| sort -k 2,2 -k 4,4n"
cmd = sprintf(tmpl, query, db, numThreads, evalue, taxid, maxMism)
```

The user can supply taxon-IDs in one of four formats: positive or negative comma-delimited lists, and files of positive or negative taxon-IDs.

50c ⟨*Deal with taxon-IDs, P. 7.1* 50c⟩≡ (50b)

```
if (taxids)
  tmpl = tmpl "-taxids " taxids " "
else if (negativeTaxids)
  tmpl = tmpl "-negative_taxids " negativeTaxids " "
else if (taxidList)
  tmpl = tmpl "-taxidlist " taxidList " "
else if (negativeTaxidList)
    tmpl = tmpl "-negative_taxidlist " negativeTaxidList " "
```

p_f 5’—TGACCGCCAATATTGCCAGT—3’
p_r 5’—TTCTTACGGGGAGACGCAAC—3’

---

p_f	5’→		TGACCGCCAATATTGCCAGT					→3’
t_f	5’→	...	TGACCGCCAATATTGCCAGT	...	GTTGCGTCTCCCCGTAAGAA	...		→3’
t_r	3’←	...	ACTGGCGGTTATAACGGTCA	...	CAACGCAGAGGGGCATTCTT	...		←5’
p_r	3’←				CAACGCAGAGGGGCATTCTT			←5’

Figure 7.1: Forward and reverse PCR primers, p_f and p_r (top panel), along the forward or reverse strands of a template, t_f and t_r (bottom panel).

The BLAST command is run and the results are saved.

51a ⟨*Save BLAST results, P. 7.1* 51a⟩≡ (48)

```
n = 1 - 1
while (cmd | getline) {
  qacc[n] = $1
  sacc[n] = $2
  staxid[n] = $6
  ssciname[n] = $9
  ⟨Decide strand, P. 7.1 51b⟩
  n++
}
close(cmd)
```

All PRC reactions take double-stranded DNA as template. To visualize the primer configuration we are looking for in our BLAST search, consider the forward and reverse primers p_f and p_r in the top panel of Figure 7.1. They bind the forward and reverse strands of a template, t_f and t_r. So regardless of which template strand has been sequenced, the 5’-primer of a potential amplicon is on the forward strand, the 3’-primer on the reverse strand.

BLAST encodes strand in the start and end positions of matches. If the start is less than the end, the match is on the forward strand; otherwise, the match is on the reverse strand. I find it more convenient to think of all matches in the customary 5’ to 3’ direction, which means I invert the coordinates of matches on the reverse strand and explicitly store the strand, 0 for forward, 1 for reverse.

51b ⟨*Decide strand, P. 7.1* 51b⟩≡ (51a)

```
if ($4 < $5) {
  sstart[n] = $4
  send[n] = $5
  strand[n] = 0
} else {
  sstart[n] = $5
  send[n] = $4
  strand[n] = 1
}
```

When iterating over the results, every 5' match on the forward strand is paired with all 3' matches on the reverse strand closer than the maximum amplicon length. Any such pair of primers is a potential amplicon and is reported with the subject accession, the start and end positions of the amplicon on that subject, and the subject taxonomy.

52a ⟨*Analyze BLAST results, P. 7.1* 52a⟩≡ (48)

```
⟨Print header, P. 7.1 52b⟩
for (i = 0; i < n - 1; i++) {
  j = i + 1
  l = send[j] - sstart[i] + 1
  while (sacc[i] == sacc[j] && j < n && l <= maxLen) {
    if (strand[i] == 0 && strand[j] == 1)
        ⟨Print result, P. 7.1 52c⟩
    j++
  }
}
```

The header is tab-delimited and marked by a hash in the first column.

52b ⟨*Print header, P. 7.1* 52b⟩≡ (52a)

```
printf "# qacc\tqacc\tsacc\tsstart\tsend\tstaxid\t"
print  "ssciname"
```

Each row of results is also printed as a tab-delimited row.

52c ⟨*Print result, P. 7.1* 52c⟩≡ (52a)

```
printf("%s\t%s\t%s\t%d\t%d\t%d\t%s\n",
        qacc[i], qacc[j], sacc[i], sstart[i], send[j], staxid[i],
        ssciname[i])
```

We can now use `checkPrim` to check pairs of primers. An example application is shown in Section 8.4.

Chapter 8

Tutorial

8.1 fur

To demonstrate fur, it is used to find regions specific to the pathogenic *E. coli* strain ST131 in the example data shown in Figure 2.1. The first step is to get the data. This is converted into a fur database and analyzed in an initial pass before the investigation is refined by varying the parameters of fur.

Program 8.1 (furTut.sh)**.**

54a ⟨*furTut.sh* 54a⟩≡
⟨*Get tutorial data, P. 8.1* 54b⟩
⟨*Make fur database, P. 8.1* 54c⟩
⟨*Analyze tutorial data, P. 8.1* 54d⟩
⟨*Refine tutorial analysis, P. 8.1* 55b⟩

The example data is copied from a networked computer and unpacked.

54b ⟨*Get tutorial data, P. 8.1* 54b⟩≡ (54a)
```
wget guanine.evolbio.mpg.de/fur/eco105.tar.gz
tar -xvzf eco105.tar.gz
```

This generates two directories of genomes in FASTA format, targets with 98 genomes, and neighbors with seven. These are converted to a fur database using makeFurDb (Chapter 1), which takes approximately half a minute.

54c ⟨*Make fur database, P. 8.1* 54c⟩≡ (54a)
```
makeFurDb -t targets -n neighbors -d furDb
```

Unique templates are found by applying fur to this database, which takes roughly fifteen seconds. The template sequences are stored in tmpl.fasta.

54d ⟨*Analyze tutorial data, P. 8.1* 54d⟩≡ (54a) 54e▷
```
fur -d furDb > tmpl.fasta
# Step                   Sequences  Nucleotides  Mutations (N)
# ------------------------------------------------------------
# Sliding window              1005       681264             0
# Presence in targets          170        69407           151
# Absence from neighbors        91        46844          4309
```

The hash-tagged progress information lists the three steps of the algorithm and the number of sequences and nucleotides contained in the template set after each one. So the initial sliding window analysis uncovers 1005 sequences totaling 681.3 kb and containing no unknown nucleotide, N. After checking for presence in the targets, 170 sequences with 69.4 kb remain. This step is carried out by intersecting the result of the previous step with the targets using phylonium (Figure 2.4). Here the matches between the reference sequence and the targets may contain mutations. Hence our result set is now sprinkled with 151 mutations (Ns). The final step of subtracting the neighbors leaves 91 sequences with 46.8 kb as the template set. In this step BLAST hits are set to N, hence the number of "mutations" has now grown to 4309. That is, 4.3 kb of N in addition to 46.8 kb of non-N.

The file tmpl.fasta consists of headers followed by sequence data. Each header in turn consists of a name and the start and end positions on the target representative. Get the first ten lines of tmpl.fasta.

54e ⟨*Analyze tutorial data, P. 8.1* 54d⟩+≡ (54a) ◁54d 55a▷
```
head tmpl.fasta
```

They happen to be:

```
>part9  (1436..1926)  2
CCCGGATTACAGTTCATAGGGTGTGACGACACTATCTCTCGTATTCCGCGTTACCTCCTCAAGCTATGCC
GCCAGTGCCTGTTTGGCCTCAATTTTCACGCTGAGAAAATCGATAAGGATATCGATATCAAATAGCCAAA
TCTTTTTGCCATTACCATTTCTCGCGCAGCTTACGCNGGTATTCGACACCATCTTCCTCACACCCTTGCC
AGATGCCAGGAACACTGGTTCTTGCTGTTTGCGATTGATTTACCAGGTATTGATCTACCGCTTCCCGTAA
TAGGTCTGCACGCGGAAGATTACGCTGCACCTCAANATCATCAAGTTGCTTAATCACCTCATTCGATAAA
TCGAGTAAAATTCTGCTCATATCCATACCTGCCAGAGGGTTCATAAATATCTCGCCAATATCATTTTGAA
TCTATGGAGAGAAAAGTACCCTTGTCGAATCTTTAAAGAAAGCGCATTTACGCATCACTTTTTATTTTTG
>part10 (3700..3958)  0
GTCCAGATACAGCTTTTGATAGTTTATTATCCTGGATGATATCAGGAGCGATATCTATAAAGTTTATGCA
```

The header consists of a name, coordinates within the concatenated reference sequence, and the number of mutations.

The file `tmpl.fasta` is supposed to contain 91 sequences with 46844 nucleotides, 4309 `N`, totaling $46844 + 4309 = 51153$ residues. To check this is actually the case, we use our progam `count` to find the expected 91 templates with a total of 51.2 kb.

55a ⟨*Analyze tutorial data, P. 8.1* 54d⟩+≡ (54a) ◁54e

```
count tmpl.fasta
# 91 tmpl, 46844 nuc, 4309 N, 51153 total
```

To make the process of template selection more transparent, the `-u` option allows printing of the unique regions found in the sliding window analysis before exiting.

55b ⟨*Refine tutorial analysis, P. 8.1* 55b⟩≡ (54a) 55c▷

```
fur -d furDb -u > unique1.fasta
# Step                    Sequences  Nucleotides  Mutations (N)
# ------------------------------------------------------------
# Sliding window               1005       681264              0
```

The file `unique1.fasta` now contains 1005 sequences with 681,264, which is checked again.

55c ⟨*Refine tutorial analysis, P. 8.1* 55b⟩+≡ (54a) ◁55b 55d▷

```
count unique1.fasta
# 1005 tmpl, 681264 nuc, 0 N, 681264 total
```

Similarly, the 170 regions present in all targets can be inspected using the `-U` option.

55d ⟨*Refine tutorial analysis, P. 8.1* 55b⟩+≡ (54a) ◁55c 55e▷

```
fur -d furDb -U > unique2.fasta
# Step                    Sequences  Nucleotides  Mutations (N)
# ------------------------------------------------------------
# Sliding window               1005       681264              0
# Presence in targets           170        69407            151
```

Check that `unique2.fasta` contains 170 sequences with $69407 + 151 = 69558$ bp.

55e ⟨*Refine tutorial analysis, P. 8.1* 55b⟩+≡ (54a) ◁55d 56a▷

```
count unique2.fasta
# 170 tmpl, 69407 nuc, 151 N, 69558 total
```

Three `fur` parameters are of interest. The first is the window length; the other two affect the sensitivity of the BLAST run, which can be modulated via its mode (algorithm) and E-value. Let's begin with the window length, which by default is 80 bp. Much longer windows result in sequences that are more difficult to find among all targets. For example, with 1 kb windows, there are 111 candidate regions, of which 26 are present in all targets. The final tally is 18 regions with 18027 nucleotides and 2990 `N`. So the final yield is quite different in spite of the fact that the amount of nucleotides returned from the sliding window analysis, 635 kb, is similar to the 681 kb found with 80 bp windows.

56a ⟨*Refine tutorial analysis, P. 8.1* 55b⟩+≡ (54a) ◁55e 56b▷

```
fur -d furDb -w 1000 > tmpl.fasta
# Step                    Sequences  Nucleotides  Mutations (N)
# ------------------------------------------------------------
# Sliding window                111       634900             0
# Presence in targets            26        28730           108
# Absence from neighbors         18        18027          2990
```

On the other hand, a small increase in window length to 90 bp happens to yield 174 templates with 53.6 kb. Clearly, `fur` is highly sensitive to the window length and this should be borne in mind when investigating other pathogens.

56b ⟨*Refine tutorial analysis, P. 8.1* 55b⟩+≡ (54a) ◁56a 56c▷

```
fur -d furDb -w 90 > tmpl.fasta
# Step                    Sequences  Nucleotides  Mutations (N)
# ------------------------------------------------------------
# Sliding window               1610       609930             0
# Presence in targets           246        72184           167
# Absence from neighbors        174        53570          2922
```

The second parameter we explore is the mode of the BLAST-search among the neighborhood sequences. By default this is the sensitive "blastn" mode. Option `-m` switches it to the faster and less sensitive "megablast" mode. This increases the yield from originally 91 fragments with 48.8 kb to 168 fragments with 69.0 kb, a 40% jump, because fewer matches between candidates and neighbors are found. However, the remaining candidates might now be less specific.

56c ⟨*Refine tutorial analysis, P. 8.1* 55b⟩+≡ (54a) ◁56b 57a▷

```
fur -d furDb -m > tmpl.fasta
# Step                    Sequences  Nucleotides  Mutations (N)
# ------------------------------------------------------------
# Sliding window               1005       681264             0
# Presence in targets           170        69407           151
# Absence from neighbors        168        69052           137
```

The third and last parameter we explore is the E-value of the BLAST-search, which is 10^{-5} by default. When decreased to, say, 10^{-20}, the yield increases from 91 fragments / 46.8 kb to 102 fragments / 50.5 kb. Again, the candidates might now have lower specificity.

57a ⟨*Refine tutorial analysis, P. 8.1* 55b⟩+≡ (54a) ◁56c 57b▷

```
fur -d furDb -e 1e-20 > tmpl.fasta
# Step                    Sequences  Nucleotides  Mutations (N)
# ------------------------------------------------------------
# Sliding window               1005       681264             0
# Presence in targets           170        69407           151
# Absence from neighbors        102        50516          2573
```

So it could be important to vary the window length (`-w`), the E-value (`-e`), and the mode in your own analyses. This can be done conveniently, as each run of `fur` is reasonably fast once the underlying database has been computed.

8.2 Test Sensitivity and Specificity, `senSpec`

The sensitivity and specificity of the markers proposed by `fur` can be calculated by `senSpec`. (Section 4). The sequences are cleaned first, and then evaluated.

57b ⟨*Refine tutorial analysis, P. 8.1* 55b⟩+≡ (54a) ◁57a 57c▷

```
cleanSeq tmpl.fasta > tmpl2.fasta
senSpec -v query=tmpl2.fasta -v db=furDb
#S_n    S_p     C
1.000   0.995   0.962
```

Given that we are testing on the same data from which the templates were extracted, it is not surprising that all three quality scores are very high. Think of this as a positive control. A more interesting test would be on a database of sequences not used in template discovery.

8.3 Making Primers, `fur2prim` & `prim2fasta`

Each template is now converted to an entry in the input to `primer3`.

57c ⟨*Refine tutorial analysis, P. 8.1* 55b⟩+≡ (54a) ◁57b 57d▷

```
fur2prim tmpl.fasta > prim.txt
```

The command-line version of `primer3` is run on the input file just created.

57d ⟨*Refine tutorial analysis, P. 8.1* 55b⟩+≡ (54a) ◁57c 57e▷

```
primer3_core prim.txt > prim.out
```

57e ⟨*Refine tutorial analysis, P. 8.1* 55b⟩+≡ (54a) ◁57d 58a▷

```
prim2fasta -v file=primer prim.out
```

This generates the primer files

```
primer1.fasta
primer2.fasta
...
primer510.fasta
```

This is perhaps a bit too much. So instead of extracting all primers, let's rank them by quality. Each primer pair has a "PRIMER_PAIR_X_PENALTY", where X is 0 for the best pair of a particular template, 1 for the second best, and so on. We can extract and sort these values to discover the overall best primer pair.

58a ⟨*Refine tutorial analysis, P. 8.1* 55b⟩+≡ (54a) ◁57e

```
grep PRIMER_PAIR_0_PENALTY prim.out |
    tr '=' '\t' |
    sort -k 2 -n |
    head -n 1
```

This has penalty 0.054910, a number that can now be used to find the corresponding primers in `prim.out`.

8.4 Checking Primers, `checkPrim`

Primers are often checked by comparing them to the complete NCBI nucleotide database, `nt`. To avoid the overhead associated with handling this huge database, I constructed a smaller example for this Tutorial. The file `p.fa` contains a pair of candidate forward and reverse primers that might be diagnostic for SARS-CoV-2. To check their potential for spurious amplification, we need two BLAST databases, a sequence database, and the BLAST taxonomy database to classify any hits we might find in the sequence database. Then two questions are asked. First, does the primer pair amplify SARS-CoV-2? This is the positive control. And then, does it amplify anything else? This is the negative control.

Program 8.2 (`checkTut.sh`)**.**

58b ⟨*checkTut.sh* 58b⟩≡

⟨*Get BLAST sequence database, P. 8.2* 58c⟩
⟨*Get BLAST taxonomy database, P. 8.2* 59a⟩
⟨*Carry out positive control, P. 8.2* 59b⟩
⟨*Carry out negative control, P. 8.2* 59c⟩

The BLAST database needs to be housed in a suitable directory. In this tutorial we use the `data` directory that is part of this software package. Our data are Betacoronavirus sequences supplied by the NCBI. This is downloaded using the program

```
update_blastdb.pl
```

which is part of the BLAST package.

58c ⟨*Get BLAST sequence database, P. 8.2* 58c⟩≡ (58b)

```
cd data
update_blastdb.pl --decompress Betacoronavirus
```

The taxonomy database is downloaded in the same way. To make BLAST aware of its location, the `BLASTDB` environment variable is set. Once the BLAST database has been constructed, we return to the base directory of the package.

59a ⟨*Get BLAST taxonomy database, P. 8.2* 59a⟩≡ (58b)

```
update_blastdb.pl --decompress taxdb
export BLASTDB=$(pwd)
cd ..
```

For the positive control, we check that the candidate primers amplify a single region in SARS-CoV-2. The virus is identified by its taxon-ID, which can be looked up on the NCBI taxonomy web site and happens to be 2697049. If everything is working, a single interval is returned for most if not all of the many SARS-CoV-2 sequences contained in the database.

59b ⟨*Carry out positive control, P. 8.2* 59b⟩≡ (58b)

```
checkPrim -v query=data/p1.fa \
                    -v db=data/Betacoronavirus \
                    -v taxids=2697049
```

For the negative control, all hits to sequences not classified as SARS-CoV-2 are printed.

59c ⟨*Carry out negative control, P. 8.2* 59c⟩≡ (58b)

```
checkPrim -v query=data/p1.fa \
                    -v db=data/Betacoronavirus \
                    -v negativeTaxids=2697049
```

If there is no cross-amplification, or the spurious amplicons are found in acceptable taxa, the searches should be repeated in a larger database, ideally the complete collection of known nucleotide sequences, `nt`. The full list of available databases is show by

```
update_blastdb.pl --showall
```

While we are primarily interested in spotting "wrong" amplicons, that is, in the results of the negative control, it is a good idea to always also perform the positive control to make sure the primers can actually be found in the test database.

List of code chunks

⟨*Set default values of optional parameters, P. 7.1* 50a⟩
⟨*Analyze tutorial data, P. 8.1* 54d⟩
⟨*Carry out negative control, P. 8.2* 59c⟩
⟨*Carry out positive control, P. 8.2* 59b⟩
⟨*checkTut.sh* 58b⟩
⟨*furTut.sh* 54a⟩
⟨*Get BLAST sequence database, P. 8.2* 58c⟩
⟨*Get BLAST taxonomy database, P. 8.2* 59a⟩
⟨*Get tutorial data, P. 8.1* 54b⟩
⟨*Make fur database, P. 8.1* 54c⟩
⟨*Refine tutorial analysis, P. 8.1* 55b⟩

Bibliography

[1] S. F. Altschul, T. L. Madden, A. A. Schäffer, J. Zhang, Z. Zhang, W. Miller, and D. Lipman. Gapped blast and psi-blast: a new generation of protein database search programs. *Nucleic Acids Research*, 25:3389–3402, 1997.

[2] B. Haubold, F. Klötzl, and P. Pfaffelhuber. `andi`: Fast and accurate estimation of evolutionary distances between closely related genomes. *Bioinformatics*, 31:1169–75, 2015.

[3] B. Haubold and T. Wiehe. *Introduction to Computational Biology: An Evolutionary Approach*. Birkhäuser, Basel, 2006.

[4] Fabian Klötzl and Bernhard Haubold. Phylonium: fast estimation of evolutionary distances from large samples of similar genomes. *Bioinformatics*, 36:2040–46, 2020.

[5] N. K. Petty, N. L. Ben Zakour, M. Stanton-Cook, E. Skippington, M. Totsika, B. M. Forde, M.-D. Phan, D. Gomes Moriel, K. M. Peters, M. Davies, B. A. Rogers, G. Dougan, J. Rodriguez-Bañ, A. Pascual, J. D. D. Pitout, M. Upton, D. L. Paterson, T. R. Walsh, M. A. Schembri, and S. A. Beatson. Global dissemination of a multidrug resistant *Escherichia coli* clone. *Proceedings of the National Academy of Sciences, USA*, 111:5694–5699, 2014.

[6] A. Pirogov, P. Pfaffelhuber, A. G. Börsch-Haubold, and B. Haubold. High-complexity regions in mammalian genomes are enriched for developmental genes. *Bioinformatics*, 2018.

[7] A. Untergasser, I. Cutcutache, T. Koressaar, J. Ye, B. C. Faircloth, M. Remm, and S. G. Rozen. Primer3—new capabilities and interfaces. *Nucleic Acids Research*, 40:e115, 2012.