Doxygen test

Generated by Doxygen 1.9.3

1 README 1

2 Class Index 3

2.1 Class List . . . 3

3 File Index 5

3.1 File List . . . 5

4 Class Documentation 7

4.1 gkn_FASTA Struct Reference . . . 7

4.1.1 Detailed Description . . . 7

4.1.2 Member Data Documentation . . . 7

4.1.2.1 def . . . 7

4.1.2.2 length . . . 8

4.1.2.3 seq . . . 8

4.2 gkn_FVEC Struct Reference . . . 8

4.2.1 Detailed Description . . . 8

4.2.2 Member Data Documentation . . . 8

4.2.2.1 elem . . . 8

4.2.2.2 last . . . 9

4.2.2.3 limit . . . 9

4.2.2.4 size . . . 9

4.3 gkn_IVEC Struct Reference . . . 9

4.3.1 Detailed Description . . . 9

4.3.2 Member Data Documentation . . . 9

4.3.2.1 elem . . . 10

4.3.2.2 last . . . 10

4.3.2.3 limit . . . 10

4.3.2.4 size . . . 10

4.4 gkn_LEN Struct Reference . . . 10

4.4.1 Detailed Description . . . 10

4.4.2 Member Data Documentation . . . 11

4.4.2.1 name . . . 11

4.4.2.2 score . . . 11

4.4.2.3 size . . . 11

4.4.2.4 tail . . . 11

4.5 gkn_MAP Struct Reference . . . 11

4.5.1 Detailed Description . . . 12

4.5.2 Member Data Documentation . . . 12

4.5.2.1 key . . . 12

4.5.2.2 keys . . . 12

4.5.2.3 level . . . 12

4.5.2.4 slots . . . 12

4.5.2.5 val . . . 12
4.5.2.6 vals . . . 13
4.6 gkn_MM Struct Reference . . . 13
4.6.1 Detailed Description . . . 13
4.6.2 Member Data Documentation . . . 13
4.6.2.1 k . . . 13
4.6.2.2 name . . . 13
4.6.2.3 score . . . 14
4.6.2.4 size . . . 14
4.7 gkn_PIPE Struct Reference . . . 14
4.7.1 Detailed Description . . . 14
4.7.2 Member Data Documentation . . . 14
4.7.2.1 gzip . . . 14
4.7.2.2 mode . . . 15
4.7.2.3 name . . . 15
4.7.2.4 stream . . . 15
4.8 gkn_PWM Struct Reference . . . 15
4.8.1 Detailed Description . . . 15
4.8.2 Member Data Documentation . . . 15
4.8.2.1 name . . . 16
4.8.2.2 score . . . 16
4.8.2.3 size . . . 16
4.9 gkn_TMAP Struct Reference . . . 16
4.9.1 Detailed Description . . . 16
4.9.2 Member Data Documentation . . . 16
4.9.2.1 hash . . . 16
4.9.2.2 tvec . . . 17
4.10 gkn_TVEC Struct Reference . . . 17
4.10.1 Detailed Description . . . 17
4.10.2 Member Data Documentation . . . 17
4.10.2.1 elem . . . 17
4.10.2.2 last . . . 17
4.10.2.3 limit . . . 18
4.10.2.4 size . . . 18
4.11 gkn_VEC Struct Reference . . . 18
4.11.1 Detailed Description . . . 18
4.11.2 Member Data Documentation . . . 18
4.11.2.1 elem . . . 18
4.11.2.2 last . . . 19
4.11.2.3 limit . . . 19
4.11.2.4 size . . . 19
4.12 gkn_XNODE Struct Reference . . . 19

4.12.1 Detailed Description . . . 19
4.12.2 Member Data Documentation . . . 19
4.12.2.1 c . . . 20
4.12.2.2 children . . . 20
4.12.2.3 data . . . 20
4.13 gkn_xtree Struct Reference . . . 20
4.13.1 Detailed Description . . . 20
4.13.2 Member Data Documentation . . . 20
4.13.2.1 alloc . . . 20
4.13.2.2 head . . . 20

5 File Documentation **21**
5.1 model.c . . . 21
5.2 model.h . . . 23
5.3 sequence.c . . . 24
5.4 sequence.h . . . 26
5.5 toolbox.c . . . 26
5.6 toolbox.h . . . 34

Index **37**

Chapter 1

README

Chapter 2

Class Index

2.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

gkn_FASTA
Struct contains the sequence length, sequence name, and the sequence as a string 7
gkn_FVEC 8
gkn_IVEC 9
gkn_LEN 10
gkn_MAP 11
gkn_MM 13
gkn_PIPE 14
gkn_PWM 15
gkn_TMAP 16
gkn_TVEC 17
gkn_VEC 18
gkn_XNODE 19
gkn_xtree 20

Chapter 3

File Index

3.1 File List

Here is a list of all documented files with brief descriptions:

model.c . . . . . ??
model.h . . . . . ??
sequence.c . . . . . ??
sequence.h . . . . . ??
toolbox.c . . . . . ??
toolbox.h . . . . . ??

Chapter 4

Class Documentation

4.1 gkn_FASTA Struct Reference

struct contains the sequence length, sequence name, and the sequence as a string

`#include <sequence.h>`

Public Attributes

- int length
- char * def
- char * seq

4.1.1 Detailed Description

struct contains the sequence length, sequence name, and the sequence as a string

Definition at line 17 of file sequence.h.

4.1.2 Member Data Documentation

4.1.2.1 def

`char* gkn_FASTA::def`

Definition at line 19 of file sequence.h.

4.1.2.2 length

```
int gkn_FASTA::length
```

Definition at line 18 of file sequence.h.

4.1.2.3 seq

```
char* gkn_FASTA::seq
```

Definition at line 20 of file sequence.h.

The documentation for this struct was generated from the following file:

- sequence.h

4.2 gkn_FVEC Struct Reference

Public Attributes

- double * elem
- int size
- int limit
- int last

4.2.1 Detailed Description

Definition at line 44 of file toolbox.h.

4.2.2 Member Data Documentation

4.2.2.1 elem

```
double* gkn_FVEC::elem
```

Definition at line 45 of file toolbox.h.

4.2.2.2 last

```
int gkn_FVEC::last
```

Definition at line 48 of file toolbox.h.

4.2.2.3 limit

```
int gkn_FVEC::limit
```

Definition at line 47 of file toolbox.h.

4.2.2.4 size

```
int gkn_FVEC::size
```

Definition at line 46 of file toolbox.h.

The documentation for this struct was generated from the following file:

- toolbox.h

4.3 gkn_IVEC Struct Reference

Public Attributes

- int * elem
- int size
- int limit
- int last

4.3.1 Detailed Description

Definition at line 31 of file toolbox.h.

4.3.2 Member Data Documentation

4.3.2.1 elem

```
int* gkn_IVEC::elem
```

Definition at line 32 of file toolbox.h.

4.3.2.2 last

```
int gkn_IVEC::last
```

Definition at line 35 of file toolbox.h.

4.3.2.3 limit

```
int gkn_IVEC::limit
```

Definition at line 34 of file toolbox.h.

4.3.2.4 size

```
int gkn_IVEC::size
```

Definition at line 33 of file toolbox.h.

The documentation for this struct was generated from the following file:

- toolbox.h

4.4 gkn_LEN Struct Reference

Public Attributes

- char * name
- int size
- double * score
- double tail

4.4.1 Detailed Description

Definition at line 45 of file model.h.

4.4.2 Member Data Documentation

4.4.2.1 name

```
char* gkn_LEN::name
```

Definition at line 46 of file model.h.

4.4.2.2 score

```
double* gkn_LEN::score
```

Definition at line 49 of file model.h.

4.4.2.3 size

```
int gkn_LEN::size
```

Definition at line 47 of file model.h.

4.4.2.4 tail

```
double gkn_LEN::tail
```

Definition at line 50 of file model.h.

The documentation for this struct was generated from the following file:

- model.h

4.5 gkn_MAP Struct Reference

Public Attributes

- int level
- int slots
- gkn_tvec keys
- gkn_vec vals
- gkn_vec * key
- gkn_vec * val

4.5.1 Detailed Description

Definition at line 83 of file toolbox.h.

4.5.2 Member Data Documentation

4.5.2.1 key

```
gkn_vec* gkn_MAP::key
```

Definition at line 88 of file toolbox.h.

4.5.2.2 keys

```
gkn_tvec gkn_MAP::keys
```

Definition at line 86 of file toolbox.h.

4.5.2.3 level

```
int gkn_MAP::level
```

Definition at line 84 of file toolbox.h.

4.5.2.4 slots

```
int gkn_MAP::slots
```

Definition at line 85 of file toolbox.h.

4.5.2.5 val

```
gkn_vec* gkn_MAP::val
```

Definition at line 89 of file toolbox.h.

4.5.2.6 vals

```
gkn_vec gkn_MAP::vals
```

Definition at line 87 of file toolbox.h.

The documentation for this struct was generated from the following file:

- toolbox.h

4.6 gkn_MM Struct Reference

Public Attributes

- char * name
- int k
- int size
- double * score

4.6.1 Detailed Description

Definition at line 30 of file model.h.

4.6.2 Member Data Documentation

4.6.2.1 k

```
int gkn_MM::k
```

Definition at line 32 of file model.h.

4.6.2.2 name

```
char* gkn_MM::name
```

Definition at line 31 of file model.h.

4.6.2.3 score

```
double* gkn_MM::score
```

Definition at line 34 of file model.h.

4.6.2.4 size

```
int gkn_MM::size
```

Definition at line 33 of file model.h.

The documentation for this struct was generated from the following file:

- model.h

4.7 gkn_PIPE Struct Reference

Public Attributes

- int mode
- char * name
- int gzip
- FILE * stream

4.7.1 Detailed Description

Definition at line 143 of file toolbox.h.

4.7.2 Member Data Documentation

4.7.2.1 gzip

```
int gkn_PIPE::gzip
```

Definition at line 146 of file toolbox.h.

4.7.2.2 mode

```
int gkn_PIPE::mode
```

Definition at line 144 of file toolbox.h.

4.7.2.3 name

```
char* gkn_PIPE::name
```

Definition at line 145 of file toolbox.h.

4.7.2.4 stream

```
FILE* gkn_PIPE::stream
```

Definition at line 147 of file toolbox.h.

The documentation for this struct was generated from the following file:

- toolbox.h

4.8 gkn_PWM Struct Reference

Public Attributes

- char * name
- int size
- double ** score

4.8.1 Detailed Description

Definition at line 18 of file model.h.

4.8.2 Member Data Documentation

4.8.2.1 name

```
char* gkn_PWM::name
```

Definition at line 19 of file model.h.

4.8.2.2 score

```
double** gkn_PWM::score
```

Definition at line 21 of file model.h.

4.8.2.3 size

```
int gkn_PWM::size
```

Definition at line 20 of file model.h.

The documentation for this struct was generated from the following file:

- model.h

4.9 gkn_TMAP Struct Reference

Public Attributes

- gkn_map hash
- gkn_tvec tvec

4.9.1 Detailed Description

Definition at line 101 of file toolbox.h.

4.9.2 Member Data Documentation

4.9.2.1 hash

```
gkn_map gkn_TMAP::hash
```

Definition at line 102 of file toolbox.h.

4.9.2.2 tvec

```
gkn_tvec gkn_TMAP::tvec
```

Definition at line 103 of file toolbox.h.

The documentation for this struct was generated from the following file:

- toolbox.h

4.10 gkn_TVEC Struct Reference

Public Attributes

- char ** elem
- int size
- int limit
- char * last

4.10.1 Detailed Description

Definition at line 57 of file toolbox.h.

4.10.2 Member Data Documentation

4.10.2.1 elem

```
char** gkn_TVEC::elem
```

Definition at line 58 of file toolbox.h.

4.10.2.2 last

```
char* gkn_TVEC::last
```

Definition at line 61 of file toolbox.h.

4.10.2.3 limit

```
int gkn_TVEC::limit
```

Definition at line 60 of file toolbox.h.

4.10.2.4 size

```
int gkn_TVEC::size
```

Definition at line 59 of file toolbox.h.

The documentation for this struct was generated from the following file:

- toolbox.h

4.11 gkn_VEC Struct Reference

Public Attributes

- void ** elem
- int size
- int limit
- void * last

4.11.1 Detailed Description

Definition at line 70 of file toolbox.h.

4.11.2 Member Data Documentation

4.11.2.1 elem

```
void** gkn_VEC::elem
```

Definition at line 71 of file toolbox.h.

4.11.2.2 last

```
void* gkn_VEC::last
```

Definition at line 74 of file toolbox.h.

4.11.2.3 limit

```
int gkn_VEC::limit
```

Definition at line 73 of file toolbox.h.

4.11.2.4 size

```
int gkn_VEC::size
```

Definition at line 72 of file toolbox.h.

The documentation for this struct was generated from the following file:

- toolbox.h

4.12 gkn_XNODE Struct Reference

Public Attributes

- gkn_vec children
- void * data
- char c

4.12.1 Detailed Description

Definition at line 114 of file toolbox.h.

4.12.2 Member Data Documentation

4.12.2.1 c

```
char gkn_XNODE::c
```

Definition at line 117 of file toolbox.h.

4.12.2.2 children

```
gkn_vec gkn_XNODE::children
```

Definition at line 115 of file toolbox.h.

4.12.2.3 data

```
void* gkn_XNODE::data
```

Definition at line 116 of file toolbox.h.

The documentation for this struct was generated from the following file:

- toolbox.h

4.13 gkn_xtree Struct Reference

Public Attributes

- gkn_xnode head
- gkn_vec alloc

4.13.1 Detailed Description

Definition at line 124 of file toolbox.h.

4.13.2 Member Data Documentation

4.13.2.1 alloc

```
gkn_vec gkn_xtree::alloc
```

Definition at line 126 of file toolbox.h.

4.13.2.2 head

```
gkn_xnode gkn_xtree::head
```

Definition at line 125 of file toolbox.h.

The documentation for this struct was generated from the following file:

- toolbox.h

Chapter 5

File Documentation

5.1 model.c

```
00001 /*****************************************************************************\
00002  model.c
00003  Copyright (C) Ian Korf
00004 \*****************************************************************************/
00005
00006 #ifndef GENOMIKON_MODEL_C
00007 #define GENOMIKON_MODEL_C
00008
00009 #include "model.h"
00010
00011 double gkn_p2s(double p) {
00012     if (p == 0) return -100; // umm...
00013     return log(p/0.25) / log(2);
00014 }
00015
00016 // PWM
00017
00018 void gkn_pwm_free(gkn_pwm pwm) {
00019     free(pwm->name);
00020     for (int i = 0; i < pwm->size; i++) {
00021         free(pwm->score[i]);
00022     }
00023     free(pwm->score);
00024     free(pwm);
00025 }
00026
00027 gkn_pwm gkn_pwm_read(gkn_pipe io) {
00028     char    *line;
00029     char     name[256];
00030     int      size;
00031     double **score = NULL;
00032     double   a, c, g, t;
00033     int      row = 0;
00034
00035     while ((line = gkn_readline(io)) != NULL) {
00036         if (line[0] == '%') {
00037             assert(sscanf(line, "%% PWM %s %d", name, &size) == 2);
00038             score = malloc(sizeof(double*) * size);
00039             for (int i = 0; i < size; i++) {
00040                 score[i] = malloc(sizeof(double) * 4);
00041             }
00042             free(line);
00043         } else if (sscanf(line, "%lf %lf %lf %lf", &a, &c, &g, &t) == 4) {
00044             score[row][0] = gkn_p2s(a);
00045             score[row][1] = gkn_p2s(c);
00046             score[row][2] = gkn_p2s(g);
00047             score[row][3] = gkn_p2s(t);
00048             row++;
00049             free(line);
00050         } else {
00051             free(line);
00052         }
00053     }
00054
00055     gkn_pwm model = malloc(sizeof(struct gkn_PWM));
00056     model->name = malloc(strlen(name)+1);
00057     strcpy(model->name, name);
00058     model->size = size;
```

```
00059     model->score = score;
00060
00061     return model;
00062 }
00063
00064 double gkn_pwm_score(const gkn_pwm pwm, const char *seq, int pos) {
00065     double p = 0;
00066     for (int i = 0; i < pwm->size; i++) {
00067         switch (seq[i+pos]) {
00068             case 'A': case 'a': p += pwm->score[i][0]; break;
00069             case 'C': case 'c': p += pwm->score[i][1]; break;
00070             case 'G': case 'g': p += pwm->score[i][2]; break;
00071             case 'T': case 't': p += pwm->score[i][3]; break;
00072         }
00073     }
00074     return p;
00075 }
00076
00077 // Markov model
00078
00079 void gkn_mm_free(gkn_mm mm) {
00080     free(mm->name);
00081     free(mm->score);
00082     free(mm);
00083 }
00084
00085 gkn_mm gkn_mm_read(gkn_pipe io) {
00086     char    *line = NULL;
00087     double  *score = NULL;
00088     char     kmer[16];
00089     char     name[256];
00090     int      size;
00091     double   p;
00092
00093     while ((line = gkn_readline(io)) != NULL) {
00094         if (line[0] == '%') {
00095             assert(sscanf(line, "%% MM %s %d", name, &size) == 2);
00096             score = malloc(sizeof(double) * size);
00097             free(line);
00098         } else if (sscanf(line, "%s %lf", kmer, &p) == 2) {
00099             int idx = gkn_ntindex(kmer, 0, strlen(kmer));
00100             if (idx == -1) gkn_exit("alphabet error in: %s", kmer);
00101             score[idx] = gkn_p2s(p);
00102             free(line);
00103         } else {
00104             free(line);
00105         }
00106     }
00107
00108     gkn_mm model = malloc(sizeof(struct gkn_MM));
00109     model->name = malloc(strlen(name)+1);
00110     strcpy(model->name, name);
00111     model->k = strlen(kmer);
00112     model->size = size;
00113     model->score = score;
00114
00115     return model;
00116 }
00117
00118 double gkn_mm_score(const gkn_mm mm, const char *seq, int pos, int end) {
00119     double p = 0;
00120     if (pos < mm->k) pos = mm->k;
00121     for (int i = pos; i < end - mm->k +2; i++) {
00122         int idx = gkn_ntindex(seq, i, mm->k);
00123         if (idx != -1) p += mm->score[idx];
00124     }
00125     return p;
00126 }
00127
00128 double * gkn_mm_cache(const gkn_mm mm, const char *seq) {
00129     int len = strlen(seq);
00130     double *score = malloc(sizeof(double) * len);
00131     for (int i = 0; i < mm->k; i++) score[i] = 0;
00132     for (int i = mm->k; i < len; i++) {
00133         int idx = gkn_ntindex(seq, i, mm->k);
00134         if (idx == -1) score[i] = score[i-1];
00135         else           score[i] = score[i-1] + mm->score[idx];
00136     }
00137     return score;
00138 }
00139
00140 double gkn_mm_score_cache(const gkn_mm mm, const double *v, int beg, int end) {
00141     return v[end - mm->k +1] - v[beg -1];
00142 }
00143
00144 // Length model
00145
```

```
00146 static double find_tail(double val, int x) {
00147     double lo = 0;
00148     double hi = 1000;
00149     double m;
00150 
00151     while (hi - lo > 1) {
00152         m = (hi + lo) / 2;
00153         double p = 1 / m;
00154         double f = pow(1-p, x-1) * p;
00155         if (f < val) lo += (m - lo) / 2;
00156         else         hi -= (hi - m) / 2;
00157     }
00158 
00159     return m;
00160 }
00161 
00162 void gkn_len_free(gkn_len model) {
00163     free(model->name);
00164     free(model->score);
00165     free(model);
00166 }
00167 
00168 gkn_len gkn_len_read(gkn_pipe io) {
00169     char    *line = NULL;
00170     double  *score = NULL;
00171     double   p;
00172     int      idx = 0;
00173     int      size;
00174     char     name[64];
00175 
00176     // read probabilities
00177     while ((line = gkn_readline(io)) != NULL) {
00178         if (line[0] == '%') {
00179             assert(sscanf(line, "%% LEN %s %d", name, &size) == 2);
00180             score = malloc(sizeof(double) * size);
00181             free(line);
00182         } else if (sscanf(line, "%lf", &p) == 1) {
00183             score[idx] = p;
00184             idx++;
00185             free(line);
00186         } else {
00187             free(line);
00188         }
00189     }
00190 
00191     gkn_len model = malloc(sizeof(struct gkn_LEN));
00192     model->name = malloc(strlen(name)+1);
00193     strcpy(model->name, name);
00194     model->score = score;
00195     model->size = size;
00196     model->tail = find_tail(score[size-1], size);
00197 
00198     // convert probabilities to scores
00199     double expect = (double) 1 / model->size;
00200     for (int i = 0; i < size; i++) {
00201         score[i] = log(score[i]/expect) / log(2); // divide by zero?
00202     }
00203 
00204     return model;
00205 }
00206 
00207 double gkn_len_score(const gkn_len len, int x) {
00208     assert(x >= 0);
00209     if (x >= len->size) {
00210         double p = 1 / len->tail;
00211         double q = pow(1-p, x-1) * p;
00212         double expect = (double)1 / len->size;
00213         double s = log(q/expect) / log(2);
00214         return s;
00215     } else {
00216         return len->score[x];
00217     }
00218 }
00219 
00220 #endif
```

5.2 model.h

```
00001 /*****************************************************************************\
00002  model.h
00003  Copyright (C) Ian Korf
00004 \*****************************************************************************/
00005 
```

```
00006 #ifndef GENOMIKON_MODEL_H
00007 #define GENOMIKON_MODEL_H
00008
00009 #include "sequence.h"
00010 #include "toolbox.h"
00011
00012 // Utilities
00013
00014 double gkn_p2s(double);
00015
00016 // Position Weight Matrix
00017
00018 struct gkn_PWM {
00019     char    *name;   // acceptor, donor
00020     int      size;   // eg. 6
00021     double **score;  // score[pos][nt]
00022 };
00023 typedef struct gkn_PWM * gkn_pwm;
00024 void    gkn_pwm_free(gkn_pwm);
00025 gkn_pwm gkn_pwm_read(gkn_pipe);
00026 double  gkn_pwm_score(const gkn_pwm, const char *, int);
00027
00028 // Markov model
00029
00030 struct gkn_MM {
00031     char   *name;   // exon, intron
00032     int     k;      // kmer size
00033     int     size;   // size of array
00034     double *score;  // score[dna2dec()] = value
00035 };
00036 typedef struct gkn_MM * gkn_mm;
00037 void     gkn_mm_free(gkn_mm);
00038 gkn_mm   gkn_mm_read(gkn_pipe);
00039 double   gkn_mm_score(const gkn_mm, const char *, int, int);
00040 double * gkn_mm_cache(const gkn_mm, const char *);
00041 double   gkn_mm_score_cache(const gkn_mm, const double *, int, int);
00042
00043 // Length model
00044
00045 struct gkn_LEN {
00046     char   *name;   // exon, intron, actually unused
00047     int     size;   // length of defined region
00048 //  int     limit;  // maximum length for scoring
00049     double *score;  // values for the defined region
00050     double  tail;   // mean of geometric tail
00051 };
00052 typedef struct gkn_LEN * gkn_len;
00053 void    gkn_len_free(gkn_len);
00054 gkn_len gkn_len_read(gkn_pipe/*, int*/);
00055 double  gkn_len_score(const gkn_len, int);
00056
00057 #endif
```

5.3 sequence.c

```
00001 /*****************************************************************************\
00002  sequence.c
00003  Copyright (C) Ian Korf
00004 \*****************************************************************************/
00005
00006 #ifndef GENOMIKON_SEQUENCE_C
00007 #define GENOMIKON_SEQUENCE_C
00008
00009 #include "sequence.h"
00010
00011 // Utilities
00012
00013 int gkn_ntindex(const char *seq, int off, int k) {
00014     int idx = 0;
00015     for (int i = 0; i < k; i++) {
00016         switch (seq[off+i]) {
00017             case 'A': case 'a': idx += pow(4, (k -i -1)) * 0; break;
00018             case 'C': case 'c': idx += pow(4, (k -i -1)) * 1; break;
00019             case 'G': case 'g': idx += pow(4, (k -i -1)) * 2; break;
00020             case 'T': case 't': idx += pow(4, (k -i -1)) * 3; break;
00021             default: return -1;
00022         }
00023     }
00024     return idx;
00025 }
00026
00027 char * gkn_revcomp (const char *seq) {
00028     int length = strlen(seq);
```

```
00029     char *str = gkn_malloc(length +1);
00030     str[strlen(seq)] = '\0';
00031 
00032     for (int i = 1; i <= length; i++) {
00033         switch (seq[i-1]) {
00034             case 'A': str[length -i] = 'T'; break;
00035             case 'a': str[length -i] = 't'; break;
00036             case 'C': str[length -i] = 'G'; break;
00037             case 'c': str[length -i] = 'g'; break;
00038             case 'G': str[length -i] = 'C'; break;
00039             case 'g': str[length -i] = 'c'; break;
00040             case 'T': str[length -i] = 'A'; break;
00041             case 't': str[length -i] = 'a'; break;
00042             case 'N': str[length -i] = 'N'; break;
00043             case 'n': str[length -i] = 'n'; break;
00044             case 'R': str[length -i] = 'Y'; break;
00045             case 'r': str[length -i] = 'y'; break;
00046             case 'Y': str[length -i] = 'R'; break;
00047             case 'y': str[length -i] = 'r'; break;
00048             case 'W': str[length -i] = 'S'; break;
00049             case 'w': str[length -i] = 's'; break;
00050             case 'S': str[length -i] = 'W'; break;
00051             case 's': str[length -i] = 'w'; break;
00052             case 'K': str[length -i] = 'M'; break;
00053             case 'k': str[length -i] = 'm'; break;
00054             case 'M': str[length -i] = 'K'; break;
00055             case 'm': str[length -i] = 'k'; break;
00056             case 'B': str[length -i] = 'V'; break;
00057             case 'b': str[length -i] = 'v'; break;
00058             case 'D': str[length -i] = 'H'; break;
00059             case 'd': str[length -i] = 'h'; break;
00060             case 'H': str[length -i] = 'D'; break;
00061             case 'h': str[length -i] = 'd'; break;
00062             case 'V': str[length -i] = 'B'; break;
00063             case 'v': str[length -i] = 'b'; break;
00064             default:  gkn_exit("alphabet error %c", seq[i-1]);
00065         }
00066     }
00067 
00068     return str;
00069 }
00070 
00071 // FASTA file
00072 
00073 void gkn_fasta_free(gkn_fasta ff) {
00074     free(ff->def);
00075     free(ff->seq);
00076     free(ff);
00077 }
00078 
00079 gkn_fasta gkn_fasta_new (const char *def, const char *seq) {
00080     gkn_fasta ff = gkn_malloc(sizeof(struct gkn_FASTA));
00081     ff->def = gkn_malloc(strlen(def) +1);
00082     ff->seq = gkn_malloc(strlen(seq) +1);
00083     ff->length = strlen(seq);
00084     strcpy(ff->def, def);
00085     strcpy(ff->seq, seq);
00086     return ff;
00087 }
00088 
00089 gkn_fasta gkn_fasta_read(gkn_pipe io) {
00090 
00091     // check for fasta header
00092     char c = fgetc(io->stream);
00093     ungetc(c, io->stream);
00094     if (c == EOF || (unsigned char)c == 255) return NULL;
00095     if (c != '>') gkn_exit("fasta? %c %d", c, (int)c);
00096 
00097     // def
00098     char *def = gkn_readline(io);
00099 
00100     // seq
00101     gkn_vec lines = gkn_vec_new();
00102     while (1) {
00103         char c = fgetc(io->stream);
00104         ungetc(c, io->stream);
00105         if (c == EOF || (unsigned char)c == 255 || c == '>' ) break;
00106         char *line = gkn_readline(io);
00107         if (line == NULL) break;
00108         gkn_vec_push(lines, line);
00109     }
00110 
00111     int letters = 0;
00112     for (int i = 0; i < lines->size; i++) {
00113         char *line = lines->elem[i];
00114         letters += strlen(line);
00115     }
```

```
00116
00117     char *seq = malloc(letters + 1);
00118     int off = 0;
00119     for (int i = 0; i < lines->size; i++) {
00120         char *line = lines->elem[i];
00121         strcpy(seq + off, line);
00122         off += strlen(line);
00123     }
00124
00125     // clean up
00126     for (int i = 0; i < lines->size; i++) {
00127         free(lines->elem[i]);
00128     }
00129     gkn_vec_free(lines);
00130
00131     // return object
00132     gkn_fasta ff = malloc(sizeof(struct gkn_FASTA));
00133     ff->def = def;
00134     ff->seq = seq;
00135     ff->length = strlen(seq);
00136
00137     return ff;
00138 }
00139
00140 static int FASTA_LINE_LENGTH = 80;
00141
00142 void gkn_fasta_write(FILE *stream, const gkn_fasta ff) {
00143     if (ff->def[0] != '>') fprintf(stream, ">");
00144     fprintf(stream, "%s", ff->def);
00145     if (ff->def[strlen(ff->def) -1] != '\n') fprintf(stream, "\n");
00146
00147     for (int i = 0; i < ff->length; i++) {
00148         fputc(ff->seq[i], stream);
00149         if ((i+1) % FASTA_LINE_LENGTH == 0) fprintf(stream, "\n");
00150     }
00151
00152     fprintf(stream, "\n");
00153 }
00154
00155 void gkn_fasta_set_line_length (int length) {
00156     FASTA_LINE_LENGTH = length;
00157 }
00158
00159 #endif
```

5.4 sequence.h

```
00001 /*****************************************************************************\
00002  sequence.h
00003  Copyright (C) Ian Korf
00004 \*****************************************************************************/
00005
00006 #ifndef GENOMIKON_SEQUENCE_H
00007 #define GENOMIKON_SEQUENCE_H
00008
00009 #include "model.h"
00010 #include "toolbox.h"
00011
00012 // Utilities
00013 int    gkn_ntindex(const char *, int, int);
00014 char * gkn_revcomp(const char*);
00015
00017 struct gkn_FASTA {
00018     int    length;
00019     char * def;
00020     char * seq;
00021 };
00022 typedef struct gkn_FASTA * gkn_fasta;
00023
00025 void      gkn_fasta_free(gkn_fasta);
00026 gkn_fasta gkn_fasta_new(const char *, const char *);
00028 gkn_fasta gkn_fasta_read(gkn_pipe);
00029 void      gkn_fasta_write(FILE *, const gkn_fasta);
00030 void      gkn_fasta_set_line_length(int);
00031
00032 #endif
```

5.5 toolbox.c

```
00001 /*****************************************************************************\
```

```
00002  toolbox.c
00003  Copyright (C) Ian Korf
00004 \*****************************************************************************/
00005
00006 #ifndef GENOMIKON_TOOLBOX_C
00007 #define GENOMIKON_TOOLBOX_C
00008
00009 #include "toolbox.h"
00010
00011 static char gkn_version_number[] = "genomikon-2021";
00012 static char gkn_program_name[256] = "name not set";
00013
00014 char * gkn_get_version_number (void) {return gkn_version_number;}
00015 void   gkn_set_program_name (const char *s) {strcpy(gkn_program_name, s);}
00016 char * gkn_get_program_name (void) {return gkn_program_name;}
00017
00018 void * gkn_malloc(size_t size) {
00019     void *mem = malloc(size);
00020     if (mem == NULL) gkn_exit("gkn_malloc %d", size);
00021     return mem;
00022 }
00023
00024 void * gkn_calloc(size_t count, size_t size) {
00025     void *mem = calloc(count, size);
00026     if (mem == NULL) gkn_exit("gkn_calloc %d %d", count, size);
00027     return mem;
00028 }
00029
00030 void * gkn_realloc(void *p, size_t size) {
00031     void *mem = realloc(p, size);
00032     if (mem == NULL) gkn_exit("gkn_realloc %d", size);
00033     return mem;
00034 }
00035
00036 void gkn_ivec_free(gkn_ivec vec) {
00037     if (vec == NULL) return;
00038     if (vec->elem) free(vec->elem);
00039     free(vec);
00040 }
00041
00042 gkn_ivec gkn_ivec_new(void) {
00043     gkn_ivec vec = gkn_malloc(sizeof(struct gkn_IVEC));
00044     vec->size  = 0;
00045     vec->limit = 0;
00046     vec->elem  = NULL;
00047     return vec;
00048 }
00049
00050 void gkn_ivec_push(gkn_ivec vec, int val) {
00051     if (vec->limit == vec->size) {
00052         if (vec->limit == 0) vec->limit  = 1;
00053         else                 vec->limit *= 2;
00054         vec->elem = gkn_realloc(vec->elem, vec->limit * sizeof(int));
00055     }
00056     vec->elem[vec->size] = val;
00057     vec->last = val;
00058     vec->size++;
00059 }
00060
00061 int gkn_ivec_pop(gkn_ivec vec) {
00062     if (vec->size == 0) gkn_exit("can't pop a zero-length vector");
00063     vec->size--;
00064     return vec->elem[vec->size];
00065 }
00066
00067 void gkn_fvec_free(gkn_fvec vec) {
00068     if (vec == NULL) return;
00069     if (vec->elem) free(vec->elem);
00070     free(vec);
00071 }
00072
00073 gkn_fvec gkn_fvec_new(void) {
00074     gkn_fvec vec = gkn_malloc(sizeof(struct gkn_FVEC));
00075     vec->size  = 0;
00076     vec->limit = 0;
00077     vec->elem  = NULL;
00078     return vec;
00079 }
00080
00081 void gkn_fvec_push(gkn_fvec vec, double val) {
00082     if (vec->limit == vec->size) {
00083         if (vec->limit == 0) vec->limit  = 1;
00084         else                 vec->limit *= 2;
00085         vec->elem = gkn_realloc(vec->elem, vec->limit * sizeof(double));
00086     }
00087     vec->elem[vec->size] = val;
00088     vec->last = val;
```

```
00089     vec->size++;
00090 }
00091
00092 double gkn_fvec_pop(gkn_fvec vec) {
00093     if (vec->size == 0) gkn_exit("can't pop a zero-length vector");
00094     vec->size--;
00095     return vec->elem[vec->size];
00096 }
00097
00098 void gkn_tvec_free(gkn_tvec vec) {
00099     if (vec == NULL) return;
00100     if (vec->elem) {
00101         for (int i = 0; i < vec->size; i++) free(vec->elem[i]);
00102         free(vec->elem);
00103     }
00104     free(vec);
00105 }
00106
00107 gkn_tvec gkn_tvec_new(void) {
00108     gkn_tvec vec = gkn_malloc(sizeof(struct gkn_TVEC));
00109     vec->size  = 0;
00110     vec->limit = 0;
00111     vec->elem  = NULL;
00112     return vec;
00113 }
00114
00115 void gkn_tvec_push(gkn_tvec vec, const char *text) {
00116     if (vec->limit == vec->size) {
00117         if (vec->limit == 0) vec->limit  = 1;
00118         else                 vec->limit *= 2;
00119         vec->elem = gkn_realloc(vec->elem, vec->limit * sizeof(char *));
00120     }
00121     vec->elem[vec->size] = gkn_malloc(strlen(text) + 1);
00122     strcpy(vec->elem[vec->size], text);
00123     vec->last = vec->elem[vec->size];
00124     vec->size++;
00125 }
00126
00127 char * gkn_tvec_pop(gkn_tvec vec) {
00128     if (vec->size == 0) gkn_exit("can't pop a zero-length vector");
00129     vec->size--;
00130     return vec->elem[vec->size];
00131 }
00132
00133 void gkn_vec_free(gkn_vec vec) {
00134     if (vec == NULL) return;
00135     if (vec->elem) free(vec->elem);
00136     free(vec);
00137 }
00138
00139 gkn_vec gkn_vec_new(void) {
00140     gkn_vec vec = gkn_malloc(sizeof(struct gkn_VEC));
00141     vec->size  = 0;
00142     vec->limit = 0;
00143     vec->elem  = NULL;
00144     return vec;
00145 }
00146
00147 void gkn_vec_push(gkn_vec vec, void *p) {
00148     if (vec->limit == vec->size) {
00149         if (vec->limit == 0) vec->limit  = 1;
00150         else                 vec->limit *= 2;
00151         vec->elem = gkn_realloc(vec->elem, vec->limit * sizeof(void *));
00152     }
00153     vec->elem[vec->size] = p;
00154     vec->last = vec->elem[vec->size];
00155     vec->size++;
00156 }
00157
00158 void * gkn_vec_pop(gkn_vec vec) {
00159     if (vec->size == 0) gkn_exit("can't pop a zero-length vector");
00160     vec->size--;
00161     return vec->elem[vec->size];
00162 }
00163
00164 // hashing materials
00165 static double HASH_MULTIPLIER[7] = {
00166     3.1415926536, // PI
00167     2.7182818285, // e
00168     1.6180339887, // golden mean
00169     1.7320508076, // square root of 3
00170     2.2360679775, // square root of 5
00171     2.6457513111, // square root of 7
00172     3.3166247904  // square root of 11
00173 };
00174 static double MAX_HASH_DEPTH = 2.0;
00175 static int HashLevelToSlots(int level) {return pow(4, level);}
```

```
00176 static int HashFunc(const gkn_map hash, const char *key) {
00177     double sum = 0;
00178     for (int i = 0; i < strlen(key); i++)
00179         sum += key[i] * HASH_MULTIPLIER[i % 7];
00180     return (int) (hash->slots * (sum - floor(sum)));
00181 }
00182
00183 static void ExpandHash(gkn_map hash) {
00184     int      oldslots = hash->slots;
00185     gkn_vec *oldkey = hash->key;
00186     gkn_vec *oldval = hash->val;
00187     gkn_vec   kvec;
00188     gkn_vec   vvec;
00189     gkn_tvec keys;
00190
00191     // create the new hash
00192     hash->level = hash->level +1;
00193     hash->slots = HashLevelToSlots(hash->level);
00194     hash->key   = gkn_malloc(hash->slots * sizeof(struct gkn_VEC));
00195     hash->val   = gkn_malloc(hash->slots * sizeof(struct gkn_VEC));
00196     for (int i = 0; i < hash->slots; i++) {
00197         hash->key[i] = gkn_vec_new();
00198         hash->val[i] = gkn_vec_new();
00199     }
00200
00201     // brand new hash?
00202     if (hash->keys->size == 0) return;
00203
00204     keys = hash->keys;
00205     hash->keys = gkn_tvec_new();
00206
00207     // transfer old stuff to new hash
00208     for (int i = 0; i < oldslots; i++) {
00209         kvec = oldkey[i];
00210         vvec = oldval[i];
00211         for (int j = 0; j < kvec->size; j++) {
00212             char *key = kvec->elem[j];
00213             char *val = vvec->elem[j];
00214             gkn_map_set(hash, key, val);
00215         }
00216     }
00217
00218     // free old stuff
00219     for (int i = 0; i < oldslots; i++) {
00220         kvec = oldkey[i];
00221         vvec = oldval[i];
00222         gkn_vec_free(kvec);
00223         gkn_vec_free(vvec);
00224     }
00225     free(oldkey);
00226     free(oldval);
00227     gkn_tvec_free(keys);
00228 }
00229
00230 void gkn_map_free(gkn_map hash) {
00231     if (hash == NULL) return;
00232     for (int i = 0; i < hash->slots; i++) {
00233         if (hash->key[i]) gkn_vec_free(hash->key[i]);
00234         if (hash->val[i]) gkn_vec_free(hash->val[i]);
00235     }
00236     gkn_tvec_free(hash->keys);
00237     gkn_vec_free(hash->vals);
00238     free(hash->key);
00239     free(hash->val);
00240     free(hash);
00241 }
00242
00243 gkn_map gkn_map_new(void) {
00244     gkn_map hash = gkn_malloc(sizeof(struct gkn_MAP));
00245     hash->level = 0;
00246     hash->slots = 0;
00247     hash->keys  = gkn_tvec_new();
00248     hash->vals  = gkn_vec_new();
00249     hash->key   = NULL;
00250     hash->val   = NULL;
00251     ExpandHash(hash);
00252     return hash;
00253 }
00254
00255 void * gkn_map_get(const gkn_map hash, const char *key) {
00256     int index = HashFunc(hash, key);
00257
00258     // resolve collisions
00259     for (int i = 0; i < hash->key[index]->size; i++) {
00260         char *string = hash->key[index]->elem[i];
00261         if (strcmp(key, string) == 0) {
00262             return hash->val[index]->elem[i];
```

```
00263         }
00264     }
00265     return NULL; // return is NULL if not found
00266 }
00267
00268 void gkn_map_set(gkn_map hash, const char *key, void *val) {
00269     int new_key = 1;
00270     int index = HashFunc(hash, key);
00271
00272     // reassign unless new key
00273     for (int i = 0; i < hash->key[index]->size; i++) {
00274         char *string = hash->key[index]->elem[i];
00275         if (strcmp(key, string) == 0) {
00276             hash->val[index]->elem[i] = val;
00277             new_key = 0;
00278             return;
00279         }
00280     }
00281
00282     if (new_key) {
00283         gkn_tvec_push(hash->keys, key);
00284         gkn_vec_push(hash->key[index], hash->keys->last);
00285         gkn_vec_push(hash->vals, val);
00286         gkn_vec_push(hash->val[index], hash->vals->last);
00287     }
00288
00289     // check if we have to expand the hash
00290     if ((double)hash->keys->size / (double)hash->slots >= MAX_HASH_DEPTH) {
00291         ExpandHash(hash);
00292     }
00293 }
00294
00295 gkn_tvec gkn_map_keys(const gkn_map hash) {
00296     gkn_tvec vec = gkn_tvec_new();
00297     for (int i = 0; i < hash->keys->size; i++) gkn_tvec_push(vec, hash->keys->elem[i]);
00298     return vec;
00299 }
00300
00301 gkn_vec gkn_map_vals(const gkn_map hash) {
00302     gkn_vec vec = gkn_vec_new();
00303     for (int i = 0; i < hash->vals->size; i++) gkn_vec_push(vec, hash->vals->elem[i]);
00304     return vec;
00305 }
00306
00307 void gkn_map_stat(const gkn_map hash) {
00308     int max = 0;
00309     int min = INT_MAX;
00310     int total = 0;
00311     for (int i = 0; i < hash->slots; i++) {
00312         int count = hash->val[i]->size;
00313         total += count;
00314         if (count > max) max = count;
00315         if (count < min) min = count;
00316     }
00317     fprintf(stdout, "HashStats: level=%d slots=%d keys=%d min=%d max=%d ave=%f\n",
00318          hash->level, hash->slots, hash->keys->size, min, max,
00319          (double)total / (double)hash->slots);
00320 }
00321
00322 // text map
00323
00324 void gkn_tmap_free(gkn_tmap t) {
00325     gkn_map_free(t->hash);
00326     gkn_tvec_free(t->tvec);
00327     free(t);
00328 }
00329
00330 gkn_tmap gkn_tmap_new(void) {
00331     gkn_tmap t = gkn_malloc(sizeof(struct gkn_TMAP));
00332     t->hash = gkn_map_new();
00333     t->tvec = gkn_tvec_new();
00334     return t;
00335 }
00336
00337 void gkn_tmap_set(gkn_tmap t, const char *key, const char *val) {
00338     gkn_tvec_push(t->tvec, val);
00339     gkn_map_set(t->hash, key, t->tvec->last);
00340 }
00341
00342 int gkn_tmap_exists(const gkn_tmap t, const char *key) {
00343     void *ref = gkn_map_get(t->hash, key);
00344     if (ref == NULL) return 0;
00345     return 1;
00346 }
00347
00348 char * gkn_tmap_get(const gkn_tmap t, const char *key) {
00349     void *ref = gkn_map_get(t->hash, key);
```

```
00350     assert(ref != NULL);
00351     return ref;
00352 }
00353
00354 gkn_tvec gkn_tmap_keys(const gkn_tmap t) {
00355     return gkn_map_keys(t->hash);
00356 }
00357
00358 // suffix tree
00359
00360 #define MAX_WORD_LENGTH 65536
00361
00362 void gkn_xnode_free(gkn_xnode xn) {
00363     gkn_vec_free(xn->children);
00364     free(xn->data);
00365     free(xn);
00366 }
00367
00368 gkn_xnode gkn_xnode_new (char c) {
00369     gkn_xnode xn = gkn_malloc(sizeof(struct gkn_XNODE));
00370     xn->children = gkn_vec_new();
00371     xn->data     = NULL;
00372     xn->c        = c;
00373     return xn;
00374 }
00375
00376 gkn_xnode gkn_xnode_search (const gkn_xnode xn, char c) {
00377     for (int i = 0; i < xn->children->size; i++) {
00378         gkn_xnode child = xn->children->elem[i];
00379         if (child->c == c) return child;
00380     }
00381     return NULL;
00382 }
00383
00384 void gkn_xtree_free (gkn_xtree xt) {
00385     for (int i = 0; i < xt->alloc->size; i++) {
00386         gkn_xnode node = xt->alloc->elem[i];
00387         gkn_xnode_free(node);
00388     }
00389     gkn_vec_free(xt->alloc);
00390     if (xt->head) gkn_xnode_free(xt->head);
00391     free(xt);
00392 }
00393
00394 gkn_xtree gkn_xtree_new (void) {
00395     gkn_xtree xt = gkn_malloc(sizeof(struct gkn_xtree));
00396     xt->head  = gkn_xnode_new(0);
00397     xt->alloc = gkn_vec_new();
00398     return xt;
00399 }
00400
00401 void gkn_xtree_set (gkn_xtree xt, const char *string, void *value) {
00402     int len = strlen(string);
00403     if (len < 1) gkn_exit("gkn_xtree_set with empty string");
00404     if (len >= MAX_WORD_LENGTH)
00405         gkn_exit("gkn_xtree word length exceeded (%d)\n", MAX_WORD_LENGTH);
00406
00407     gkn_xnode parent = xt->head;
00408     for (int i = 0; i < len; i++) {
00409         char c = string[i];
00410         gkn_xnode child = gkn_xnode_search(parent, c);
00411         if (child == NULL) {
00412             child = gkn_xnode_new(c);
00413             gkn_vec_push(parent->children, child);
00414             gkn_vec_push(xt->alloc, child);
00415         }
00416         parent = child;
00417     }
00418
00419     parent->data = value;
00420 }
00421
00422 void * gkn_xtree_get (const gkn_xtree xt, const char *string) {
00423     int len = strlen(string);
00424     if (len < 1) gkn_exit("gkn_xtree_get with empty string");
00425
00426     gkn_xnode parent = xt->head;
00427     for (int i = 0; i < len; i++) {
00428         char c = string[i];
00429         gkn_xnode child = gkn_xnode_search(parent, c);
00430         if (child == NULL) return NULL;
00431         parent = child;
00432     }
00433     return parent->data;
00434 }
00435
00436 int gkn_xtree_check (const gkn_xtree xt, const char *string) {
```

```
00437     int len = strlen(string);
00438     if (len < 1) gkn_exit("gkn_xtree_check with empty string");
00439
00440     gkn_xnode parent = xt->head;
00441     for (int i = 0; i < len; i++) {
00442         char c = string[i];
00443         gkn_xnode child = gkn_xnode_search(parent, c);
00444         if (child == NULL) return 0;
00445         parent = child;
00446     }
00447     return 1;
00448 }
00449
00450 gkn_xnode gkn_xtree_node (const gkn_xtree xt, const char *string) {
00451     int len = strlen(string);
00452     if (len < 1) gkn_exit("gkn_xtree_node with empty string");
00453
00454     gkn_xnode parent = xt->head;
00455     for (int i = 0; i < len; i++) {
00456         char c = string[i];
00457         gkn_xnode child = gkn_xnode_search(parent, c);
00458         if (child == NULL) return 0;
00459         parent = child;
00460     }
00461     return parent;
00462 }
00463
00464 static void xtree_add_keys (const gkn_xnode parent, gkn_tvec keys, char *key,
00465     int length)
00466 {
00467     if (parent->data) gkn_tvec_push(keys, key);
00468
00469     for (int i = 0; i < parent->children->size; i++) {
00470         gkn_xnode child = parent->children->elem[i];
00471         key[length] = child->c;
00472         key[length+1] = '\0';
00473         xtree_add_keys(parent->children->elem[i], keys, key, length+1);
00474     }
00475 }
00476
00477 gkn_tvec gkn_xtree_keys (const gkn_xtree xt) {
00478     char     key[MAX_WORD_LENGTH];
00479     gkn_tvec keys = gkn_tvec_new();
00480
00481     for (int i = 0; i < xt->head->children->size; i++) {
00482         gkn_xnode parent = xt->head->children->elem[i];
00483         key[0] = parent->c;
00484         key[1] = '\0';
00485         xtree_add_keys(parent, keys, key, 1);
00486     }
00487
00488     return keys;
00489 }
00490
00491
00492 // command line options
00493
00494 static gkn_tvec COMMAND_LINE = NULL;
00495 static gkn_map CL_REGISTER   = NULL;
00496 static gkn_map CL_OPTIONS    = NULL;
00497
00498 void gkn_register_option(const char *name, int flag) {
00499     if (COMMAND_LINE == NULL) {
00500         COMMAND_LINE = gkn_tvec_new();
00501         CL_REGISTER  = gkn_map_new();
00502         CL_OPTIONS   = gkn_map_new();
00503     }
00504
00505     switch (flag) {
00506         case 0: gkn_map_set(CL_REGISTER, name, (void *)1); break;
00507         case 1: gkn_map_set(CL_REGISTER, name, (void *)2); break;
00508         default: gkn_exit("gkn_register_option: flag 0 or 1");
00509     }
00510 }
00511
00512 void gkn_parse_options(int *argc, char **argv) {
00513     for (int i = 0; i < *argc; i++) {
00514         char *token = argv[i];
00515         if (token[0] == '-' && strlen(token) > 1) {
00516             switch ((size_t)gkn_map_get(CL_REGISTER, token)) {
00517                 case 0:
00518                     gkn_exit("unknown option (%s)", token);
00519                     break;
00520                 case 1:
00521                     gkn_map_set(CL_OPTIONS, token, token);
00522                     break;
00523                 case 2:
```

```
00524                     gkn_map_set(CL_OPTIONS, token, argv[i+1]);
00525                     i++;
00526                     break;
00527                 default:
00528                     gkn_exit("not possible");
00529             }
00530         } else {
00531             gkn_tvec_push(COMMAND_LINE, argv[i]);
00532         }
00533     }
00534
00535     *argc = COMMAND_LINE->size;
00536     for (int i = 0; i < COMMAND_LINE->size; i++) {
00537         argv[i] = COMMAND_LINE->elem[i];
00538     }
00539 }
00540
00541 char * gkn_option(const char *tag) {
00542     return gkn_map_get(CL_OPTIONS, tag);
00543 }
00544
00545 // pipe
00546
00547 void gkn_pipe_close(gkn_pipe pipe) {
00548     pipe->mode = 0;
00549     free(pipe->name);
00550     if (pipe->gzip) pclose(pipe->stream);
00551     else            fclose(pipe->stream);
00552     pipe->gzip = 0;
00553     free(pipe);
00554 }
00555
00556 gkn_pipe gkn_pipe_open(const char *name, const char *mode) {
00557     char    command[1024];
00558     int     length = strlen(name);
00559     gkn_pipe pipe = gkn_malloc(sizeof(struct gkn_PIPE));
00560
00561     if      (strcmp(mode, "r") == 0)  pipe->mode = 0;
00562     else if (strcmp(mode, "w") == 0)  pipe->mode = 1;
00563     else if (strcmp(mode, "r+") == 0) pipe->mode = 2;
00564     else gkn_exit("r, w, or r+ only in gkn_pipe");
00565
00566     pipe->name = gkn_malloc(length + 1);
00567     strcpy(pipe->name, name);
00568
00569     pipe->gzip = 0;
00570
00571     if (name[length -3] == '.' &&
00572         name[length -2] == 'g' &&
00573         name[length -1] == 'z') pipe->gzip = 1; // .gz
00574     if (name[length -2] == '.' &&
00575         name[length -1] == 'z') pipe->gzip = 1; // .z
00576     if (name[length -2] == '.' &&
00577         name[length -1] == 'Z') pipe->gzip = 1; // .Z
00578
00579     if (pipe->gzip) {
00580         if (pipe->mode != 0) gkn_exit("compressed pipes are read only");
00581         sprintf(command, "gunzip -c %s", name);
00582         pipe->stream = popen(command, "r");
00583     } else {
00584         if (strcmp(name, "-") == 0) pipe->stream = stdin;
00585         else                        pipe->stream = fopen(name, mode);
00586     }
00587
00588     if (pipe->stream == NULL) {
00589         gkn_exit("failed to open %s\n", name);
00590     }
00591
00592     return pipe;
00593 }
00594
00595 char * gkn_readline(gkn_pipe io) {
00596     char line[4096];
00597     int  read = 0;
00598     while (fgets(line, sizeof(line), io->stream) != NULL) {
00599         if (line[0] == '#') continue; // skipping comments
00600         if (strlen(line) == 0) continue;
00601         read = 1;
00602         break;
00603     }
00604     if (read == 0) return NULL;
00605
00606     // stripping newline
00607     if (line[strlen(line) -1] == '\n') line[strlen(line) -1] = '\0';
00608     char *out = malloc(strlen(line) + 1);
00609     strcpy(out, line);
00610
```

```
00611     return out;
00612 }
00613
00614 void gkn_exit(const char* format, ...) {
00615     va_list args;
00616     fflush(stdout);
00617     fprintf(stderr, "ERROR from program %s, libarary %s\n",
00618         gkn_get_program_name(),
00619         gkn_get_version_number());
00620     va_start(args, format);
00621     vfprintf(stderr, format, args);
00622     va_end(args);
00623     fprintf(stderr, "\n");
00624     exit(1);
00625 }
00626
00627 #endif
```

5.6 toolbox.h

```
00001 /*****************************************************************************\
00002  toolbox.h
00003  Copyright (C) Ian Korf
00004 \*****************************************************************************/
00005
00006 #ifndef GENOMIKON_TOOLBOX_H
00007 #define GENOMIKON_TOOLBOX_H
00008
00009 #include <assert.h>
00010 #include <ctype.h>
00011 #include <errno.h>
00012 #include <float.h>
00013 #include <limits.h>
00014 #include <math.h>
00015 #include <stdarg.h>
00016 #include <stdlib.h>
00017 #include <stdio.h>
00018 #include <string.h>
00019
00020 // library and program info
00021 char * gkn_get_version_number(void);
00022 void   gkn_set_program_name(const char *);
00023 char * gkn_get_program_name(void);
00024
00025 // memory
00026 void * gkn_malloc(size_t);
00027 void * gkn_calloc(size_t, size_t);
00028 void * gkn_realloc(void *, size_t);
00029
00030 // integer vector
00031 struct gkn_IVEC {
00032     int * elem;
00033     int   size;
00034     int   limit;
00035     int   last;
00036 };
00037 typedef struct gkn_IVEC * gkn_ivec;
00038 void     gkn_ivec_free(gkn_ivec);
00039 gkn_ivec gkn_ivec_new(void);
00040 void     gkn_ivec_push(gkn_ivec, int);
00041 int      gkn_ivec_pop(gkn_ivec);
00042
00043 // float vector
00044 struct gkn_FVEC {
00045     double * elem;
00046     int      size;
00047     int      limit;
00048     int      last;
00049 };
00050 typedef struct gkn_FVEC * gkn_fvec;
00051 void     gkn_fvec_free(gkn_fvec);
00052 gkn_fvec gkn_fvec_new(void);
00053 void     gkn_fvec_push(gkn_fvec, double);
00054 double   gkn_fvec_pop(gkn_fvec);
00055
00056 // text vector
00057 struct gkn_TVEC {
00058     char ** elem;
00059     int     size;
00060     int     limit;
00061     char  * last;
00062 };
00063 typedef struct gkn_TVEC * gkn_tvec;
```

```
00064 void     gkn_tvec_free(gkn_tvec);
00065 gkn_tvec gkn_tvec_new(void);
00066 void     gkn_tvec_push(gkn_tvec, const char *);
00067 char *   gkn_tvec_pop(gkn_tvec);
00068
00069 // generic void * vector
00070 struct gkn_VEC {
00071     void ** elem;
00072     int     size;
00073     int     limit;
00074     void  * last;
00075 };
00076 typedef struct gkn_VEC * gkn_vec;
00077 void    gkn_vec_free(gkn_vec);
00078 gkn_vec gkn_vec_new(void);
00079 void    gkn_vec_push(gkn_vec, void *);
00080 void *  gkn_vec_pop(gkn_vec);
00081
00082 // generic map (text key, void * value)
00083 struct gkn_MAP {
00084     int      level;
00085     int      slots;
00086     gkn_tvec keys;
00087     gkn_vec   vals;
00088     gkn_vec * key;
00089     gkn_vec * val;
00090 };
00091 typedef struct gkn_MAP * gkn_map;
00092 void     gkn_map_free(gkn_map);
00093 gkn_map   gkn_map_new(void);
00094 void     gkn_map_set(gkn_map, const char *, void *);
00095 void *   gkn_map_get(const gkn_map, const char *);
00096 gkn_tvec gkn_map_keys(const gkn_map);
00097 gkn_vec   gkn_map_vals(const gkn_map);
00098 void     gkn_map_stat(const gkn_map);
00099
00100 // text map
00101 struct gkn_TMAP {
00102     gkn_map  hash;
00103     gkn_tvec tvec;
00104 };
00105 typedef struct gkn_TMAP * gkn_tmap;
00106 void     gkn_tmap_free(gkn_tmap);
00107 gkn_tmap gkn_tmap_new(void);
00108 void     gkn_tmap_set(gkn_tmap, const char *, const char *);
00109 char *   gkn_tmap_get(const gkn_tmap, const char *);
00110 int      gkn_tmap_exists(const gkn_tmap, const char *);
00111 gkn_tvec gkn_tmap_keys(const gkn_tmap);
00112
00113 /* generic suffix tree */
00114 struct gkn_XNODE {
00115     gkn_vec children;
00116     void  *data;
00117     char   c;
00118 };
00119 typedef struct gkn_XNODE * gkn_xnode;
00120 void      gkn_xnode_free(gkn_xnode);
00121 gkn_xnode gkn_xnode_new(char);
00122 gkn_xnode gkn_xnode_search(const gkn_xnode, char c);
00123
00124 struct gkn_xtree {
00125     gkn_xnode head;
00126     gkn_vec   alloc;
00127 };
00128 typedef struct gkn_xtree * gkn_xtree;
00129 void      gkn_xtree_free(gkn_xtree);
00130 gkn_xtree gkn_xtree_new(void);
00131 void *    gkn_xtree_get(const gkn_xtree, const char *);
00132 int       gkn_xtree_check(const gkn_xtree, const char *);
00133 gkn_xnode gkn_xtree_node(const gkn_xtree, const char *);
00134 void      gkn_xtree_set(gkn_xtree, const char *, void *);
00135 gkn_tvec  gkn_xtree_keys(const gkn_xtree);
00136
00137 // command line processing
00138 void   gkn_register_option(const char *, int);
00139 void   gkn_parse_options(int *, char **);
00140 char * gkn_option(const char *);
00141
00142 // pipe
00143 struct gkn_PIPE {
00144     int    mode; // 0 = read, 1 = write, 2 = r+
00145     char * name;
00146     int    gzip;
00147     FILE * stream;
00148 };
00149 typedef struct gkn_PIPE * gkn_pipe;
00150 gkn_pipe gkn_pipe_open(const char *, const char *);
```

```
00151 void     gkn_pipe_close(gkn_pipe);
00152
00153 // input/output
00154 char * gkn_readline(gkn_pipe);
00155 void   gkn_exit(const char *, ...);
00156
00157 #endif
```

Index

alloc
    gkn_xtree, 20

c
    gkn_XNODE, 19
children
    gkn_XNODE, 20

data
    gkn_XNODE, 20
def
    gkn_FASTA, 7

elem
    gkn_FVEC, 8
    gkn_IVEC, 9
    gkn_TVEC, 17
    gkn_VEC, 18

gkn_FASTA, 7
    def, 7
    length, 7
    seq, 8
gkn_FVEC, 8
    elem, 8
    last, 8
    limit, 9
    size, 9
gkn_IVEC, 9
    elem, 9
    last, 10
    limit, 10
    size, 10
gkn_LEN, 10
    name, 11
    score, 11
    size, 11
    tail, 11
gkn_MAP, 11
    key, 12
    keys, 12
    level, 12
    slots, 12
    val, 12
    vals, 12
gkn_MM, 13
    k, 13
    name, 13
    score, 13
    size, 14
gkn_PIPE, 14
    gzip, 14
    mode, 14
    name, 15
    stream, 15
gkn_PWM, 15
    name, 15
    score, 16
    size, 16
gkn_TMAP, 16
    hash, 16
    tvec, 16
gkn_TVEC, 17
    elem, 17
    last, 17
    limit, 17
    size, 18
gkn_VEC, 18
    elem, 18
    last, 18
    limit, 19
    size, 19
gkn_XNODE, 19
    c, 19
    children, 20
    data, 20
gkn_xtree, 20
    alloc, 20
    head, 20
gzip
    gkn_PIPE, 14

hash
    gkn_TMAP, 16
head
    gkn_xtree, 20

k
    gkn_MM, 13
key
    gkn_MAP, 12
keys
    gkn_MAP, 12

last
    gkn_FVEC, 8
    gkn_IVEC, 10
    gkn_TVEC, 17
    gkn_VEC, 18
length

gkn_FASTA, 7
level
gkn_MAP, 12
limit
gkn_FVEC, 9
gkn_IVEC, 10
gkn_TVEC, 17
gkn_VEC, 19

mode
gkn_PIPE, 14

name
gkn_LEN, 11
gkn_MM, 13
gkn_PIPE, 15
gkn_PWM, 15

score
gkn_LEN, 11
gkn_MM, 13
gkn_PWM, 16
seq
gkn_FASTA, 8
size
gkn_FVEC, 9
gkn_IVEC, 10
gkn_LEN, 11
gkn_MM, 14
gkn_PWM, 16
gkn_TVEC, 18
gkn_VEC, 19
slots
gkn_MAP, 12
stream
gkn_PIPE, 15

tail
gkn_LEN, 11
tvec
gkn_TMAP, 16

val
gkn_MAP, 12
vals
gkn_MAP, 12