BLAST Scoring Parameters

E. Michael Gertz

March 16, 2005

Contents

1 Introduction

BLAST is a tool that is used to align biological sequences. In its most basic form, BLAST finds local alignments of a *query* amino acid or nucleotide sequence to another amino acid or nucleotide sequence, known as a *subject* sequence. Most commonly, a single BLAST run finds local alignments of a query sequence to a database of subject sequences.

An alignment may or may not contain gaps, depending on the type of BLAST search that is performed. A search for an alignment that may contain gaps is known as a *gapped search.* An optimal alignment that results from a gapped search may happen to not involve any gaps. Nonetheless, the result of a gapped search is known as a *gapped alignment.* Similarly, an *ungapped search* finds alignments that must not contain gaps, and such alignments are known as *ungapped alignments.* The terms gapped and ungapped are used frequently in describing the operation of BLAST, and the meaning should be clear from context.

Every alignment computed by BLAST has a *similarity score* associated with it. A BLAST search uses a matrix that associates every pair of matched or mismatched characters in an alignment with a score. For an ungapped alignment, the scoring matrix suffices to determine a score. For alignments that contain gaps, three additional parameters are required:

gap_open the penalty for opening a gap;

gap_extend the penalty for extending an open gap by a single amino acid or nucleotide; and

decline_align the penalty for declining to align two characters in a gap region.

The `decline_align` parameter is currently always set to an effectively infinite value, disabling its use. The purpose of this parameter is discussed in Altschul [3].

Karlin-Altschul parameters are used by BLAST to evaluate the significance of high-scoring alignments. Some BLAST programs evaluate the significance of each distinct alignment separately, but others evaluate the significance of several distinct alignments taken together as a linked set. The expected value (E-value) of a single distinct alignment may be calculated by the formula

$$E = Kmne^{-\lambda S}, \tag{1}$$

where K and λ are Karlin-Altschul parameters and m and n are the effective lengths (defined later) of the query sequence and database, respectively. Karlin and Altschul [10] and Dembo, Karlin and Zeitouni [8] discuss the motivation for formula (1). Separate sets of Karlin-Altschul parameters are used to evaluate the significance of gapped and ungapped alignments. We defer discussion of evaluating the significance of multiple distinct alignments to section 7.

An alignment is less likely to start near the right edge of a sequence than it is to start away from that edge. To compensate for this effect, equation (1) uses the *effective lengths* of the query and database sequences, rather than their actual lengths. Effective lengths, and how they are calculated, are discussed in section 5 of this document, in Altschul and Gish [5] and in Altschul et al. [4].

The values K and λ are also used to compute normalized scores, which are used to compare the scores of alignments computed using different scoring systems. One common form of normalized score is the *bit score*, calculated by the formula

$$S_B = (\lambda S - \ln K)/\ln 2. \tag{2}$$

Less common, but also used, is the *nat score*, computed by the formula

$$S_N = \lambda S - \ln K.$$

For ungapped alignments, Karlin-Altschul parameters may be explicitly calculated. For gapped alignments, they must be obtained by simulation using random sequences of "standard" composition and a specific choice of matrix and of `gap_open`, `gap_extend` and `decline_align` parameters. For this reason, BLAST calculates ungapped Karlin-Altschul parameters based on the composition of two sequences, but obtains gapped Karlin-Altschul parameters from a set of precomputed tables. Routines that adjust scoring systems for gapped alignments to account for the composition of the query and subject sequences have been developed (see section 9.)

An additional statistical parameter that is computed and saved is H, which is known as the entropy of the scoring system. For ungapped alignments,

$$H = \lambda \sum_{i=\ell}^{u} i P_1(i) e^{i\lambda},$$

where $P_1(i)$ is the probability that score i occurs when one character of the subject sequence is aligned with one character of the query, and

$$\ell = \min\{i \mid P_1(i) > 0\} \text{ and } u = \max\{i \mid P_1(i) > 0\}.$$

The value H has theoretical implications but is rarely used in the BLAST code. The value is used by the `BlastKarlinLHtoK` routine to compute K and by the `BLAST_CalcEffLengths` routine to compute the effective lengths of the query and database sequences.

Discussion of the BLAST algorithm is complicated by the fact that there are several types of BLAST search. For instance, blastx takes a nucleotide query, translates it in six reading frames to an amino acid sequence and aligns each frame to a protein database. PSI-BLAST aligns an amino-acid query to a protein database using a position-specific score matrix. There are many more types of search, but despite their differences they are all conducted using a similar set of concepts. So while it is important to understand the differences between the various BLAST programs, we focus on unifying concepts whenever possible.

The purpose of this document is to put in one place, for the first time, all aspects of how Karlin-Altschul statistical parameters are used in the BLAST code. Development of this document exposed several, now resolved, inconsistencies within the code, and between the code and theory. At this time, there is production BLAST code in the NCBI C toolkit and development code in the NCBI C++ toolkit. This document refers to the version of the C++ toolkit BLAST code current as of January 2005, unless otherwise noted.

2 Karlin block objects

The Karlin-Altschul parameters λ, K and H are stored in objects of datatype `Blast_KarlinBlk`. The following definition of the `Blast_KarlinBlk` datatype was obtained from the file `blast_stat.h`.

```
typedef struct Blast_KarlinBlk {
  double  Lambda;
  double  K;
  double  logK;
  double  H;
  double  paramC;
} Blast_KarlinBlk;
```

We refer to objects of type `Blast_KarlinBlk` as Karlin blocks.

The `Lambda`, `H` and `K` fields correspond to the statistical parameters discussed in the previous section. The `logK` field holds the natural logarithm of K. The `paramC` field is used only in PHI-BLAST. (See Zhang et al. [13].) The field is set in the `s_PHIScoreBlkFill` or `Blast_ScoreBlkKbpGappedCalc` routine and used in the `Blast_HSPPHIGetEvalue` routine. We do not discuss PHI-BLAST or `paramC` further in this document.

2.1 Blast_KarlinBlkNew

`Blast_KarlinBlkNew` creates and returns a new Karlin block. It is declared in `blast_stat.h` with the following prototype.

```
Blast_KarlinBlk* Blast_KarlinBlkNew (void)
```

Memory for the block is allocated using `calloc`. Thus the various fields of the Karlin block are initialized with the bit pattern zero.

2.2 Blast_KarlinBlkFree

Karlin blocks are deleted by the routine `Blast_KarlinBlkFree`, which is defined in `blast_stat.c` with the following prototype.

```
Blast_KarlinBlk* Blast_KarlinBlkFree(Blast_KarlinBlk* kbp)
```

The return value of this routine is always `NULL`.

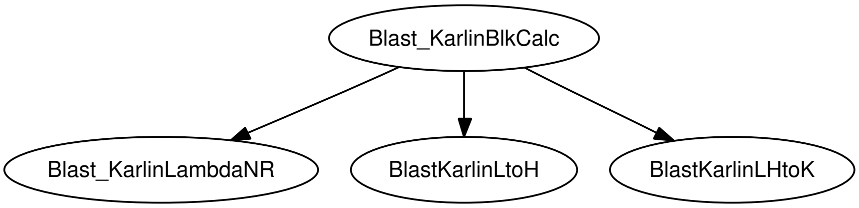

Figure 1: `Blast_KarlinBlkUngappedCalc` invokes several computational subroutines

3 Calculating λ, K and H for ungapped alignment

The routines in this section are used to find values for the Karlin-Altschul parameters λ, K and H for ungapped alignments. For a theoretical discussion of the computation of these parameters, see Karlin and Altschul [10].

The parameters are calculated from a set of score frequencies, which are represented by an object of type `Blast_ScoreFreq`. The score frequencies are based on the scoring matrix used and on the distribution of characters in the subject and query sequences. In some circumstances, the query or subject sequence is assumed to have a standard distribution of amino acids or nucleotides. In other circumstances, the actual composition of the query or subject sequence is used.

3.1 Blast_KarlinBlkUngappedCalc

The static routine `Blast_KarlinBlkUngappedCalc` is defined in `blast_stat.c` with the following prototype.

```
Int2
Blast_KarlinBlkUngappedCalc(Blast_KarlinBlk* kbp,
                            Blast_ScoreFreq* sfp)
```

The definition of the routine is preceded by a lengthy developer comment that may be found in Appendix A.1.

`Blast_KarlinBlkUngappedCalc` invokes several numerical routines (see Figure 1) to initialize the Karlin block pointed to by the function argument `kbp`. It calculates appropriate values of λ, H and K for an ungapped alignment based on the score frequencies provided by the `BLAST_ScoreFreq` object `sfp`. There are conditions under which λ, H and K are set to -1 to indicate an error. We discuss the usual case in this subsection and defer a discussion of the error conditions to section A.2.

To simplify discussion, we introduce some notation used throughout this section. In BLAST, scores are integer-valued. Let

$$P_j(i) = \text{the probability of a local alignment of length } j \text{ with score } i. \qquad (3)$$

There are only finitely many i for which $P_1(i) > 0$. Let

$$\ell = \min\{i \mid P_1(i) > 0\} \text{ and } u = \max\{i \mid P_1(i) > 0\}. \tag{4}$$

For theoretically valid scoring systems, ℓ must be negative, and u must be positive; see Karlin and Altschul [10]. The function $P_1(i)$ may be represented by an array p with initial index zero and length $u - \ell + 1$ such that

$$P_1(i) = p[i - \ell]. \tag{5}$$

The Blast_ScoreFreq object sfp has data members that represent the values ℓ, u and p. The fields are named obs_min, obs_max and sprob respectively. However, the pointer sprob has been manipulated so that sprob$[\ell] = p[0]$.

3.2 Blast_KarlinLambdaNR

The value of λ is computed by the Blast_KarlinLambdaNR routine, which is declared in blast_stat.h with the following prototype.

```
double
Blast_KarlinLambdaNR(Blast_ScoreFreq* sfp,
                     double initialLambdaGuess)
```

This routine uses a Newton-Raphson method to compute the unique positive solution to the equation

$$\phi(\lambda) = -1 + \sum_{i=\ell}^{u} P_1(i) e^{i\lambda} = 0. \tag{6}$$

Blast_KarlinLambdaNR does not apply Newton's method to $\phi(\lambda)$ directly, but rather invokes the NlmKarlinLambdaNR routine, which uses the algorithm given in section A.3.2 to solve equation (6).

3.3 BlastKarlinLtoH

The BlastKarlinLtoH routine is defined in blast_stat.c with prototype

```
static double
BlastKarlinLtoH(Blast_ScoreFreq* sfp, double lambda)
```

It computes H using the formula

$$H = \lambda \sum_{i=\ell}^{u} i P_1(i) e^{i\lambda}. \tag{7}$$

3.4 BlastKarlinLHtoK

The `BlastKarlinLHtoK` routine is defined in `blast_stat.c` as

```
static double
BlastKarlinLHtoK(Blast_ScoreFreq* sfp,
                 double lambda, double H)
```

The computation of K, performed by `BlastKarlinLHtoK`, is more complex than the calculation of λ or H. Assume $\ell \neq -1$ and $u \neq 1$; the cases in which $\ell = -1$ or $u = 1$ are treated separately and are discussed below. The `BlastKarlinLHtoK` computes a value $\bar{\sigma}$ that approximates the infinite sum

$$\sigma = \sum_{j=1}^{\infty} \frac{1}{j} \left[\sum_{i=-\infty}^{-1} P_j(i) e^{i\lambda} + \sum_{i=0}^{\infty} P_j(i) \right]. \tag{8}$$

The `BlastKarlinLHtoK` routine computes the values of $P_j(i)$ using the formula

$$P_j(i) = \sum_{k=-\infty}^{\infty} P_1(k) P_{j-1}(i-k), \tag{9}$$

which assumes that the probability of appending an item to the end of a sequence is independent of what has gone before. There are only finitely many i for which $P_1(i) > 0$, and thus an inductive argument using equation (9) shows that for any j, there are only finitely many i for which $P_j(i) > 0$. Therefore for any fixed value of j,

$$\sum_{i=-\infty}^{-1} P_j(i) e^{i\lambda} + \sum_{i=0}^{\infty} P_j(i)$$

may be computed in a finite number of operations. On the other hand, the computation must approximate the sum over all $j = 1, \ldots, \infty$. The sum over all j is truncated when terms in the sum become sufficiently small.

Once $\bar{\sigma}$ has been computed, K is obtained from the formula

$$K = \frac{\delta\lambda \exp(-2\bar{\sigma})}{H(1 - e^{-\delta\lambda})}, \tag{10}$$

where δ is the greatest common divisor of all scores that have nonzero probability. Usually $\delta = 1$.

If $u = \delta$ or $\ell = -\delta$, then the computation of $\bar{\sigma}$ is not performed. If both $u = \delta$ and $\ell = -\delta$, then

$$K = [P_1(\delta) - P_1(-\delta)]^2 / P_1(-\delta).$$

Otherwise, if $u = \delta$ but $\ell \neq -\delta$, then K is calculated by the formula

$$K = \frac{H}{\delta\lambda}(1 - e^{-\delta\lambda}). \tag{11}$$

In the remaining case, which is that $\ell = -\delta$ but $u \neq \delta$, the value of K is computed using the formula

$$K = \frac{\lambda(1 - e^{-\delta\lambda})}{\delta H} \left(\sum_{i=\ell}^{u} i P_1(i) \right)^2 . \tag{12}$$

4 Looking up λ, K and H for gapped alignment

The statistical theory for gapped alignments is not as complete as the theory for ungapped alignments. BLAST cannot calculate the Karlin-Altschul parameters, as it does for ungapped alignments, but rather looks them up in a table of precomputed values that have been obtained by simulation. (See Altschul et al. [4].)

4.1 Blast_KarlinBlkGappedCalc

The `Blast_KarlinBlkGappedCalc` routine is declared in the file `blast_stat.h` with the following prototype.

```
Int2
Blast_KarlinBlkGappedCalc(
    Blast_KarlinBlk* kbp, Int4 gap_open,
    Int4 gap_extend, Int4 decline_align,
    const char* matrix_name, Blast_Message** error_return)
```

The routine invokes the `Blast_KarlinBlkGappedFill` routine to look up Karlin-Altschul parameters. It reports diagnostic information to the user whenever the `Blast_KarlinkGapBlkFill` routine fails.

4.2 Blast_KarlinBlkGappedFill

The `Blast_KarlinBlkGappedFill` routine is declared in `blast_stat.h` to have the following prototype.

```
Int2
Blast_KarlinBlkGappedFill(Blast_KarlinBlk* kbp,
                          Int4 gap_open,
                          Int4 gap_extend,
                          Int4 decline_align,
                          const char* matrix_name)
```

The purpose of this routine is to obtain a set of Karlin-Altschul parameters appropriate for a specific choice of matrix and specific values of `gap_open`, `gap_extend` and `decline_align`.

Several named collections of parameter values, each corresponding to a particular scoring matrix, are defined in the file `blast_stat.c` as static global variables. These global variables are values of the Karlin-Altschul parameters for different settings of gap_open, gap_extend and decline_align. The meaning of parameters gap_open, gap_extend and decline_align is discussed in section 1.

The Blast_KarlinBlkGappedFill routine uses the value of its function argument matrix_name to access a collection of parameter values corresponding to a named score matrix. Within this collection, the data representing the parameter values is stored as a two-dimensional array of double precision values. Each row of the array stores the parameter values appropriate for a particular value of the triple (gap_open, gap_extend, decline_align). The following developer comment, found in blast_stat.c, describes the data format in more detail.

> How the statistical parameters for the matrices are stored:
>
> The parameters are stored in a two-dimensional array double (i.e., doubles), which has as its first dimensions the number of different gap existence and extension combinations and as it's [sic] second dimension 8. The eight different columns specify:
>
> (1) gap existence penalty (INT2_MAX denotes infinite); (2) gap extension penalty (INT2_MAX denotes infinite); (3) decline to align penalty (INT2_MAX denotes infinite); (4) λ; (5) K; (6) H; (7) α; and (8) β.

Note that in the developer comment the first column has index one, whereas in the C programming language the first column has index zero. The α and β parameters mentioned in the comment are related to effective length calculations; these parameters are discussed in Altschul et al. [4] and below in section 5.2.

Blast_KarlinBlkGappedFill searches the named collection for an array of scoring parameters that matches the function arguments gap_open, gap_extend and decline_align. An array is considered to be a match if all the following conditions hold:

- that gap_open matches the corresponding element in the array to the nearest integer;
- that gap_extend matches the corresponding element in the array to the nearest integer; and
- that either decline_align matches the corresponding element in the array to the nearest integer, or the element in the array is INT2_MAX, which is used to represent infinity.

If a match is found, the routine sets λ, H and K using the values stored in the array.

The error conditions that might occur are described in the following developer comment.

> return values:
>
> **-1** if matrix_name is NULL;
>
> **1** if matrix not found

2 if matrix found, but open, extend etc. values not supported.

If the routine is successful, it returns zero.

5 Routines for computing the effective size of the search space

An optimal alignment is less likely to start near the right edge of a sequence than it is to start away from that edge. To compensate for this effect, BLAST uses "effective lengths" of the database and query sequence when calculating statistics, rather than actual lengths. (See Altschul and Gish [5].)

Let m be the effective length of the query and m_a be its actual length. Similarly, let n be the effective length of the database and n_a be its actual length. Furthermore, let N be the number of sequences in the database. Then the effective lengths of the query and database are related to the actual lengths through the formulas

$$\begin{aligned} m &= m_a - \bar{\ell} \\ n &= n_a - N\bar{\ell}, \end{aligned}$$

where $\bar{\ell}$ is a nonnegative integer known as the *length adjustment*. The effective size of the search space is the product of m and n, in other words

$$\text{effective\_search\_space} = \left(m_a - \bar{\ell}\right)\left(n_a - N\bar{\ell}\right). \tag{14}$$

5.1 BLAST_CalcEffLengths

The `BLAST_CalcEffLengths` routine is declared in the file `blast_setup.h` with the following prototype.

```
Int2 BLAST_CalcEffLengths (EBlastProgramType program_number,
    const BlastScoringOptions* scoring_options,
    const BlastEffectiveLengthsParameters* eff_len_params,
    const BlastScoreBlk* sbp, BlastQueryInfo* query_info)
```

Each BLAST search has one or more query contexts. Depending on the type of search, a context may represent a distinct translation frame of the nucleotide query, a distinct strand of a double-stranded molecule or simply a distinct query sequence. In general each query context has a distinct length, and so a separate value of the length adjustment and an effective length must be calculated for each context. `BLAST_CalcEffLengths` computes these values and stores them in the appropriate locations in the arrays `eff_searchsp_array` and `length_adjustments`. These arrays are fields of the `BlastQueryInfo` object to which the function argument `query_info` points.

For each context, the `BLAST_CalcEffLengths` routine invokes the subroutine `BLAST_ComputeLengthAdjustment` to compute an appropriate value for the length adjustment. If `eff_len_params->options->searchsp_eff` is zero, the size of the effective search space is computed from the length adjustment using equation (14). On the other hand, if the `searchsp_eff` field is nonzero, then its value is used for the search space size in every context and the length adjustment and search space size may not satisfy equation (14).

5.2 BLAST_ComputeLengthAdjustment

The `BLAST_ComputeLengthAdjustment` routine computes the adjustment to the lengths of the query and database sequences that is used to compensate for edge effects when computing E-values. The routine is declared in `blast_stat.h` with the following prototype.

```
Int4
BLAST_ComputeLengthAdjustment(
    double K, double logK,
    double alpha_d_lambda, double beta,
    Int4 query_length, Int8 db_length, Int4 db_num_seqs,
    Int4 * length_adjustment)
```

The parameters to the routine have the following meanings:

`K` the statistical parameter K;

`logK` the natural logarithm of K;

`alpha_d_lambda` the ratio of the statistical parameters α and λ (for ungapped alignments the theoretically correct value of `alpha_d_lambda` is $1/H$);

`beta` the statistical parameter β (for ungapped alignments, $\beta = 0$);

`query_length` the length of the query sequence, which we denote m_a;

`db_length` the length of the database, which we denote n_a;

`db_num_seqs` the number of sequences in the database, which we denote N; and

`length_adjustment` the computed value of the length adjustment, which we denote $\bar{\ell}$.

The computed length adjustment $\bar{\ell}$ is an integer-valued approximation to the fixed point of the function

$$f(\ell) = \frac{\alpha}{\lambda} \ln \{K(m_a - \ell)(n_a - N\ell)\} + \beta,$$

The computed value $\bar{\ell}$ is always an integer smaller than the fixed point of $f(\ell)$. Usually, it will be the largest such integer. However, $\bar{\ell}$ is also restricted to satisfy the inequality

$$K \left(m_a - \bar{\ell}\right) \left(n_a - N\bar{\ell}\right) \geq \max(m_a, n_a).$$

or is zero in the extraordinary case that no positive $\bar{\ell}$ satisfies the inequality. Moreover, an iterative method described in Appendix B is used to compute $\bar{\ell}$, and under unusual circumstances the iterative method does not converge.

The routine returns zero if $\bar{\ell}$ is known to be the largest integer less than the fixed point of $f(\ell)$. Otherwise, it returns one.

6 Routines that compute statistics for a single distinct alignment

The following routines relate a "raw" score S to an E-value E using formula

$$E = Kmne^{-\lambda S},$$

where the integers m and n are the effective lengths of the query and database sequences, respectively. This formula is introduced in section 1 as equation (1). It is appropriate for evaluating the significance of a single distinct alignment only.

6.1 BLAST_KarlinStoE_simple

The `BLAST_KarlinStoE_simple` routine calculates an E-value E from a score S. The routine is defined in `blast_stat.c` with the following prototype.

```
double
BLAST_KarlinStoE_simple(Int4 S, const Blast_KarlinBlk* kbp,
                        Int8 searchsp)
```

In the notation of equation (1), `searchsp` is the effective search space, mn.

If any of the values λ, K or H is negative, then the function returns -1. Otherwise, it returns E, as computed by the formula

$$E = \text{searchsp} \times \exp\left(-\lambda S + \ln(K)\right). \tag{15}$$

The value of $\ln(K)$ is not computed by `BLAST_KarlinStoE_simple`. Instead, the cached value `kbp->logK` is used.

6.2 BlastKarlinEtoS_simple

`BlastKarlinEtoS_simple` computes a score S from an E-value E. This static routine is declared in the file `blast_stat.c` with the following prototype.

```
static Int4
BlastKarlinEtoS_simple(double E, const Blast_KarlinBlk* kbp,
                       double  searchsp)
```

In the notation of equation (1), the value of searchsp is mn.

This routine may be used to compute the minimum score a distinct alignment must attain to be assigned an E-value no larger than E. Thresholds used by the blast algorithm are often expressed as E-values, which are later converted to cutoff scores that are relative to a particular scoring system; see section 12 for a discussion of how this is done.

The routine uses the macro constants BLAST_SCORE_MIN, which is #defined in the file blast_stat.h to equal INT2_MIN, and BLASTKAR_SMALL_FLOAT, which is #defined to be 10^{-297} in blast_stat.c. If any of the values λ, K or H is negative, then the routine returns the value BLAST_SCORE_MIN. Otherwise, it returns S, computed as follows:

$$S = \left\lceil \ln\left((K \times \text{searchsp})/\widehat{E} \right)/\lambda \right\rceil, \tag{16}$$

where $\widehat{E} = \max(E, \texttt{BLASTKAR\_SMALL\_FLOAT})$.

7 Routines for evaluating multiple local alignments

For some types of search, BLAST joins multiple distinct alignments into linked sets and evaluates the statistical significance of each set as a whole. As of January 2005, this is done for ungapped alignments and for alignments in which one or both of the sequences is translated. For ungapped alignments, linked sets are calculated as an alternative to performing a gapped alignment. For translated queries, linked sets are computed to relate significant alignments that may not be in the same translation frame.

Three rules are needed to perform linking:

- a rule for producing permissible and promising linked sets from a collection of distinct alignments;
- a rule for evaluating the significance of a proposed linked set; and
- a rule for adjusting E-values to compensate for the effect of choosing the best among linked sets of differing sizes.

Because BLAST chooses the best among collections of linked sets with a differing number of elements, multiple tests are performed. Thus, an E-value obtained from equation (1) is not an appropriate measure of significance when linking is performed, even if a linked set contains only one alignment.

Section 11.3 discusses rules for partitioning a collection of distinct alignments into linked sets. This section discusses the BLAST_SmallGapSumE routine, the BLAST_LargeGapSumE routine and the BLAST_UnevenGapSumE routine, which are used to evaluate the statistical significance of linked sets. (See Altschul [2]; and Karlin and Altschul [11].)

Each of these routines is based on a different assumption about what restrictions are placed on the locations at which adjacent alignments may start. Placing a restriction on the number of starting positions effectively limits the maximum size of the gap between alignments. However, a zero-length gap or some overlap between alignments is permitted, so the interval of permitted starting points is always longer than the maximum gap size.

`BLAST_UnevenGapSumE` computes E-values when a restriction on the number of starting points is imposed for the query sequence, and a possibly different restriction is imposed for the subject sequence. `BLAST_SmallGapSumE` computes E-values in the special case in which the restriction is the same in both the query and the subject sequence. `BLAST_LargeGapSumE` computes E-values based on the assumption that no restriction has been imposed on the size of a gap between adjacent alignments.

The alignments in a linked set may come from distinct query contexts (see section 8 for a discussion of query contexts). Therefore, the computed E-value is based on a sum of the scores of the HSPs, each score individually normalized in units of nats. The sum score is

$$\texttt{xsum} = \sum_{i=1}^{N} (\lambda_i S_i - \ln K_i), \tag{17}$$

where S_i is the score of an individual alignment in the collection, and λ_i and K_i are the Karlin-Altschul statistical parameters appropriate for that alignment. It is convenient to allow the calling routine to compute the normalized sum score, and so the routines for computing E-values do not use Karlin-Altschul parameters directly.

The `BLAST_GapDecayDivisor` routine, discussed in this section, is the appropriate method for obtaining weights to compensate for the effect of performing multiple tests when evaluating linked sets. See section 7.4 for further discussion.

7.1 BLAST_UnevenGapSumE

The `BLAST_UnevenGapSumE` routine calculates the E-value of a collection of distinct alignments. It is used to compute E-values when a restriction on the number of starting points between adjacent alignments is imposed for the query sequence, and a possibly different restriction is imposed for the subject sequence.

The function declaration, found in `blast_stat.h`, is as follows.

```
double
BLAST_UnevenGapSumE(Int4 query_start_points,
                    Int4 subject_start_points,
                    Int2 num, double xsum,
                    Int4 query_length, Int4 subject_length,
                    Int8 searchsp_eff,
                    double weight_divisor)
```

The meaning of the various function arguments is as follows:

query_start_points the number of starting points permitted in the query sequence between adjacent alignments;

subject_start_points the number of starting points permitted in the subject sequence between adjacent alignments;

num the number of distinct alignments in this collection;

xsum the sum of the scores of these alignments, each individually normalized using appropriate values of λ and K (see equation (17));

query_length the effective length of the query sequence;

subject_length the effective length of the subject sequence;

searchsp_eff effective size of the search space; and

weight_divisor a divisor used to weight the E-value when multiple collections of alignments are being considered by the calling routine.

We represent the integers `num` by r, `query_length` by m, `subject_length` by n_S, `query_start_points` by g_P and `subject_start_points` by g_N. Let w represent the double-precision quantity `weight_divisor`.

The `BLAST_UnevenGapSumE` routine uses the following equations to compute its return value, denoted here by $\widehat{E}_S$.

$$\begin{aligned} S' &\leftarrow \text{xsum} - \ln(mn_S) - (r-1)(\ln g_P + \ln g_N) - \ln(r!) && \text{(18a)} \\ P_S &\leftarrow \text{BlastSumP}(r, S') && \text{(18b)} \\ E_S &\leftarrow \text{searchsp\_eff} \times \text{BlastKarlinPtoE}(P_S)/(mn_S) && \text{(18c)} \\ \widehat{E}_S &\leftarrow E_S/w && \text{(18d)} \end{aligned}$$

`BlastSumP` and `BlastKarlinPtoE` are routines defined in `blast_stat.c`. According to developer comments in `blast_stat.c`, for $r \neq 1$ the `BlastSumP` routine approximates the value

$$\frac{r^{r-2}}{(r-1)!(r-2)!} \int_{\text{xsum}}^{\infty} \exp(-y) \int_0^{\infty} x^{r-2} \exp(-\exp(x - y/r))\, dx\, dy. \tag{19}$$

For the special case of $r = 1$, the `BlastSumP` routine returns

$$1 - \exp[-e^{-\text{xsum}}]. \tag{20}$$

The `BlastKarlinPtoE` function is defined as follows:

$$\text{BlastKarlinPtoE}(p) = \begin{cases} \texttt{INT4\_MIN} & \text{if } p < 0 \text{ or } p > 1; \\ \texttt{INT4\_MAX} & \text{if } p = 1; \text{ and} \\ -\ln(-p+1) & \text{otherwise.} \end{cases} \tag{21}$$

The `BLAST_UnevenGapSumE` routine treats the case $r = 1$ specially. In this case, the E-value is

$$\widehat{E}_S = \text{searchsp\_eff} \times \exp(-\text{xsum})/w. \tag{22}$$

Ignoring numerical error, this value computed by this formula is the same as the value computed by the general rule (18) when $r = 1$. The special rule was introduced to eliminate the small differences due to numerical error that used to occur when `BLAST_SmallGapSumE` routine, the `BLAST_LargeGapSumE` and `BLAST_UnevenGapSumE` were applied to the same singleton linked set.

7.2 BLAST_SmallGapSumE

The `BLAST_SmallGapSumE` routine calculates the E-value of a collection of distinct alignments. This routine is a special case of the `BLAST_UnevenGapSumE` routine in which the restriction on the number of starting locations permitted between adjacent alignments is the same in both the query and the subject sequence.

The following declaration is found in `blast_stat.h`.

```
double
BLAST_SmallGapSumE(Int4 starting_points,
                   Int2 num, double xsum,
                   Int4 query_length, Int4 subject_length,
                   Int8 searchsp_eff,
                   double weight_divisor)
```

Each argument of `BLAST_SmallGapSumE` corresponds exactly to a function argument of `BLAST_UnevenGapSumE`, with one exception; the `starting_points` argument replaces the `query_start_points` and `subject_start_points` arguments of `BLAST_UnevenGapSumE`. The `starting_points` argument represents the number of starting points permitted between adjacent alignments in both the query and subject sequence.

We denote the the value of the `starting_points` argument by g, and refer to the rest of the arguments of `BLAST_SmallGapSumE` using the notation of section 7.1. The `BLAST_SmallGapSumE` routine uses the following equations to compute its return value, denoted here by $\widehat{E}_S$.

$$S' \leftarrow \mathrm{xsum} - \ln(mn_S) - (r-1)(2\ln g) - \ln(r!) \tag{23a}$$

$$P_S \leftarrow \mathrm{BlastSumP}(r, S') \tag{23b}$$

$$E_S \leftarrow \mathrm{searchsp\_eff} \times \mathrm{BlastKarlinPtoE}(P_S)/(mn_S) \tag{23c}$$

$$\widehat{E}_S = E_S/w, \tag{23d}$$

where `BlastSumP` and `BlastKarlinPtoE` are functions defined in `blast_stat.c` that are described above by equations (19), (20) and (21).

`BLAST_SmallGapSumE` uses the rule (22) in the special case that $r = 1$.

7.3 BLAST_LargeGapSumE

The `BLAST_LargeGapSumE` routine calculates the expected value of a collection of distinct alignments when no restriction is put on the size of the gap between adjacent alignments. The function declaration, found in blast_stat.h, is as follows.

```
double
BLAST_LargeGapSumE(Int2 num, double xsum,
                   Int4 query_length, Int4 subject_length,
                   Int8 searchsp_eff, double weight_divisor)
```

The names of the function arguments are the same as the names of the arguments to the function BLAST_UnevenGapSumE, and the arguments have the same meaning. However, the query_start_points and subject_start_points arguments are present in the function BLAST_UnevenGapSumE but not in the function BLAST_LargeGapSumE.

We refer to the arguments of BLAST_LargeGapSumE using the notation of section 7.1. The BLAST_LargeGapSumE routine uses the following equations to compute its return value, denoted here by $\widehat{E}_S$.

$$S' \leftarrow \mathrm{xsum} - r \ln mn_S - \ln(r!) \tag{24a}$$

$$P_S \leftarrow \mathrm{BlastSumP}(r, S') \tag{24b}$$

$$E_S \leftarrow \mathrm{searchsp\_eff} \times \mathrm{BlastKarlinPtoE}(P_S)/(mn_S) \tag{24c}$$

$$\widehat{E}_S \leftarrow E_S/w, \tag{24d}$$

where BlastSumP and BlastKarlinPtoE are routines, defined in blast_stat.c, that are described by equations (19), (20) and (21).

BLAST_LargeGapSumE uses the rule (22) in the special case that $r = 1$.

7.4 BLAST_GapDecayDivisor

An algorithm that searches for a statistically significant collection of multiple distinct alignments usually chooses the most significant of several collections. One must weight the E-value of each collection to compensate for the effect of choosing the best among collections of different size. A technique for weighting E-values to compensate for this effect is described in Altschul [2]; and Karlin and Altschul [11]. The BLAST_GapDecayDivisor routine computes the weights used by this technique.

The routine is defined in blast_stat.h with the following prototype.

```
double BLAST_GapDecayDivisor(double decayrate,
                             unsigned nsegs)
```

Let r denote the value of the nsegs parameter, α denote the value of decayrate and w denote the return value of the BLAST_GapDecayDivisor routine. The return value is computed by the formula

$$w = (1 - \alpha)\alpha^{r-1}.$$

For a collection of size r, one should divide the E-value by w *before* comparing the collection with other collections. The routines BLAST_SmallGapSumE, BLAST_LargeGapSumE and BLAST_UnevenGapSumE should each be passed the value of w in the parameter named weight_divisor to cause this division to be performed.

For both gapped and ungapped blastn searches, α is set to 0.5. For other BLAST programs, α is set to 0.5 for ungapped searches and 0.1 for gapped searches. The value of α is obtained from the gap_decay_rate field of an object of type BlastLinkHSPParameters. This field is set within the function BlastLinkHSPParametersNew when the BlastLinkHSPParameters object is first initialized. The BlastLinkHSPParametersNew routine initializes the gap_decay_rate field using the constant BLAST_GAP_DECAY_RATE or the constant BLAST_GAP_DECAY_RATE_GAPPED, both of which are #defined in the file blast_parameters.h.

8 Karlin block objects in BlastScoreBlk objects

Each BLAST search has one or more query contexts. Depending on the type of search, a context may represent a distinct translation frame of the nucleotide query, a distinct strand of a double-stranded molecule or simply a distinct query sequence. In general, each query context needs an individual set of Karlin-Altschul parameters to evaluate ungapped alignments and a different set to evaluate gapped alignments. Furthermore PSI-BLAST needs to have both the parameters for the position-specific search and parameters for the general search available. One purpose of BlastScoreBlk objects is to hold the several sets of Karlin block objects needed to perform a particular BLAST search.

Typically, Karlin blocks are accessed either through pointers passed as function arguments or through the data fields of an object of type BlastScoreBlk. Structures of type BlastScoreBlk have as fields four arrays of pointers to Karlin blocks: kbp_std, kbp_psi, kbp_gap_std and kbp_gap_psi. These arrays are dynamically allocated as separate arrays by BlastScoreBlkNew. All four arrays are allocated to have the same length, represented by the number_of_contexts field of the BlastScoreBlk. The BlastScoreBlkFree function deletes the dynamically allocated arrays and the Blast_KarlinBlk objects that they point to. The four arrays of Karlin blocks are initialized by one of several routines discussed in this section: s_PHIScoreBlkFill, Blast_ScoreBlkKbpUngappedCalc or Blast_ScoreBlkKbpGappedCalc.

In contrast, the kbp and kbp_gap fields of a BlastScoreBlk object are aliases to existing arrays, rather than arrays themselves. The kbp field always points to either kbp_std or kbp_psi. Similarly, the kbp_gap field always points to either kbp_gap_td or kbp_gap_psi. These pointers are used by routines that do not need to know whether or not a PSI-BLAST search is being performed; they only need an appropriate set of Karlin-Altschul parameters.

There is an anomalous use of aliasing among the arrays of Karlin blocks. The s_PHIScoreBlkFill routine sets kbp_std to be an alias to kbp_gap_std. There appears to be no other routine that treats these two arrays as equivalent.

A BlastScoreBlk object also has a field named kbp_ideal that is a pointer

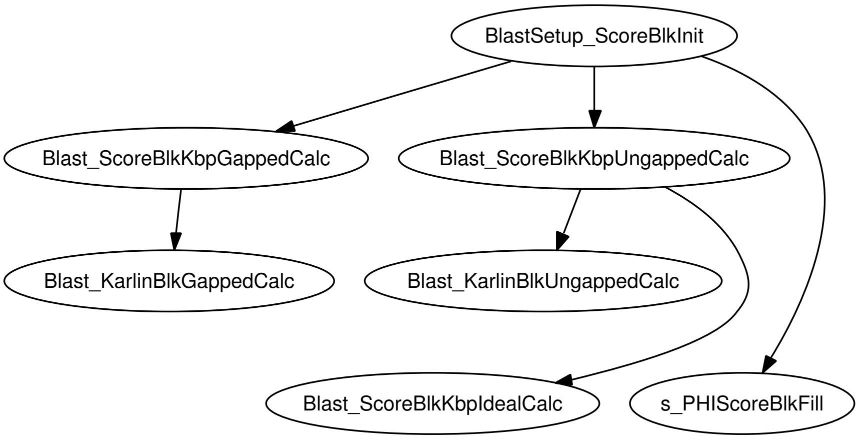
BlastSetup_ScoreBlkInit
Blast_ScoreBlkKbpGappedCalc
Blast_ScoreBlkKbpUngappedCalc
Blast_KarlinBlkGappedCalc
Blast_KarlinBlkUngappedCalc
Blast_ScoreBlkKbpIdealCalc
s_PHIScoreBlkFill

Figure 2: `BlastSetup_ScoreBlkInit` invokes one or more subroutines to initialize Karlin blocks within a `BlastScoreBlk`.

to a single Karlin block. This Karlin block holds values of the ungapped Karlin-Altschul parameters calculated as if both the query and the subject sequence had exactly average amino acid or nucleotide composition. For amino acid sequences, the average frequencies are obtained from Robinson and Robinson [12]. For nucleotide sequences, an average frequency of 0.25 is used for each nucleotide.

8.1 BlastSetup_ScoreBlkInit

The `BlastSetup_ScoreBlkInit` function is declared in the `blast_setup.h` file to have the prototype.

```
Int2
BlastSetup_ScoreBlkInit(
    BLAST_SequenceBlk* query_blk,
    BlastQueryInfo* query_info,
    const BlastScoringOptions* scoring_options,
    EBlastProgramType program_number,
    Boolean phi_align,
    BlastScoreBlk* *sbpp,
    double scale_factor,
    Blast_Message* *blast_message)
```

The purpose of `BlastSetup_ScoreBlkInit` is to initialize a `BlastScoreBlk`. `BlastSetup_ScoreBlkInit` invokes one or more subroutines to initialize the Karlin blocks within a `BlastScoreBlk`; see Figure 2. It invokes the function `s_PHIScoreBlkFill` to obtain all Karlin-Altschul parameters if the program is PHI-BLAST. Otherwise, it invokes the `Blast_ScoreBlkKbpUngappedCalc` routine to calculate the ungapped Karlin-Altschul parameters, and, if the search is gapped, invokes the `Blast_ScoreBlkKbpGappedCalc` routine to calculate the gapped parameters.

8.2 Blast_ScoreBlkKbpUngappedCalc

The `Blast_ScoreBlkKbpUngappedCalc` routine is defined in `blast_stat.c` with the following prototype.

```
Int2
Blast_ScoreBlkKbpUngappedCalc(EBlastProgramType program,
                              BlastScoreBlk* sbp,
                              Uint1* query,
                              BlastQueryInfo* query_info)
```

The `Blast_ScoreBlkKbpUngappedCalc` routine initializes those Karlin blocks within a `BlastScoreBlk` object that are used to evaluate the significance of ungapped alignments.

The routine first invokes `Blast_ScoreBlkKbpIdealCalc` to initialize the field `kbp_ideal`. For each query context, the routine invokes `BlastScoreFreqCalc` to calculate a set of amino acid or nucleotide frequencies and then invokes `Blast_KarlinBlkUngappedCalc` to initialize the Karlin blocks

```
sbp->kbp_std[context_number]
```

and

```
sbp->kbp_psi[context_number]
```

based on that set of amino acid or nucleotide frequencies. Both of these Karlin blocks are set to the same values, because `Blast_KarlinBlkUngappedCalc` is invoked twice with the same set of input parameters; there does not appear to be any intervening code that would alter the calculation.

For tblastx, blastx and RPS-tBLASTn, programs that all translate nucleotide queries, the computed values of λ in the `kbp_std` array are compared with the value of λ in the `kbp_ideal` field. If

$$\texttt{kbp\_std[context]->Lambda} \geq \texttt{kbp\_ideal->Lambda},$$

then `kbp_std[context]` is replaced with `kbp_ideal`. Therefore for translated searches, the smaller, more conservative λ is used.

8.3 Blast_ScoreBlkKbpGappedCalc

The routine `Blast_ScoreBlkKbpGappedCalc` is defined with the following prototype in the source file `blast_setup.c`.

```
Int2
Blast_ScoreBlkKbpGappedCalc(BlastScoreBlk * sbp,
    const BlastScoringOptions * scoring_options,
    EBlastProgramType program, BlastQueryInfo * query_info);
```

The purpose of this routine is to initialize the Karlin blocks that contain parameters for evaluating the significance of gapped alignments.

For ungapped alignments the Karlin-Altschul parameters may be calculated, but for gapped alignments the parameters are obtained by simulation. The required simulations have only been performed for protein alignments. If the program is blastn, then the Blast_ScoreBlkKbpGappedCalc routine simply duplicates the ungapped Karlin blocks. If the program is not blastn, then the Blast_ScoreBlkKbpGappedCalc routine calls the Blast_KarlinBlkGappedCalc routine to initialize the kbp_gap_std array.

If the program is not blastn, the Blast_KarlinBlkGappedCalc routine duplicates the values of the kbp_gap_std array in the kbp_gap_psi array. As of January 2005, the kbp_gap_psi array is not initialized for blastn, because PSI-BLAST has not yet been implemented for nucleotide sequences.

8.4 s_PHIScoreBlkFill

The static routine s_PHIScoreBlkFill is defined in blast_setup.c with the following prototype.

```
static Int2
s_PHIScoreBlkFill(
    BlastScoreBlk* sbp, const BlastScoringOptions* options,
    Blast_Message** blast_message)
```

The routine initializes the Karlin block sbp->kbp_gap_std[0]. It also makes kbp_std an alias for kbp_gap_std. This is the only place in the code that these two pointers are made aliases to the same array.

The values of λ, K and paramC are obtained from one of several precomputed sets of values. Which set of values is used depends on the matrix name, the gap open penalty and the gap extend penalty. In other words, the set of parameters chosen depends on the values the following three expressions:

- options->matrix;
- options->gap_open; and
- options->gap_extend.

s_PHIScoreBlkFill initializes only the first element of kbp_std and thus assumes that there is only one context. This is unlikely to be a problem since PHI-BLAST accepts only a single protein query.

8.5 Blast_ScoreBlkKbpIdealCalc

The Blast_ScoreBlkKbpIdealCalc routine is declared in the file blast_stat.h with the following prototype.

```
Int2 Blast_ScoreBlkKbpIdealCalc(BlastScoreBlk* sbp)
```

The purpose of this routine is to initialize the Karlin block sbp->kbp_ideal. This Karlin block holds values of the ungapped Karlin-Altschul parameters calculated as if both the query and the subject sequence had exactly average amino acid or nucleotide composition.

The Blast_KarlinBlkIdealCalc routine calls the BlastResFreqStdComp routine and the BlastScoreFreqCalc routine to obtain a "standard" object of type Blast_ScoreFreq that represents the residue frequencies. It passes this object to Blast_KarlinBlkUngappedCalc to obtain values for the statistical parameters.

9 Composition-based statistics

The routines of section 8 initialize gapped Karlin-Altschul parameters using precomputed tables generated by simulation using sequences of typical composition. The routines initialize the ungapped Karlin-Altschul parameters using the amino-acid or nucleotide composition of the query, but use a hypothetical subject sequence of standard composition.

BLAST is also able to evaluate alignments using a scoring system that takes both the composition of the query and the composition of a specific subject sequence into account. This feature is available for blastp and is under active development for tblastn. Composition-based scoring systems provide a more relevant measure of significance for sequences of biased composition than do standard scoring systems. To generate a composition-based scoring system, BLAST alters both the ungapped Karlin-Altschul parameters and the substitution matrix that is used to score alignments.

9.1 Kappa_RedoAlignmentCore

The routines for applying composition-based statistics are located in the file blast_kappa.c. The sole external entry point to the functionality in this file is the Kappa_RedoAlignmentCore, which has the following prototype.

```
Int2
Kappa_RedoAlignmentCore(
    EBlastProgramType program_number,
    BLAST_SequenceBlk * queryBlk,
    BlastQueryInfo* queryInfo,
    BlastScoreBlk* sbp,
    BlastHSPStream* hsp_stream,
    const BlastSeqSrc* seqSrc,
    const Uint1* gen_code_string,
    BlastScoringParameters* scoringParams,
    const BlastExtensionParameters* extendParams,
    const BlastHitSavingParameters* hitParams,
    const PSIBlastOptions* psiOptions,
    BlastHSPResults* results)
```

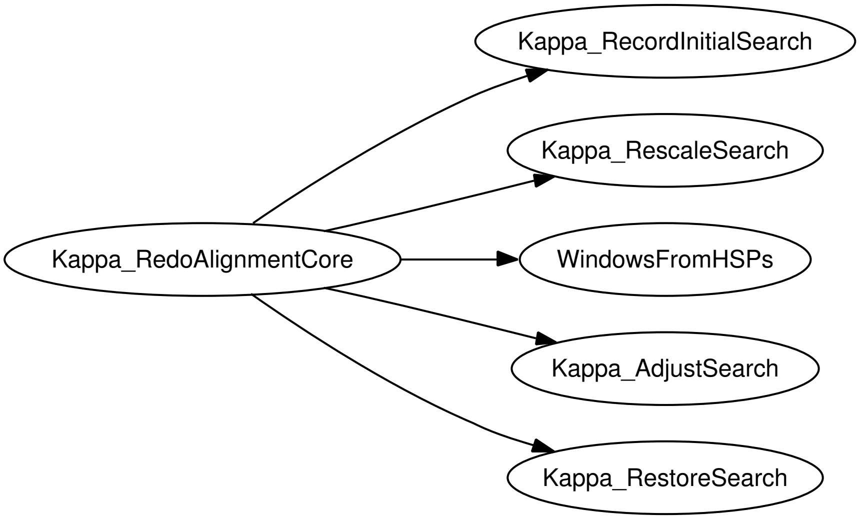

Figure 3: `Kappa_RedoAlignmentCore` uses several subroutines to implement composition-based statistics

The `Kappa_RedoAlignmentCore` routine takes a list of candidate alignments, generally representing several query-subject pairs, and recomputes the alignments and their scores. It optionally creates a composition-based scoring system for each query-subject pair before recomputing the alignments; the flag

```
extendParams->options->compositionBasedStats
```

determines whether or not composition-based statistics are applied.

As of January 2005, the `Kappa_RedoAlignmentCore` function is used only for blastp and PSI-BLAST searches. A version of the BLAST code that may use `Kappa_RedoAlignmentCore` for tblastn and PSI-tBLASTn searches is under active development, and much of the functionality for applying composition-based statistics to translated subject sequences is already present.

The `Kappa_RedoAlignmentCore` routine uses several subroutines to implement composition-based statistics; see Figure 3. `Kappa_RecordInitialSearch` records the initial values of all relevant parameters and `Kappa_RestoreSearch` restores these values. Thus, composition-based scoring systems are used only within the `Kappa_RedoAlignmentCore` routine, and the original scoring scoring system is restored before the routine exits. The remaining three routines, `Kappa_RescaleSearch`, `WindowsFromHSPs` and `Kappa_AdjustSearch`, are described in this section.

`Kappa_RedoAlignmentCore` also has the ability to recompute alignments for specific query-subject pairs using the rigorous Smith-Waterman algorithm; this is orthogonal to whether the scoring system has been altered for composition. The routines for computing a Smith-Waterman alignment do invoke

BLAST_KarlinStoE_simple to assign an E-value to the alignment.

9.2 Kappa_RescaleSearch

The static routine Kappa_RescaleSearch is defined in blast_kappa.c with prototype

```
static double
Kappa_RescaleSearch(Kappa_SearchParameters * sp,
                    BLAST_SequenceBlk* queryBlk,
                    BlastQueryInfo* queryInfo,
                    BlastScoreBlk* sbp,
                    BlastScoringParameters* scoringParams)
```

The purpose of Kappa_RescaleSearch is to alter the scale of the initial scoring system, in particular the initial scoring matrix. Because BLAST scores are integers, increasing the scale of the scoring system allows higher precision scores to be used. It is important to use these higher-precision scores when adjusting matrix entries to reflect the composition of the query and subject sequences.

If composition-based statistics are not being used, Kappa_RescaleSearch does nothing but return the floating-point value 1.0. Otherwise it returns the factor by which the scoring system is scaled. If the matrix is not BLOSUM62_20, then the scale factor is the value of the constant SCALING_FACTOR, which is #defined in blast_kappa.c to be 32. The BLOSUM62_20 matrix, which is used only for internal NCBI experiments, uses the scale factor SCALING_FACTOR/10, where the division is integer division.

The scale of a scoring system is reflected directly by the parameter λ. As equation (1) suggests, if scores are multiplied by a scale factor, then λ must be divided by the same factor if HSPs are to be assigned the same E-values. Rather than simply multiplying all scores by a factor, Kappa_RescaleSearch divides an ungapped λ by a scale factor to establish the desired scale of the scoring system. It then generates a matrix that nearly matches that scale. For blastp, the routine calls the computeScaledStandardMatrix, which generates an appropriately scaled matrix using a table of frequency ratios. By generating matrices in this fashion, Kappa_RescaleSearch avoids scaling the rounding errors contained in the standard matrices. As of January 2005, PSI-BLAST is still under active development within the NCBI C++ toolkit. When PSI-BLAST is fully implemented, Kappa_RescaleSearch will perform a similar computation using position-specific pseudo-frequencies.

9.3 WindowsFromHSPs

The static routine WindowsFromHSPs is defined in the file blast_kappa.c with the following prototype.

```
static void
WindowsFromHSPs(
    BlastHSP * hsp_array[], Int4 hspcnt, Int4 border,
    Int4 sequence_length, Kappa_WindowInfo ***pwindows,
    Int4 * nWindows, Int4 * lWindows, Int4 * window_of_hsp)
```

Windows are intervals in a translation frame of the subject sequence. This routine takes a list of HSPs and produces a list of windows, so that the subject range and translation frame of each HSP is contained in exactly one window. The range and frame of a window specifies the elements of the subject sequence that are used when composition-based statistics are computed. Recomputed alignments are also constrained to lie within their containing window.

For blastp and PSI-BLAST the WindowsFromHSPs routine creates exactly one window. The subject range is the entire length of the subject sequence, and the subject is untranslated so there is only one possible frame. For tblastn and PSI-tBLASTn there would typically be more than one window, and each window would be unlikely to include the entire length of the subject sequence. However, as of January 2005, composition-based statistics have not been enabled for tblastn and PSI-tBLASTn, and for these programs, the rules for generating a list of windows from a list of HSPs are the subject of active research.

9.4 Kappa_AdjustSearch

The static routine Kappa_AdjustSearch is defined in blast_kappa.c with the following prototype.

```
    static Int4
    Kappa_AdjustSearch(
        Kappa_SearchParameters * sp, Int4 queryLength,
        Kappa_SequenceData * subject, Int4 ** matrix)
```

This routine creates a composition-based scoring system.

Kappa_AdjustSearch uses the composition of the subject data, the composition of the query data and the given matrix to compute a set of score frequencies. It then invokes impalaKarlinLambdaNR with this set of frequencies to generate a value of λ appropriate for ungapped alignments of sequences with this composition. In general, the value of λ resulting from this computation is different from the scaled λ used by Kappa_RescaleSearch, which assumed a query and subject sequence of standard composition.

Let λ_C be the value of λ determined from the composition of the query and subject and let λ_S be the value of λ used by Kappa_RescaleSearch. The Kappa_AdjustSearch routine computes a restricted ratio of these two values

$$r_\lambda = \text{median}\{\texttt{LambdaRatioLowerBound},\ \lambda_C/\lambda_S,\ 1\},$$

where LambdaRatioLowerBound is a constant #defined in blast_kappa.c to be 0.5. It then invokes scaleMatrix function, which changes the scale of the entries in the scoring matrix by multiplying their unrounded, floating point values by r_λ and rounding the result to the nearest integer.

10 PSI-BLAST and RPS-BLAST

As of January 2005, PSI-BLAST and RPS-BLAST are still under development in the NCBI C++ toolkit code. PSI-BLAST creates a position-specific matrix based on the residue profile of a group of related alignments. It uses Karlin-Altschul parameters to determine the scale of the elements in the position-specific matrix. RPS-BLAST searches a set of position specific matrices for a good match to a protein sequence. We do not discuss these programs further in this document.

11 Routines operating on an array of HSPs

The routines of this section are part of the computational core of BLAST. Each of these routines performs some operation on an array of High Scoring Pairs (HSPs). An HSP is an object of type `BlastHSP`, defined in the file `blast_hits.h` as follows.

```
typedef struct BlastHSP {
  Int4 score;
  Int4 num_ident;
  double bit_score;
  double evalue;
  BlastSeg query;
  BlastSeg subject;
  Int4     context;
  GapEditBlock* gap_info;
  Int4 num;
  Uint4 pattern_length;
} BlastHSP;
```

Taken together, the `query`, `subject`, `context` and `gap_info` fields describe an alignment of a segment of the query sequence with a segment of a database sequence. The `gap_info` field, if not `NULL`, contains traceback information for the alignment, i.e. the location of gaps and contiguous segments within an alignment. The `query` and `subject` fields are of type `BlastSeg`, defined in `blast_hits.h` as follows.

```
typedef struct BlastSeg {
  Int2 frame;  /**< Translation frame */
  Int4 offset; /**< Start of hsp */
  Int4 length; /**< Length of hsp */
  Int4 end;    /**< End of HSP */
  Int4 gapped_start;/**< Where the gapped extension
                         started. */
} BlastSeg;
```

11.1 Blast_HSPListGetEvalues

The `Blast_HSPListGetEvalues` routine is declared in the file `blast_hits.h` with the following prototype.

```
Int2
Blast_HSPListGetEvalues(
    const BlastQueryInfo* query_info,
    BlastHSPList* hsp_list, Boolean gapped_calculation,
    BlastScoreBlk* sbp, double gap_decay_rate)
```

The routine calculates E-values of each HSP in the array `hsp_list` by invoking the `BLAST_KarlinStoE_simple` function with the `score` field of the HSP, an appropriate Karlin block and an appropriate effective length. Let `hsp` denote the HSP under consideration. If the program is not blastn and the calculation is gapped, then the following Karlin block is used.

```
sbp->kbp_gap[hsp->context]
```

Otherwise, the Karlin block

```
sbp->kbp[hsp->context]
```

is used. If `sbp->effective_search_sp` is not zero, then its value is used for the effective search space for every HSP. Otherwise the value of

```
query_info->eff_searchsp_array[hsp->context]
```

is used.

11.2 Blast_HSPListGetBitScores

The `Blast_HSPListGetBitScores` routine is declared in the file `blast_hits.h` with the following prototype.

```
Int2
Blast_HSPListGetBitScores(BlastHSPList* hsp_list,
                          Boolean gapped_calculation,
                          BlastScoreBlk* sbp)
```

It assigns normalized “bit” scores to each HSP in the list, using the formula

$$S_B = (\lambda S - \ln K)/\ln 2,$$

which was introduced as equation (2). For gapped searches, statistical parameters are taken from the Karlin block

`sbp->kbp_gap[hsp->context]`,

and for ungapped searches, they are taken from

`sbp->kbp[hsp->context]`.

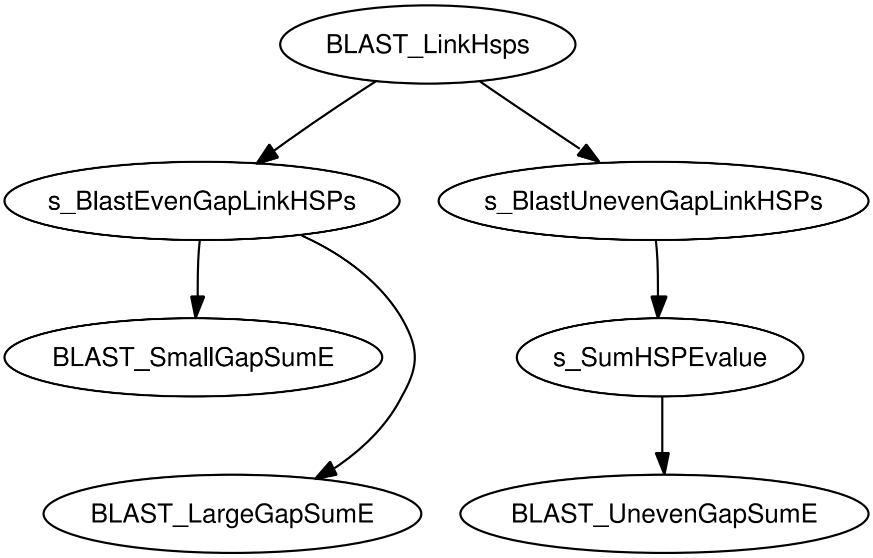

Figure 4: The `BLAST_LinkHsps` routine invokes either `s_BlastEvenGapLinkHSPs` or `s_BlastUnevenGapLinkHSPs`

11.3 HSP Linking For Sum Statistics

Section 7 discusses routines for evaluating the significance of a linked set of alignments. In this section, we discuss routines that partition a collection of distinct alignments into linked sets.

The external routine for applying HSP linking is `BLAST_LinkHsps`. This routine will call one of two subroutines to perform the necessary computation: `s_BlastEvenGapLinkHSPs` or `s_BlastUnevenGapLinkHSPs`. Each of these two routines uses a different algorithm for creating candidate linked sets.

11.3.1 BLAST_LinkHsps

The `BLAST_LinkHsps` routine is declared in the file `link_hsps.c` with the following prototype.

```
Int2
BLAST_LinkHsps(EBlastProgramType program_number,
               BlastHSPList* hsp_list,
               BlastQueryInfo* query_info,
               Int4 subject_length,
               BlastScoreBlk* sbp,
               const BlastLinkHSPParameters* link_hsp_params,
               Boolean gapped_calculation)
```

The BLAST_LinkHsps routine creates a set of doubly-linked lists of HSPs; each list represents an ordered collection of distinct local alignments and each HSP is contained in exactly one list. The routine uses sum statistics to evaluate the significance of multiple distinct alignments. Sum statistics are ultimately computed using Karlin blocks by involving the BLAST_SmallGapSumE, BLAST_LargeGapSumE and BLAST_UnevenGapSumE routines, as described in section 7.

The BLAST_LinkHsps routine does not perform any computation itself, but rather calls either s_BlastEvenGapLinkHSPs or s_BlastUnevenGapLinkHSPs to perform the computation; see Figure 4. The routine that is used is chosen by the following rule. If the program is tblastn or PSI-tBLASTn and

$$\texttt{hit\_params->options->longest\_intron} > 0,$$

then BLAST_LinkHsps invokes s_BlastUnevenGapLinkHSPs. Otherwise, it invokes s_BlastEvenGapLinkHSPs.

11.3.2 s_BlastUnevenGapLinkHSPs and s_SumHSPEvalue

The static s_BlastUnevenGapLinkHSPs routine is defined with the following prototype in the file link_hsps.c.

```
static Int2
s_BlastUnevenGapLinkHSPs(
     EBlastProgramType program,
     BlastHSPList* hsp_list, BlastQueryInfo* query_info,
     Int4 subject_length, BlastScoreBlk* sbp,
     const BlastLinkHSPParameters* link_hsp_params,
     Boolean gapped_calculation)
```

This routine is used predominantly for searches in which one sequence is translated, but the other is not. The terminology "Uneven Gap" refers to the fact that different, and possibly longer, gaps are allowed in between HSPs in the translated sequence than are allowed in the other sequence.

Not every set of HSPs may be linked together. We describe the rules used by s_BlastUnevenGapLinkHSPs to determine whether a sequence is admissible; the rules used by s_BlastEvenGapLinkHSPs are similar but differ in some details. We then outline, in Algorithm 11.1, the rules that s_BlastUnevenGapLinkHSPs uses to choose candidate sets to test for admissibility.

Let $\mathcal{A} = \{A_j\}$ for $j = 1, \ldots, \text{size}(\mathcal{A})$ be a collection of HSPs, sorted in ascending order by the offset in the query sequence. Let us introduce the following notation for parameters to the algorithm.

$$\begin{aligned} \gamma_s &= \texttt{link\_hsp\_params->longest\_intron} \\ \gamma_q &= \texttt{link\_hsp\_params->gap\_size} \\ \omega &= \texttt{link\_hsp\_params->overlap\_size}, \end{aligned}$$

where `link_hsp_params` is an argument to `s_BlastUnevenGapLinkHSPs`. The γ_s parameter represents the longest gap permitted in the subject sequence, which is typically a nucleotide sequence. The γ_q parameter is the longest gap permitted in the query, which is typically a protein sequence. The `BLAST_UnevenGapSumE` routine handles blastx searches specially; for blastx searches, it reverses the role of query and subject. It does this because for blastx, the query is the translated sequence. For searches that use translated sequences, the restrictions on the gap and overlap size are enforced on the translated sequence rather than on the original sequence.

A set $\{A_j \mid j \in J\}$ is admissible if the following seven conditions hold for all adjacent pairs of indices j, $k \in J$, with $k > j$:

$$\begin{aligned}
\text{query\_end}(A_k) &\geq \text{query\_end}(A_j); && \text{(25a)}\\
\text{query\_offset}(A_k) &\geq \text{query\_end}(A_j) - \omega; && \text{(25b)}\\
\text{query\_offset}(A_k) &\leq \text{query\_end}(A_j) + \gamma_q; && \text{(25c)}\\
\text{subject\_offset}(A_k) &\geq \text{subject\_offset}(A_j); && \text{(25d)}\\
\text{subject\_end}(A_k) &\geq \text{subject\_end}(A_j); && \text{(25e)}\\
\text{subject\_offset}(A_k) &\geq \text{subject\_end}(A_j) - \omega; \text{ and} && \text{(25f)}\\
\text{subject\_offset}(A_k) &\leq \text{subject\_end}(A_j) + \gamma_s. && \text{(25g)}
\end{aligned}$$

Note that because $\mathcal{A}$ is sorted by query offset in ascending order, it follows that for adjacent pairs of indices j, $k \in J$, $k > j$,

$$\text{query\_offset}(A_k) \geq \text{query\_offset}(A_j).$$

Note further that singleton sets are always admissible because there are no pairs of indices that must meet the conditions (25). If either the subject or query sequence is a DNA sequence, we impose the further condition that all alignments in the set are to the same strand of DNA.

Given an index set, the following function computes a normalized sum score, in nats, for those HSPs that correspond to an index in the set.

$$S(J) = \sum_{j \in J} [\lambda_j \times \text{score}(A_j) - \ln K_j], \tag{26}$$

where λ_j and K_j denote the values of the statistical parameters appropriate for use with A_j. If the search is gapped, the statistical parameters are taken from

```
sbp->kbp_gap[hsp->context],
```

where `hsp` refers to A_j. If the search is ungapped, the parameters are taken from

```
sbp->kbp[hsp->context].
```

From the normalized sum score, one can compute an E-value for a collection of alignments. This E-value is computed using the static `s_SumHSPEvalue` routine, defined in the `link_hsps.c` file with the following prototype.

```
static double
s_SumHSPEvalue(EBlastProgramType program_number,
    BlastQueryInfo* query_info, Int4 subject_length,
    const BlastLinkHSPParameters* link_hsp_params,
    LinkHSPStruct* head_hsp, LinkHSPStruct* new_hsp,
    double* xsum)
```

We do not describe this routine in detail, but simply note that it ultimately invokes `BLAST_UnevenGapSumE` which is discussed in detail in section 7. Let us denote by $E_U(J)$ the result of invoking `s_SumHSPEvalue` on the HSPs with indices in the set J.

The following pseudocode generates a collection of linked sets $\{L_i\}$, where $A_i \in L_i$ for each $i = 1, \ldots, \text{size}(\mathcal{A})$. The collection $\{L_i\}$ partitions the set $\mathcal{A}$ of HSPs. Note that if $A_i \in L_i$ and $A_j \in L_i$ then $L_i = L_j$, so in general $\{L_i\}$ will have fewer than size($\mathcal{A}$) unique members.

```
Algorithm 11.1. find_linked_sets($\mathcal{A}$, $\omega$, $\gamma_q$, $\gamma_s$)
Let $n = \text{size}(\mathcal{A})$.
$L_k \leftarrow \{A_k\}$ for $k = 1, \ldots, n$
while $\mathcal{A} \neq \emptyset$ do
    Choose $A_i$ so that $\text{score}(A_k) = \max\{\text{score}(A_k) \mid A_k \in \mathcal{A}\}$.
    $N \leftarrow L_i$
    while $\{L_k \mid L_k \neq N$ and $L_k \cup N$ is admissible$\}$ is not empty do
        Choose $L_j$ so that
            $E_U(L_k \cup N) = \min\{E_U(L_k \cup N) \mid L_k \cup N$ is admissible$\}$.
        if $E_U(L_j \cup N) < E_U(N)$ and $E_U(L_j \cup N) < E_U(L_j)$ then
            $N \leftarrow L_j \cup N$
            $L_k \leftarrow N$ for all $k$ such that $A_k \in N$
        else
            break
        end if
    end do
    $\mathcal{A} \leftarrow \mathcal{A} \setminus N$
end do
```

We leave unspecified the rules for choosing an HSP with highest score and linked set with smallest E-value, but assert that ties are broken deterministically. The parameters ω, γ_q and γ_s are used to test whether a linked set is admissible, using the rules (25).

11.3.3 s_BlastEvenGapLinkHSPs

The static routine `s_BlastEvenGapLinkHSPs` is defined in the file `link_hsps.c` with the following prototype.

```
static Int2
s_BlastEvenGapLinkHSPs(EBlastProgramType program_number,
    BlastHSPList* hsp_list,
    BlastQueryInfo* query_info, Int4 subject_length,
    BlastScoreBlk* sbp,
    const BlastLinkHSPParameters* link_hsp_params,
    Boolean gapped_calculation)
```

Like s_BlastUnevenGapLinkHSPs, this routine uses a heuristic to generate a collection of linked sets. The s_BlastEvenGapLinkHSPs routine is the older of the two and suffers from the restriction that the maximum gap size in the query and the subject be the same. It has been retained for backwards compatibility with earlier versions of BLAST and is not recommended for use in new code.

The rules for which linked sets are admissible are similar to the rules given by (25). There are three important differences. The first is that, as previously mentioned, the maximum size of the gap in the query must equal the maximum size of the gap in the subject. For the remainder of this section, these two equivalent values will be denoted by γ.

The second difference is that the maximum permitted overlap between to distinct alignments is a function of the length of each alignment. Rather than stating the rules for admissibility in detail, we simply assert that a suitable function

$$\textsc{admissible}(\mathcal{A}, I, \gamma)$$

may be defined, where $\mathcal{A}$ is an array of HSPs and I is a collection of indices into $\mathcal{A}$. Moreover, s_BlastEvenGapLinkHSPs may consider both sets for which the value of γ is finite and sets for which γ is infinite. These cases are know as the "small gap rule" and "large gap rule" respectively.

The significance of linked sets admissible to the small gap rule is evaluated using the BLAST_SmallGapSumE routine, whereas the significance of linked sets admissible to the large gap rule is evaluated using the BLAST_LargeGapSumE routine. In either case, the sum score for a linked set of HSPs is computed using equation (26). For a discussion of the other parameters to the sum statistics routines, see section 7. In this section, we denote the result of the BLAST_SmallGapSumE routine with appropriate parameters for linked set L by $E_S(L)$ and the result of the BLAST_LargeGapSumE routine by $E_L(L)$.

The general operation of the s_BlastEvenGapLinkHSPs routine is to find a heuristically desirable linked set admissible to the large gap rule and a desirable linked set admissible to the small gap rule. It evaluates the significance of each linked set and chooses the better of the two. As described below in the text and in Algorithm 11.3, the parameter link_hsp_params->gap_prob is used to weight the E-values of these linked sets to compensate for the effect of choosing the better of the two. It must be noted, however, that there are parameter settings that will cause the routine to use either the small or large gap rule exclusively.

Once a linked set has been chosen, it removes the set from further consideration, and reapplies the algorithm to the remaining HSPs. The desirability of a given linked set is measured by a weighted sum function

$$w(\mathcal{A}, I, \nu) = \sum_{i \in I} \left(\text{score}(A_i) - \nu \right),$$

where I is an index set and ν a positive integer weight; the weight for the small gap rule is typically different than the weight for the large gap rule. The following pseudocode is a simplified version of the algorithm that `link_hsps` uses to find the optimal linked set for a given weight ν and (possibly infinite) gap size γ.

ALGORITHM 11.2. FIND_BEST_WEIGHTED_SCORE($\mathcal{A}$, ν, γ)

```
$I_0 \leftarrow \emptyset$; best $\leftarrow 0$
for $i = 1, \ldots, \text{size}(\mathcal{A})$ do
   $j \leftarrow 0$
   if score$(A_i) > \nu$ then
       for $k = i - 1$ down to $0$ do
           if ADMISSIBLE$(\mathcal{A}, I_k \cup \{i\}, \gamma)$ and
               $w(\mathcal{A}, I_k \cup \{i\}, \nu) > w(\mathcal{A}, I_j \cup \{i\}, \nu)$
           then
               $j \leftarrow k$
           end if
       end do
   end if
   $I_i \leftarrow I_j \cup \{i\}$
   if best $= 0$ or $w(\mathcal{A}, I_i, \nu) \geq w(\mathcal{A}, I_{\text{best}}, \nu)$ then best $\leftarrow i$; end if
end do
return $I_{\text{best}}$
```

The set $I_0 \cup \{i\} = \{i\}$ is always admissible, so for $0 \leq i \leq \text{size}(\mathcal{A})$ the index set I_i exists. Algorithm 11.3 returns an index set that yields a maximal weighted score. The algorithm omits many details present in the BLAST code that accelerate the search but that do not affect the answer.

We introduce the following notation for parameters to the algorithm.

$$\begin{aligned} \gamma &= \texttt{link\_hsp\_params->gap\_size} \\ \nu_S &= \texttt{link\_hsp\_params->cutoff\_small\_gap} \\ \nu_L &= \texttt{link\_hsp\_params->cutoff\_large\_gap} \\ \beta &= \texttt{link\_hsp\_params->gap\_prob}, \end{aligned}$$

where `link_hsp_params` is an argument to `s_BlastEvenGapLinkHSPs`. The rule the routine uses to choose a linked set is similar to the following pseudocode.

```
ALGORITHM 11.3. FIND_BEST($\mathcal{A}$, $\nu_L$, $\nu_S$, $\beta$, $\gamma$)
Let $I_L \leftarrow$ FIND_BEST_WEIGHTED_SCORE($\mathcal{A}$, $\nu_L$, $\infty$)
if $\nu_S = 0$ then
    return $I_L$
else
    Let $e_L \leftarrow E_L\left(\{A_i \in \mathcal{A} \mid i \in I_L\}\right)$.
    if size($I_L$) $> 1$ then
        if $1 - \beta \neq 0$ then $e_L \leftarrow e_L/(1-\beta)$; else $e_L \leftarrow \infty$; end if
    end if
    Let $I_S \leftarrow$ FIND_BEST_WEIGHTED_SCORE($\mathcal{A}$, $\nu_S$, $\gamma$)
    Let $e_S \leftarrow E_S\left(\{A_i \in \mathcal{A} \mid i \in I_S\}\right)$.
    if size($I_S$) $> 1$ then
        if $\beta \neq 0$ then $e_S \leftarrow e_S/\beta$; else $e_S \leftarrow \infty$; end if
    end if
    if $e_S \leq e_L$ then
        return $I_S$
    else
        return $I_L$
    end if
end if
```

The `s_BlastEvenGapLinkHSPs` routine removes the resulting linked set from the collection of HSPs to be considered for linking, and reapplies the equivalent of Algorithm 11.3 on the remaining HSPs, until the set of remaining HSPs is empty. The `s_BlastEvenGapLinkHSPs` routine is able to take advantage of partial results from one run of Algorithm 11.3 to accelerate the next run. We do not discuss this type of optimization here; Algorithm 11.3 is intended to be only a rough sketch of the operation of `s_BlastEvenGapLinkHSPs`.

12 Routines that initialize parameters used to compute alignments

BLAST generates a collection of HSPs in stages. BLAST starts with one or more *word hits*, short matches between a query and subject sequence, and extends them into ungapped and then gapped alignments. At various stages in the process, BLAST will evaluate the significance of the alignment produced so far to determine whether to continue processing the current hit.

The routines of this section find values for cutoff scores for allowing a current hit to proceed from one stage of the BLAST algorithm to the next. The routines also compute a different set of cutoff scores, known as x-drop values, that control when BLAST stops searching for an optimal extension. In general, cutoff scores are calculated in one of two ways. In some cases, `BLAST_Cutoffs` routine is used to relate an E-value to a score sufficient to produce that E-value. In other cases, cutoff scores are defined as constant, scale-independent bit scores. These bit scores are rescaled to lie in the current scoring system, through the use of appropriate values of λ and K.

12.1 s_BlastFindValidKarlinBlk

Sometimes it is not possible to compute Karlin-Altschul parameters for all frames of a translated query. It may be that a frame has severely atypical composition which results in a positive average score, or it may be that a query frame is completely masked. In any case, the s_BlastFindValidKarlinBlk searches an array of Karlin blocks to find the first element for which H, K and λ are all positive; negative values are used to indicate that the parameters could not be computed.

The static s_BlastFindValidKarlinBlk routine is defined with the following prototype in the file blast_parameters.c.

```
static Int2
s_BlastFindValidKarlinBlk(Blast_KarlinBlk** kbp_in,
                          const BlastQueryInfo* query_info,
                          Blast_KarlinBlk** kbp_ret)
```

12.2 BLAST_Cutoffs

The BLAST_Cutoffs routine computes the minimum score that must be attained by an HSP for that HSP to proceed to the next stage of the BLAST algorithm. The cutoff score is based on the number of HSPs that are expected to achieve a score that is at least that large; see Altschul et al. [6].

The function is declared in blast_stat.h as follows.

```
Int2
BLAST_Cutoffs(Int4 *S, double* E,
              Blast_KarlinBlk* kbp, Int8 searchsp,
              Boolean dodecay, double gap_decay_rate)
```

The argument kbp is a Karlin block that supplies statistical parameters that are used to convert between a score and an E-value. The searchsp argument is the effective size of the search space, which is also needed to convert between scores and E-values. The do_decay parameter is a flag that, if true, indicates that E-values are to be weighted to compensate for the effect of comparing collections of multiple distinct alignments with a varying number of elements. If do_decay is true, then gap_decay_rate is used to compute an appropriate weight; see section 7.4.

In general, the "*E" and "*S" parameters have different values on entry than they do on exit. In this section, we use E and S to indicate the values of the parameters on entry and use $\widehat{E}$ and $\widehat{S}$ to indicate the corresponding values on exit.

Unless the BLAST_Cutoffs routine is passed exceptional values for its input parameters, it computes the smallest value of $\widehat{S}$ that yields an E-value no larger than E. To express the operation the routine in reasonably compact mathematical notation, we introduce the following functions.

$$\begin{aligned} E_S(S) &= \text{BLAST\_KarlinStoE}(S, \text{kbp}, \text{searchsp}) && \text{(27a)} \\ S_E(E) &= \text{BlastKarlinEtoS}(E, \text{kbp}, \text{searchsp}) && \text{(27b)} \end{aligned}$$

We also introduce the scalar

$$\mu = \begin{cases} \texttt{BLAST\_GapDecayDivisor}(r, 1) & \text{if } \texttt{dodecay} = \texttt{TRUE} \text{ and } 0 < r < 1; \\ 1 & \text{otherwise,} \end{cases} \tag{28}$$

where r is the function argument `gap_decay_rate`. Under normal conditions,

$$\widehat{S} = S_E(\mu E) \text{ and } \widehat{E} = E. \tag{29}$$

The `BLAST_Cutoffs` routine contains code to handle exceptional values of S and E. In the BLAST code, as of January 2005, S is always explicitly set to zero in the calling routine before `BLAST_Cutoffs` is invoked. However, the `BLAST_Cutoffs` routine has code to handle the case in which S is positive. Similarly, the routine has code to handle the case in which E is nonpositive; we see no valid circumstance in the current calls to `BLAST_Cutoffs` in which the value passed as E would be nonpositive.

The logic that the `BLAST_Cutoffs` routine uses to compute $\widehat{S}$ and $\widehat{E}$, including logic to handle exceptional values of S and E, follows. We emphasize that for nonexceptional values of S and E, this logic reduces to equation (29).

If $E \leq 0$, then $\widehat{E}$ and $\widehat{S}$ are computed as follows.

$$\widehat{S} = \max(S, 1) \text{ and } \widehat{E} = E_S(\widehat{S})/\mu.$$

If, on the other hand, $E > 0$, then the values are computed by the equations

$$\begin{aligned} \widehat{S} &= \max(S, S_E(\mu E)) \\ \widehat{E} &= \begin{cases} E_S(\widehat{S})/\mu & \text{if } S = \widehat{S}; \\ E & \text{otherwise.} \end{cases} \end{aligned}$$

12.3 Cutoff values used to compute and save ungapped alignments

Cutoff values used to compute and save ungapped alignments are stored in an object of type `BlastHitSavingParameters`, which is defined in the file `blast_parameters.h` to be

```
typedef struct BlastInitialWordParameters {
    BlastInitialWordOptions* options;
    Int4 x_dropoff_init;
    Int4 x_dropoff;
    Int4 cutoff_score;
} BlastInitialWordParameters
```

The options field contains values that are used to compute the other fields of a BlastInitialWordParameters, but which do not depend on the scoring system. The cutoff_score field is the minimum score an ungapped alignment must attain to be saved for processing by the next stage of BLAST. If an ungapped search is being performed, then the next stage is the computation of linked sets of HSPs. If a gapped search is being performed, then the next stage is gapped extension. In the case of a gapped search, not all ungapped alignments saved are ultimately extended, due to tests that eliminate alignments completely contained within the endpoints of a higher-scoring gapped alignment. We do not discuss these containment tests further in this document.

The x_dropoff field is a cutoff used during an ungapped extension. Whenever the current score of an ungapped leftward or rightward extension falls more the x_dropoff below the current best score for the extension, the algorithm stops extending the alignment in that direction. The x_dropoff_init field is set in the BlastInitialWordParametersNew function and used to set x_dropoff in the BlastInitialWordParametersUpdate function.

We remark that the gap_x_dropoff and gap_x_dropoff_final fields of a BlastExtensionParameters object serve a similar purpose to the x_dropoff field of a BlastInitialWordParameters object, but are used exclusively for *gapped* extensions. The BlastExtensionParameters datatype is discussed in Section 12.4.

12.3.1 BlastInitialWordParametersNew

The BlastInitialWordParametersNew routine creates and initializes a new instance of BlastInitialWordParameters. The routine is declared in the file blast_parameters.h as follows.

```
Int2
BlastInitialWordParametersNew(
    EBlastProgramType program_number,
    const BlastInitialWordOptions* word_options,
    const BlastHitSavingParameters* hit_params,
    BlastScoreBlk* sbp,
    BlastQueryInfo* query_info,
    Uint4 subject_length,
    BlastInitialWordParameters* *parameters)
```

The options field of the structure is given the value of the function argument named word_options. The x_dropoff_init field is set using the formula

$$\texttt{x\_dropoff\_init} = \left\lceil r \times \frac{d_0 \ln 2}{\lambda} \right\rceil, \tag{31}$$

where d_0 is the value of word_options->x_dropoff and the factor r is the value of sbp->scale_factor. As of January 2005, sbp->scale_factor is set to the constant 1.0 in blast_stat.c, but this is expected to change as more modules are ported from the C toolkit to the C++ toolkit. The value of λ used by equation (31) is taken from the Karlin block found when the routine s_BlastFindValidKarlinBlk is applied to the array sbp->kbp_std. The field x_dropoff represents the dropoff value of an ungapped extension, so the value of λ from kbp_std is used.

A call to BlastInitialWordParametersUpdate initializes the other fields of the BlastInitialWordParameters object.

12.3.2 BlastInitialWordParametersUpdate

The BlastInitialWordParametersUpdate routine is declared with the following prototype in the file blast_parameters.h.

```
Int2
BlastInitialWordParametersUpdate(
    EBlastProgramType program_number,
    const BlastHitSavingParameters* hit_params,
    BlastScoreBlk* sbp,
    BlastQueryInfo* query_info, Uint4 subj_length,
    BlastInitialWordParameters* parameters)
```

This routine finishes the calculation of parameters that is begun by the routine BlastInitialWordParametersNew. For some types of search, it is also used to calculate parameters specific to each subject sequence.

To calculate a value for the cutoff_score field of the function argument parameters, this routine first calculates an intermediate value that is denoted by cutoff_s. If sbp->scale_factor > 0 it scales cutoff_s before proceeding, by executing the following line:

```
cutoff_s *= (Int4)sbp->scale_factor;
```

The routine then chooses cutoff_score to be the smaller of cutoff_s and hit_params->cutoff_score_max.

The calculation of cutoff_s is done differently for blastn than it is done for other searches. Let us discuss first how cutoff_s is calculated for searches other than blastn; we discuss the case of blastn later. Let gap_trigger be defined by the formula

$$\texttt{gap\_trigger} = (t_0 \ln 2 + \ln K_u)/\lambda_u,$$

where t_0 is the value of

```
parameters->options->gap_trigger
```

and λ_u and K_u are obtained by applying s_BlastFindValidKarlinBlk to the array sbp->kbp_std. For gapped searches cutoff_s = gap_trigger. If the search is ungapped, then an additional score, denoted here by S_E, is calculated by applying BLAST_Cutoffs to the E-value returned by s_GetCutoffEvalue. The s_GetCutoffEvalue routine returns a constant value for each type of search.

The statistical parameters used by BLAST_Cutoffs are obtained by applying s_BlastFindValidKarlinBlk to the array sbp->kbp_std. The other parameters to BLAST_Cutoffs are described below. For ungapped searches, cutoff_s is set to the smaller of gap_trigger and S_E.

For blastn searches, the value of gap_trigger is ignored. Instead S_E computed by invoking BLAST_Cutoffs as above, except that for gapped blastn the Karlin block is obtained by applying s_BlastFindValidKarlinBlk to the array sbp->kbp_gap. The routine then sets cutoff_s to S_E unconditionally.

In all the calls to BLAST_Cutoffs mentioned above, the effective search space size is given by the expression

```
MIN(subj_length, (Uint4) avg_qlen)*subj_length
```

where subj_length is a function argument and avg_qlen is the average length of a query sequence, taken over all contexts.

BLAST_Cutoffs optionally applies an adjustment to the input E-value to compensate for the effect of choosing the best among several linked sets of HSPs if sum statistics are used. If sum statistics are be used, the decay rate is the value of

```
hit_params->link_hsp_params->gap_decay_rate
```

For gapped blastn, and any other searches for which sum statistics are disabled, the decay rate is set to zero, which has the effect of causing the decay rate to be ignored.

Finally, BlastInitialWordParametersUpdate also sets the field x_dropoff. Normally, x_dropoff is set to the value of the field x_dropoff_init, but there is code to handle exceptional circumstances. If both the x_dropoff_init field and the cutoff_score field are nonzero, then x_dropoff is set to the lesser of cutoff_score and the field x_dropoff_init. Because x_dropoff_init is normally less than the minimum score of a seed word, it would be exceptional for cutoff_score to be smaller than x_dropoff_init. Furthermore, if x_dropoff_init is zero and cutoff_score is nonzero, then x_dropoff would be set unconditionally to cutoff_score. The default value of x_dropoff_init corresponds to a score of 7 or 20 bits, and it would be extraordinary to use a scoring system where these bit scores scale to zero. However, some BLAST executables provide a command line option that can be used to override the default value of x_dropoff_init.

12.4 BlastExtensionParametersNew

The BlastExtensionParametersNew routine creates and initializes a new instance of BlastExtensionParameters. The function is declared in the file blast_parameters.h with the following prototype.

```
    Int2 BlastExtensionParametersNew(
        EBlastProgramType program_number,
        const BlastExtensionOptions* options,
        BlastScoreBlk* sbp,
        BlastQueryInfo* query_info,
        BlastExtensionParameters* *parameters)
```

The `BlastExtensionParameters` datatype is defined in `blast_parameters.h` as follows.

```
   typedef struct BlastExtensionParameters {
       BlastExtensionOptions* options;
       Int4 gap_x_dropoff;
       Int4 gap_x_dropoff_final;
   } BlastExtensionParameters;
```

The datatype is closely related to the `BlastExtensionOptions` datatype, which contains many similarly named fields. `BlastExtensionOptions` datatype is defined in `blast_options.h` as follows.

```
    typedef struct BlastExtensionOptions {
        double gap_x_dropoff;
        double gap_x_dropoff_final;
        EBlastPrelimGapExt ePrelimGapExt;
        EBlastTbackExt eTbackExt;
        Boolean compositionBasedStats;
} BlastExtensionOptions;
```

The distinction between the similarly named fields of two datatypes is that scale independent, "bit" values are stored as double precision values in an instance of `BlastExtensionOptions`, whereas the corresponding values scaled to a particular value of λ and K are stored as integers in an instance of `BlastExtensionParameters`.

Let d and d^F represent the `gap_x_dropoff` and `gap_x_dropoff_final` fields of the `BlastExtensionParameters` object. Let d_0 and d_0^F be the corresponding fields of the `BlastExtensionOptions` object. For any given set of parameters λ and K, the fields are related by the following equations.

$$\begin{aligned} d &= \lfloor d_0 \ln 2/\lambda \rfloor \\ d^F &= \lfloor d_0^F \ln 2/\lambda \rfloor \end{aligned}$$

The value of λ is taken from the Karlin block obtained by applying the routine `s_BlastFindValidKarlinBlk` to the `sbp->kbp_gap` array.

If `sbp->scale_factor` > 1, then the computed values of `gap_x_dropoff` and `gap_x_dropoff_final` are rescaled by multiplication by `sbp->scale_factor`.

12.5 Routines that set fields in BlastHitSavingParameters object

The BlastHitSavingParameters datatype is defined in blast_parameters.h as follows.

```
typedef struct BlastHitSavingParameters {
    BlastHitSavingOptions* options;
    Int4 cutoff_score;
    Int4 cutoff_score_max;
    BlastLinkHSPParameters* link_hsp_params;
} BlastHitSavingParameters;
```

It is difficult to characterize exactly what a BlastHitSavingParameters object represents. The cutoff_score field of the object represents the minimum score that must be attained by an HSP after a gapped extension for that HSP to proceed to the next stage of the BLAST algorithm. For ungapped searches, the field is not relevant. The cutoff_score_max field is the largest permissible value of the cutoff_score field. The BlastInitialWordParametersUpdate function also uses cutoff_score_max field of a BlastHitSavingParameters object as the maximum permissible value of the cutoff for saving ungapped extensions. If HSP linking is to be performed, then link_hsp_params is set to a non-nil value.

12.5.1 BlastHitSavingParametersNew

The BlastHitSavingParametersNew routine creates and initializes a new instance of the BlastHitSavingParameters datatype. The function is declared in the file blast_parameters.h with the following prototype.

```
Int2
BlastHitSavingParametersNew(
    EBlastProgramType program_number,
    const BlastHitSavingOptions* options,
    BlastScoreBlk* sbp, BlastQueryInfo* query_info,
    Int4 avg_subj_length,
    BlastHitSavingParameters* *parameters)
```

The BlastHitSavingParameters code does not use Karlin-Altschul parameters directly, but rather invokes BlastHitSavingParametersUpdate, which uses the scoring parameters to initialize the cutoff_score and cutoff_score_max fields of the new BlastHitSavingParameters object.

12.5.2 BlastHitSavingParametersUpdate

The BlastHitSavingParametersUpdate function is declared with the following prototype in the blast_parameters.h.

```
Int2
BlastHitSavingParametersUpdate(
    EBlastProgramType program_number,
    BlastScoreBlk* sbp, BlastQueryInfo* query_info,
    Int4 avg_subject_length,
    BlastHitSavingParameters* parameters);
```

This function sets the `cutoff_score` and `max_cutoff_score` fields of an object of type `BlastHitSavingParameters`. If `parameters->options->cutoff_score` is set to a value greater than zero, then both `cutoff_score` and `max_cutoff_score` are set to the product of that value with `sbp->scale_factor`. Otherwise the following procedure is used to set the fields.

`BlastHitSavingParametersUpdate` first selects a Karlin-block by invoking the routine `s_BlastFindValidKarlinBlk` on the `sbp->kbp_gap` array for a gapped search or the `sbp->kbp` array for an ungapped search. It then invokes `BLAST_Cutoffs` on `options->expect_value` to obtain a value for the field `max_cutoff_score`. The effective search space used is the search space of the first context. The gap decay rate is not used.

For ungapped searches or if sum statistics are not being used, `cutoff_score` is set to `max_cutoff_score`. Otherwise `BLAST_Cutoffs` is invoked again with a E-value of 1.0 and a different search space, which is given by the following expression.

```
MIN(avg_qlen, avg_subject_length) * (Int8)avg_subject_length
```

The datatype `Int8` is a signed integer type with at least 64 bits of precision; the size of the search space is frequently large enough to overflow 32 bit integers. The value `avg_subject_length` is a function argument and `avg_qlen` is the average length of all query contexts. The gap decay rate, which is taken from the value

`params->link_hsp_params->gap_decay_rate`,

is used in the call to `BLAST_Cutoffs`. The value of `cutoff_score` is the lesser of the result from `BLAST_Cutoffs` and the field `cutoff_score_max`.

Finally the computed values of `cutoff_score_max` and `cutoff_score` are rescaled by multiplication by `sbp->scale_factor`.

12.6 CalculateLinkHSPCutoffs

The `CalculateLinkHSPCutoffs` routine is declared in the `blast_parameters.h` file with the following prototype.

```
void
CalculateLinkHSPCutoffs(
    EBlastProgramType program,
    BlastQueryInfo* query_info,
    BlastScoreBlk* sbp,
    BlastLinkHSPParameters* link_hsp_params,
    const BlastInitialWordParameters* word_params,
    Int8 db_length, Int4 subject_length)
```

The purpose of this routine it to calculate the weights used by the deprecated `s_BlastEvenGapLinkHSPs` routine to generate linked sets. These weights are also effectively cutoffs, since no HSP that has score smaller than the given weight is added to a linked set.

The `s_BlastEvenGapLinkHSPs` routine can be invoked for gapped searches, but the routine always uses an ungapped Karlin block, specifically

`sbp->kbp[query_info->first_context]`,

to calculate the weights.

Appendix

A Blast_KarlinBlkUngappedCalc details

This section is a detailed discussion of the `Blast_KarlinBlkUngappedCalc` routine, described initially in section 3.1.

A.1 Developer comments

Version 1.0	February 2, 1990
Version 1.2	July 6, 1990
Program by:	Stephen Altschul
Address:	National Center for Biotechnology Information National Library of Medicine National Institutes of Health Bethesda, MD 20894
Internet:	altschul@ncbi.nlm.nih.gov

See: Karlin, S. and Altschul, S.F. “Methods for Assessing the Statistical Significance of Molecular Sequence Features by Using General Scoring Schemes,” Proc. Natl. Acad. Sci. USA 87 (1990), 2264-2268.

Computes the parameters λ and K for use in calculating the statistical significance of high-scoring segments or subalignments.

The scoring scheme must be integer valued. A positive score must be possible, but the expected (mean) score must be negative.

A program that calls this routine must provide the value of the lowest possible score, the value of the greatest possible score, and a pointer to an array of probabilities for the occurrence of all scores between these two extreme scores. For example, if score -2 occurs with probability 0.7, score 0 occurs with probability 0.1, and score 3 occurs with probability 0.2, then the subroutine must be called with `low` $= -2$, `high` $= 3$, and `pr` pointing to the array of values $\{0.7, 0.0, 0.1, 0.0, 0.0, 0.2\}$. The calling program must also provide pointers to λ and K; the subroutine will then calculate the values of these two parameters. In this example, $\lambda = 0.330$ and $K = 0.154$.

The parameters λ and K can be used as follows. Suppose we are given a length N random sequence of independent letters. Associated with each letter is a score, and the probabilities of the letters determine the probability for each score. Let S be the aggregate score of the highest scoring contiguous segment of this sequence. Then if N is sufficiently large (greater than 100), the following bound on the probability that S is greater than or equal to x applies:

$$P(S \geq x) \leq 1 - \exp[-KNe^{-\lambda x}].$$

In other words, the p-value for this segment can be written as

$$1 - \exp[-KNe^{-\lambda S}].$$

This formula can be applied to pairwise sequence comparison by assigning scores to pairs of letters (e.g. amino acids), and by replacing N in the formula with $N \times M$, where N and M are the lengths of the two sequences being compared.

In addition, letting $y = KNe^{-\lambda S}$, the p-value for finding m distinct segments all with score $\geq S$ is given by:

$$1 - [1 + y + y^2/2! + ... + y^{m-1}/(m-1)!]e^{-y}$$

Notice that for $m = 1$ this formula reduces to $1 - e^{-y}$, which is the same as the previous formula.

A.2 Error conditions

We use the notation of section 3.1.

If any routine invoked by `Blast_KarlinBlkUngappedCalc` fails, then the values of λ, K and H are set to -1; `logK` is set to a large positive number; and the `Blast_KarlinBlkUngappedCalc` routine returns 1 to indicate an error.

Let ℓ and u be the lowest and highest scores that occur with nonzero probability. The `Blast_KarlinLambdaNR` routine and the `BlastKarlinLtoH` routine both verify

- that the expected value of the scores is negative;

- that $\ell < 0 < u$;
- that $\ell \geq$ BLAST_SCORE_MIN and $u \leq$ BLAST_SCORE_MAX; and
- that $u - \ell \leq$ BLAST_SCORE_RANGE_MAX.

If any of the preceding conditions is not met, the routines fail and immediately return -1.

The BlastKarlinLHtoK routine verifies that the average score and the parameters λ and H are all nonnegative. It returns -1 otherwise. The routine does not, however, validate ℓ and u. The BlastKarlinLHtoK routine tries to allocate a work array and returns a -1 if the allocation fails.

A.3 Numerical comments

A.3.1 Evaluation of series via Horner's rule

We use the notation defined section 3.1, particularly the definition of $P_j(i)$ given by equation (3) and the definitions of ℓ and u given in equation (4).

The Blast_KarlinLambdaNR routine, the BlastKarlinLtoH routine and the BlastKarlinLHtoK routine all compute sums of the form $s = \sum_{i=\ell}^{u} c_i e^{i\lambda}$ for some set of constants $\{c_i\}$. The sums are computed using Horner's rule, i.e.

$$\begin{aligned} t_0 &= c_\ell \\ t_i &= c_{\ell+i} + e^{-\lambda} t_{i-1}, \end{aligned}$$

where $\hat{s} = t_{u-\ell}$. Then $s = \hat{s}/e^{-\lambda u}$, which can be evaluated as

$$s = \exp(\lambda u + \ln \hat{s}) \tag{32}$$

if underflow of $e^{-\lambda u}$ is an issue. It is crucial that these sums not be computed naively, by computing $e^{(\ell-1)\lambda}$ and powers of this quantity. In practice, these sums may be need to be computed for values of λ that cause $e^{(\ell-1)\lambda}$ to underflow.

Horner's rule is the preferred method of evaluating polynomials and has been proved to be accurate. (See e.g. Higham [9].) There might be some concern, however, that we have introduced overflow conditions into the computation. That is not the case. Because $e^{-\lambda} < 1$,

$$|t_k| \leq k \max_{\ell \leq i \leq k} (|c_i|). \tag{33}$$

Thus, for reasonable values of the coefficients, Horner's rule cannot overflow. Underflow is still possible in any of the products $e^{-\lambda} t_{i-1}$. However, if the values of c_i are reasonably scaled (i.e. not all close to the smallest positive or largest negative double precision value), underflow does not affect the accuracy of the computation. Horner's rule is simpler to implement than a method the computes powers of $e^{-\lambda}$ explicitly, better protected against overflow and underflow and almost always more efficient. The possible exception is the case in which $e^{-\lambda u}$ underflows but $e^{(\ell-1)\lambda}$ does not, in which case one must use (32) to calculate s from $\hat{s}$.

A.3.2 Newton's method for computing λ^*.

Recall that the Karlin-Altschul parameter λ^* is the unique, positive root of the function

$$\phi(\lambda) = -1 + \sum_{i=\ell}^{u} P_1(i) e^{i\lambda},$$

introduced in equation (6). It is possible to compute λ^* by applying a safeguarded Newton iteration to the function $\phi(\lambda)$, but a better approach is to apply Newton's method to the polynomial

$$q(x) = -x^u + \sum_{k=0}^{u-\ell} P_1(u-k) x^k.$$

By definition, $\phi(\lambda) = e^{u\lambda} \times q(e^{-\lambda})$. Thus $\lambda^* > 0$ is a root of $\phi(\lambda)$ if and only if $x^* = e^{-\lambda^*}$ is a root of $q(x)$. Therefore we may solve for λ^* by applying Newton's method to $q(x)$.

There are several advantages to applying Newton's method to $q(x)$ rather than $\phi(\lambda)$. The polynomial $q(x)$ is faster to evaluate. As is discussed below, it is straightforward to define a safeguarded Newton iteration on $q(x)$ because the root of $q(x)$ must lie in the interval $(0, 1)$. Furthermore, for $x \in [0, 1]$

$$|q(x)| \leq x^u + \sum_{k=0}^{u-\ell} P_1(u-k) x^k \leq 1 + \sum_{k=0}^{u-\ell} P_1(u-k) \leq 2.$$

Similar arguments can be used to show that the evaluation of $q(x)$ and its derivative by Horner's rule does not overflow. Overflow or underflow may occur in practice in the computation of $\phi(\lambda)$.

In addition to generating a sequence of iterates, a safeguarded Newton iterations generates a sequence of intervals $\{(a_k, b_k)\}$, where $(a_{k+1}, b_{k+1}) \subset (a_k, b_k)$ for all iteration indices k. These intervals, known as intervals of uncertainty, each contain a zero of the polynomial $q(x)$. The polynomial has exactly two nonnegative roots, one at $x = 1$ and the other at $x = e^{-\lambda^*}$. Because $0 < e^{-\lambda^*} < 1$, the interval $(0, 1)$ may serve as an initial interval of uncertainty for a safeguarded Newton iteration.

Because $\phi'(0) < 0$, it follows that $q'(1) > 0$. Thus it is clear that $q(x) < 0$ in the interval $(x^*, 1)$, and $q(x) > 0$ in the interval $[0, x^*)$. Then for any point x for which $q'(x) \leq 0$, it follows that the Newton iterate $x - q(x)/q'(x)$ is further from x^* than x is. Thus whenever $q'(x) \leq 0$, it makes sense to bisect the interval of uncertainty.

In the following, safeguarded algorithm, we implement the usual rule that a bisection step is taken if the proposed Newton iterate lies outside the current interval of uncertainty. We also require that, for some $\gamma \in (0, 1)$,

$$|q(x_k)| \leq \gamma |q(x_{k-1})| \tag{34}$$

for every iterate x_k that is derived from x_{k-1} by a Newton step. A typical value of γ is .9. If x_k does not meet condition (34), then x_{k+1} is chosen by bisection.

Thus it is clear that the iteration must converge to a zero of $q(x)$ in the interval $[0, 1]$. We also implement the rule by which take a bisection step whenever $q'(x_k) \geq 0$. Because there is an interval about $x = 1$ for which $q'(x) \geq 0$, the iteration cannot converge to $x = 1$, and thus converges to x^*. ALGORITHM A.1.

SAFEGUARDED NEWTON ITERATION FOR $q(x)$

```
Let $\tau > 0$ be a solution tolerance.
Let $k_{\max}$ and $x_0 \in (0, 1)$ be given.
$\lambda \leftarrow \lambda_0$, $a \leftarrow 0$ and $b \leftarrow 1$.
isNewton ← **false**
**for** $k = 1, \ldots, k_{\max}$ **do**
    wasNewton ← isNewton
    isNewton ← **false**
    Let $q_k = q(x)$ and $q'_k = q'(x)$.
    **if** $q_k = 0$ **then stop**
    **if** $q_k > 0$ **then** $a \leftarrow x$; **else** $b \leftarrow x$; **end if**
    **if** $b - a < 2a(1 - b)\tau$ **then** $x \leftarrow (b + a)/2$; **stop**; **end if**
    **if** (wasNewton **and** $|q_k| > \gamma|q_{k-1}|$) **or** $q'_k \geq 0$ **then**
        $x \leftarrow (a + b)/2$
    **else**
        $y \leftarrow x - q_k/q'_k$
        **if** $y \leq a$ **or** $y \geq b$ **then**
            $y \leftarrow (b + a)/2$
        **else**
            isNewton ← **true**
            $x_- \leftarrow x$; $x \leftarrow y$
            **if** $|(x - x_-)| \leq x(1 - x)\tau$ **then stop**; **end if**
        **end if**
    **end if**
**end do**
```

The convergence tests of Algorithm A.1 merit some discussion. Suppose we wish to compute λ^* with relative accuracy τ. In other words, we wish to find a computed value λ such that

$$-\tau \leq \frac{\lambda - \lambda^*}{\lambda} \leq \tau. \tag{35}$$

But then $e^{-\lambda\tau} \leq e^{\lambda - \lambda^*} \leq e^{\lambda\tau}$. If $x = e^{-\lambda}$ and $x^* = e^{\lambda^*}$, then

$$x^\tau - 1 \leq \frac{x^* - x}{x} \leq x^{-\tau} - 1. \tag{36}$$

We may expand $x^\tau - 1$ in a Taylor series about $x = 1$ to find that for $x \in (0, 1]$,

$$x^\tau - 1 = \tau(x - 1) + \frac{1}{2}\tau(\tau - 1)\xi^{\tau-1}(x - 1)^2$$

for some $\xi \in [x, 1]$. But $\tau(\tau - 1) < 0$ for $0 < \tau < 1$, and thus the second term in the expansion is negative. It follows that $x^\tau - 1 < -\tau(1 - x)$ for $x \in (0, 1]$.

One may use a similar argument to show that $\tau(1-x) < x^{-\tau} - 1$. Therefore the termination criterion

$$|x - x^*| \le x(1-x)\tau \tag{37}$$

is at least as stringent as (36).

If x is close to one, the criterion (37) can be very stringent. However, if x is close to one, $\lambda = -\ln x$ is close to zero. It is not difficult to see that if x is close to one, then the quantity $(x^* - x)/x$ closely approximates $\lambda - \lambda^*$ and the quantity $\tau(1-x)$ closely approximates $\lambda\tau$. Thus condition (37) is no more difficult to satisfy than (35).

B The calculation of length adjustments

This section discusses the iteration used by `BLAST_ComputeLengthAdjustment` to compute the length adjustments that are used when calculating E-values. For a discussion of length adjustments and how these relate to the effective lengths of the query and database sequences, see section 5.

In this section, we use the notation of section 5, except that we refer to the actual lengths of the query sequence and the database as m and n, rather than as m_a and n_a. We do not refer to the effective lengths of the query and database in this section, so there should be no cause for confusion.

B.1 A fixed-point iteration

Let us define the function

$$f(\ell) = \bar{\alpha} \ln \{K(m-\ell)(n-N\ell)\} + \beta, \tag{38}$$

where $\beta = 0$ for an ungapped alignment, and

$$\bar{\alpha} = \begin{cases} \alpha/\lambda & \text{for a gapped alignment; and} \\ 1/H & \text{for an ungapped alignment.} \end{cases}$$

Let us define the interval $\Omega = [0, \min(m, n/N)]$. We seek a fixed point ℓ^* of $f(\ell)$ in Ω. In other words, we seek a point $\ell^* \in \Omega$ at which $\ell^* = f(\ell^*)$.

The derivative of $f(\ell)$ is

$$f'(\ell) = -\bar{\alpha}\left(\frac{1}{m-\ell} + \frac{N}{n-N\ell}\right). \tag{39}$$

Thus $f(\ell)$ is strictly decreasing in Ω. Consider the function $h(\ell) = f(\ell) - \ell$, which is defined so that $f(\ell) = \ell$ if and only if $h(\ell) = 0$. The function $h(\ell)$ is also strictly decreasing in Ω. Due to the the logarithm in the definition of $h(\ell)$, the function $h(\ell)$ is positively infinite in the limit as ℓ approaches negative infinity, and

$$\lim_{\ell \to \min(m, n/N)^-} h(\ell) = -\infty.$$

Therefore, there must be a value $\ell^* < \min(m, n/N)$ for which $h(\ell^*) = 0$. Furthermore, $h(\ell)$ is strictly decreasing, so ℓ^* is unique. If $h(0) = f(0) > 0$, then $\ell^* \in \Omega$. Conversely, if $h(0) = f(0) < 0$, there can neither be a zero of $h(\ell)$ nor a fixed point of $f(\ell)$ in Ω.

Because $f(\ell)$ is strictly decreasing for $\ell \in \Omega$, one may easily determine whether a given value of ℓ is less than or greater than ℓ^*.

Proposition B.1.1. *Suppose $\ell \in \Omega$. Then*

- *$\ell^* < f(\ell)$ if and only if $\ell < \ell^*$; and*
- *$\ell^* > f(\ell)$ if and only if $\ell > \ell^*$.*

Proof. If $\ell < \ell^*$, then because $f(\ell)$ is strictly decreasing, $f(\ell^*) < f(\ell)$. But $\ell^* = f(\ell^*)$, and so $\ell^* < f(\ell)$. Similarly, if $\ell > \ell^*$ then $\ell^* > f(\ell)$. All possibilities are exhausted, so the result follows. □

B.2 A safeguarded fixed-point iteration

The following algorithm is used to define the `BLAST_ComputeLengthAdjustment` routine, described in section 5.1.

If $Kmn \geq \max(m, n)$, we impose the restriction that the computed value $\bar{\ell}$ satisfies

$$K(m - \bar{\ell})(n - N\bar{\ell}) \geq \max(m, n). \tag{40}$$

Suppose $Kmn \geq \max(m, n)$ and let $\ell_{\max}$ be the smaller of the two necessarily nonnegative roots of

$$K(m - \ell)(n - N\ell) = \max(m, n).$$

Then $\bar{\ell} \in \Omega$ satisfies inequality (40) if and only if $\bar{\ell} \in [0, \ell_{\max}]$. This property is a immediate consequence of the fact that the left hand side of inequality (40) is a convex quadratic function with a negative slope at zero. In the exceptional case in which $Kmn < \max(m, n)$, no $\ell \in \Omega$ satisfies condition (40), and we take both $\ell_{\max}$ and $\bar{\ell}$ to be zero.

The algorithm for computing $\bar{\ell}$ employs an iteration with iteration index i and variables ℓ_i and y_i. For $i = 1, \ldots,$ maxits, the value ℓ_{i-1} is restricted to lie in $[0, \ell_{\max}]$, and y_i is chosen by the rule $y_i = f(\ell_{i-1})$. Typically, one will choose ℓ_i by the rule $\ell_i = y_i$. However, there are circumstances in which ℓ_i can not be chosen by this rule; in particular, ℓ_i must lie within $[0, \ell_{\max}]$.

Let the sequence of intervals $\{[a_i, b_i]\}$ be ordered by inclusion, and let $a_0 = 0$ and $b_0 = \ell_{\max}$. For any iteration i, if $\ell_i \in [a_i, b_i]$, then $\ell_i \in [0, \ell_{\max}]$. The rule

$$\ell_i = \begin{cases} y_i & \text{if } a_i \leq y_i \leq b_i\text{; and} \\ (a_i + b_i)/2 & \text{otherwise} \end{cases}$$

necessarily chooses a value of $\ell_i \in [a_i, b_i]$. For $i \geq 1$, we choose the interval $[a_i, b_i]$ by the following rule.

$$[a_i, b_i] = \begin{cases} [\ell_{i-1}, b_{i-1}] & \text{whenever } \ell_{i-1} \leq y_i\text{; and} \\ [a_{i-1}, \ell_{i-1}] & \text{otherwise.} \end{cases} \tag{41}$$

Because $\ell_{i-1} \in [a_{i-1}, b_{i-1}]$ and because rule (41) chooses one of the two endpoints of $[a_i, b_i]$ to be ℓ_i, it follows that $[a_i, b_i] \supset [a_{i+1}, b_{i+1}]$ for every i.

Proposition B.2.1. *If $\ell^* \in [0, \ell_{\max}]$ and $[a_i, b_i]$ is chosen by rule (41) for $i \geq 1$, then $\ell^* \in [a_i, b_i]$ for every i.*

Proof. By assumption $\ell^* \in [a_0, b_0]$. Suppose $\ell^* \in [a_i, b_i]$. If $\ell_i \leq y_i$, then from Proposition B.1.1 and rule (41) it follows that $\ell^* \in [a_{i+1}, b_{i+1}] = [\ell_i, b_i]$. Similarly, if $\ell_i > y_i$, then $\ell_i > \ell_*$ and $\ell_* \in [a_{i+1}, b_{i+1}] = [a_i, \ell_i]$. □

Based on these observations, we define the following algorithm that seeks ℓ^*.

```
ALGORITHM B.1. COMPUTE EFFECTIVE LENGTHS
Let $m$, $n$, $N$, $\bar{\alpha}$, $\beta$, $K$, maxits and $\ell_{\max}$ be given.
$a_0 \leftarrow 0$; $\ell_0 \leftarrow 0$; $\bar{\ell} \leftarrow 0$; $b_0 \leftarrow \ell_{\max}$
converged ← **false**
**for** $i = 1, \ldots,$ maxits **do**
    $y_i \leftarrow \bar{\alpha} \ln[K(m - \ell_{i-1})(n - N\ell_{i-1})] + \beta$
    **if** $\ell_{i-1} \leq y_i$ **then**
        $a_i \leftarrow \ell_{i-1}$; $\bar{\ell} \leftarrow \ell_{i-1}$; $b_i \leftarrow b_{i-1}$
        **if** $y_i - \bar{\ell} \leq 1$ **then** converged ← **true**; **stop**
    **else**
        $a_i \leftarrow a_{i-1}$; $b_i \leftarrow \ell_{i-1}$
    **end if**
    **if** $a_i \leq y_i \leq b_i$ **then**
        $\ell_i \leftarrow y_i$
    **else**
        **if** $i = 1$ **then** $\ell_i \leftarrow \ell_{\max}$; **else** $\ell_i \leftarrow (a_i + b_i)/2$; **end if**
    **end if**
**end do**
```

From the real-valued quantity $\bar{\ell}$, one must obtain an integer value for the length adjustment. Proposition B.3.2 states that if Algorithm B.1 converges, then $\ell^* \in [\bar{\ell}, \bar{\ell}+1]$. Thus when the iteration converges, either floor(ℓ^*) = floor$(\bar{\ell})$ or floor(ℓ^*) = floor$(\bar{\ell}) + 1$. One must compute $f(\text{floor}(\bar{\ell}) + 1)$ and apply Proposition B.1.1 to determine which relationship holds. Whenever possible, one uses floor(ℓ^*) as the integer-valued length adjustment. On the rare occasions in which the iteration does not converge or in which one floor$(\ell^*) > \ell_{\max}$, one uses floor$(\bar{\ell})$ as the length adjustment.

B.3 Convergence properties

In this section, we state and prove convergence results for Algorithm B.1.

Proposition B.3.1. *If the true fixed point ℓ^* is nonnegative, then $\bar{\ell} \leq \ell^*$ at the final iteration of Algorithm B.1.*

Proof. The initial value of $\bar{\ell}$ is zero. Therefore if ℓ^* is nonnegative, then $\bar{\ell} \leq \ell^*$ at the first iteration. The value of $\bar{\ell}$ is updated only if $\ell_{i-1} \leq y_i$, which, by Proposition B.1.1, occurs only when $\ell_{i-1} \leq \ell^*$. □

Proposition B.3.2. *If Algorithm B.1 converges, then $\ell^* \in [\bar{\ell}, \bar{\ell} + 1]$.*

Proof. Algorithm B.1 can converge only if $\ell_{i-1} \leq y_i$ and $y_i - \ell_{i-1} \leq 1$ for some index i. By Proposition B.1.1, if $\ell_{i-1} \leq y_i$, then $\ell_{i-1} \leq \ell^*$. Furthermore, $\ell^* \leq y_i$ because $f(\ell)$ is strictly decreasing. Thus

$$\ell_{i-1} \leq \ell^* \leq y_i \leq \ell_{i-1} + 1.$$

The updating rules of Algorithm B.1 choose $\bar{\ell} = \ell_{i-1}$ for the value of i at which convergence occurs. The result follows immediately. □

Proposition B.3.3. *If ℓ^* is negative, then the value of $\bar{\ell}$ is zero at the final iteration of Algorithm B.1.*

Proof. Algorithm B.1 restricts ℓ_i to be nonnegative for every i. Thus, if ℓ^* is negative, then $\ell^* < \ell_{i-1}$ for every $i \geq 1$. From Proposition B.1.1, it follows that $y_i < \ell_{i-1}$ for every $i \geq 1$. But the value of $\bar{\ell}$ is updated only when $y_i \geq \ell_{i-1}$. Thus the value of $\bar{\ell}$ at the final iteration of Algorithm B.1 equals its initial value, which is zero. □

Proposition B.3.4. *If $\ell^* > \ell_{\max}$, then the value of $\bar{\ell}$ is $\ell_{\max}$ at the final iteration of Algorithm B.1.*

Proof. Because $\ell_0 = 0 \leq \ell_{\max} < \ell^*$, it follows from Proposition B.1.1 that both $\ell_0 < y_1$ and $\ell_{\max} < y_1$. Thus Algorithm B.1 chooses $b_1 = b_0 = \ell_{\max}$. Furthermore $b_1 < y_1$, and so the algorithm chooses $\ell_1 = \ell_{\max}$. Therefore $\ell_1 < \ell^*$, and so $\ell_1 < y_2$. Inspection of Algorithm B.1 then shows that

- $b_2 = b_1 = \ell_{\max}$;
- the value of $\bar{\ell}$ is updated to equal $\ell_1 = \ell_{\max}$; and
- the iteration terminates with $i = 2$.

Thus the value of $\bar{\ell}$ at the final iteration of Algorithm B.1 is $\ell_{\max}$. □

It is possible, but unlikely, that $\ell^* \in [0, \ell_{\max}]$ but that the iteration does not converge. In this case, by Proposition B.3.1, the algorithm produces an underestimate of the fixed point. By the well-known fixed point theorem (see e.g. Atkinson [7], Theorem 2.6), if there is a point ℓ^* for which $\ell^* = f(\ell^*)$ and for which $|f'(\ell^*)| = \gamma < 1$, then there is an interval about ℓ^* for which the iteration $\ell_{k+1} = f(\ell_k)$ converges to ℓ^*. The rate of convergence is at worst linear. If $\gamma \neq 0$, then the iteration converges linearly at an asymptotic rate of γ. Inspection of equation (39) suggests that $|f'(\ell^*)|$ would be considerably less than one for typical values of ℓ^*, m, n, N, H, α and λ; and thus that convergence would be quick.

References

[1] S.F. Altschul, W. Gish, W. Miller, E.W. Myers, and D.J. Lipman. Basic local alignment search tool. *J. Mol. Biol.*, 215:403–410, 1990.

[2] Stephen F. Altschul. Evaluating the statistical significance of multiple distinct local alignments. In Suhai, editor, *Theoretical and Computational Methods in Genome Research*, pages 1–14. Plenum Press, New York, 1997.

[3] Stephen F. Altschul. Generalized affine gap costs for protein sequence alignment. *Proteins*, 32:88–96, 1998.

[4] Stephen F. Altschul, Ralf Bundschuh, Rolf Olsen, and Terence Hwa. The estimation of statistical parameters for local alignment score distributions. *Nucleic Acids Res.*, 29:351–361, 2001.

[5] Stephen F. Altschul and W. Gish. Local alignment statistics. *Meth. Enzymol.*, 266:460–480, 1996.

[6] Stephen F. Altschul, Thomas L. Madden, Alejandro A. Schäffer, Jinghui Zhang, Zheng Zhang, Webb Miller, and David J. Lipman. Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. *Nucleic Acids Res.*, 25:3389–3402, 1997.

[7] Kendal E. Atkinson. *An Introduction to Numerical Analysis.* Wiley, 1989.

[8] A. Dembo, S. Karlin, and O. Zeitouni. Limit distribution of maximal non-aligned two-sequence segmental score. *Ann. Prob.*, 22:2022–2039, 1994.

[9] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms.* SIAM, 1996.

[10] Samuel Karlin and Stephen F. Altschul. Methods for assessing the statistical significance of molecular sequence features by using general scoring schemes. *Proc. Nat. Acad. Sci. USA*, 87:2264–2268, 1990.

[11] Samuel Karlin and Stephen F. Altschul. Applications and statistics for multiple high-scoring segments in molecular sequences. *Proc. Nat. Acad. Sci. USA*, 90:5873–5877, 1993.

[12] Arthur B. Robinson and Laurelee R. Robinson. Distribution of glutamine and asparagine residues and their near neighbors in peptides and proteins. *Proc. Nat. Acad. Sci. USA*, 88:8880–8884, 1991.

[13] Zheng Zhang, Alejandro A. Schäffer, Webb Miller, Thomas L. Madden, David J. Lipman, Eugene V. Koonin, and Stephen F. Altschul. Protein sequence similarity searches using patterns as seeds. *Nucleic Acids Res.*, 26:3986–3990, 1998.