

# fjoin: Simple and Efficient Computation of Feature Overlaps

JOEL E. RICHARDSON<sup>1</sup>

## ABSTRACT

Sets of biological features with genome coordinates (e.g., genes and promoters) are a particularly common form of data in bioinformatics today. Accordingly, an increasingly important processing step involves comparing coordinates from large sets of features to find overlapping feature pairs. This paper presents **fjoin**, an efficient, robust, and simple algorithm for finding these pairs, and a downloadable implementation. For typical bioinformatics feature sets, fjoin requires  $O(n \log(n))$  time ( $O(n)$  if the inputs are sorted) and uses  $O(I)$  space. The reference implementation is a stand-alone Python program; it implements the basic algorithm and a number of useful extensions, which are also discussed in this paper.

## 1. INTRODUCTION

In biology today and for the foreseeable future, data comprising features with genome coordinates are both very common and very central. Major offerings of prominent data providers are of this form (e.g., SNPs from dbSNP (dbSNP 2006) or gene models from Ensembl (Ensembl 2006)), as are the output of the most commonly used tools (e.g., BLAT (Kent 2002), SIM4 (Florea, *et al.* 1998), FASTA (Pearson 2000)). There are defined standard file formats for representing sets of features (e.g., GFF3 (GFF3 2004), PSL (PSL 2006)). Finally, the main bioinformatics visualization paradigm, the genome browser, is devoted to viewing and exploring large sets of features (GBrowse (GBrowse 2006), UCSC Genome Browser (UCSC 2006)).

Given the prevalence of feature-set data and tools, it is not surprising that a common problem is to compare such sets, looking for coordinate-based overlap or (more generally) proximity between the members. In the Mouse Genome Informatics program (MGI 2006, Eppig, *et al.* 2005, Hill, *et al.* 2004, Krupke, *et al.* 2005), we routinely perform this kind of analysis. One example is processing new mouse transcript sequences from Genbank to determine if they represent known or novel genes. As one part of this analysis, we perform a high-stringency genome alignment to assign coordinates to the transcripts and then analyze the overlaps with known genes.

The trivial solution to finding all overlapping feature pairs is a nested loop:

```
def nestedLoops(X, Y):  
    // Compares every x in X to every y in Y.  
    foreach(x in X):  
        foreach(y in Y):  
            if( overlaps(x,y) ):  
                output(x,y)
```

---

<sup>1</sup> Mouse Genome Informatics, The Jackson Laboratory, Bar Harbor, ME 04609.

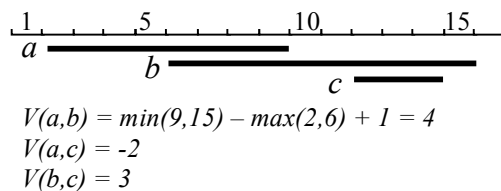
For small data sets, this algorithm is fine. However, it quickly becomes impractical for large data sets because of the quadratic increase in the number of iterations through the inner loop. For example, in collaboration with the Mouse Genome Sequencing and Analysis Consortium (MGSC 2002), a new project at MGI is to compare the NCBI and Ensembl gene models to help generate a nonredundant set of consensus genes for the mouse. One of the analysis steps compares all exons from the two sources for coordinate based overlaps. For mouse build 36, these total about 292,000 exons from NCBI and 278,000 from Ensembl. Using nested loops, we must perform  $> 8 \times 10^{10}$  tests for overlap, a process that takes days to run. In contrast, fjoin compares the NCBI and Ensembl gene models in under 5 minutes on a Mac PowerBook G4 laptop.

The problem of finding overlapping genome features is a special case of finding overlapping regions in  $n$ -dimensional space. There is a long history of algorithms research for spatial/geometric data (Samet 1990), and there are a number of relevant data structures that could be applied to our problem, e.g., segment trees (Bentley 1977), interval trees (Edelsbrunner 1980), and R-trees (Guttman 1984). Today, several database systems such as Postgres and Oracle provide R-tree indexing as an option, and their use for answering feature-overlaps queries has been previously reported (Lapp, *et al.*, 2003). In that report, using R-trees in GBrowse's backing store achieved a 4-5x speedup in queries for features within a viewing range.

The motivation for our work was the need for an easy-to-use command-line tool, suitable for use in file-processing applications and pipelines. While using spatial indexes is a reasonable approach, we did not want to require installation of a database system as a prerequisite for using the tool. The fjoin algorithm was developed while considering alternatives. One contribution of this paper is in achieving  $O(n \log(n))$  performance with a remarkably simple algorithm, requiring no sophisticated index structures or backing database system. fjoin has been implemented as a command-line tool suitable for use in file processing pipelines. The program comprises a few hundred lines of Python code (Python 2006).

## 2. THE ALGORITHM

**Definitions.** For any feature,  $x$ , we'll refer to its start and end coordinates as  $x.start$  and  $x.end$ , respectively. We assume all coordinates are integer base positions and that  $x.start \leq x.end$  for all features. The *overlap*,  $V$ , between two features,  $x$  and  $y$ , is defined as:  $V(x,y) = \min(x.end, y.end) - \max(x.start, y.start) + 1$ . Figure 1 shows three features,  $a$ ,  $b$ , and  $c$ , and computes their overlaps. Note that disjoint features such as  $a$  and  $c$  have a negative overlap, whose absolute value is the distance between them.



**Figure 1.** Three features,  $a$ ,  $b$ , and  $c$ , and their overlaps.

**Outline.** Fjoin is a specialized sort-merge join (Korth and Silberschatz, 1986). We first sort each input by start coordinate, if not already done. We then perform a “merge pass” over the sorted lists; that is, we move along them, more or less in parallel, and check for overlapping pairs as we go. Since robust and efficient sorting tools are readily available<sup>2</sup>, our focus is on the merge pass. In the following, we refer to the sorted feature sets as “streams”.

**The merge pass.** The merge pass is a sliding window algorithm. For each stream, we maintain a *window*, which is a sub-sequence of the features seen so far in that stream. We also maintain a *current feature*, which is the most recently returned feature from that stream; the *current position* of a stream is the start position of its current feature. We also assume the existence of a special *sentinel feature*, whose start coordinate is infinite<sup>3</sup>; the sentinel is returned after the last real feature has been returned by a stream. In each round, we choose the “lagging” stream, i.e., the one with the smaller current position, and *scan* (i.e., compare) its current feature with the features in the opposite stream’s window. We report any overlaps and update the windows as described below. This procedure is repeated until we reach the end of both streams.

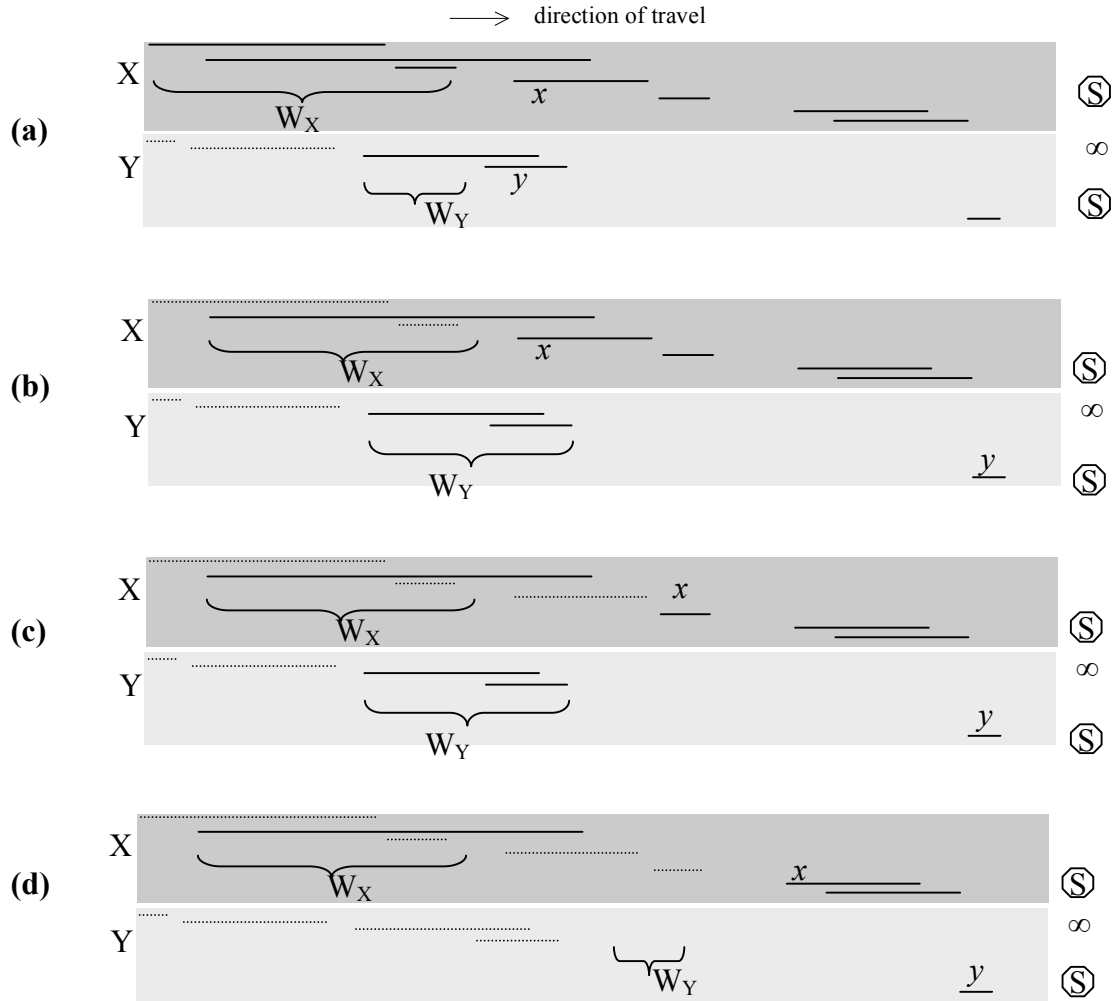
**Managing windows.** The strategy for maintaining windows derives from the following simple observation about sorted streams. If feature  $x$  from stream  $X$  is to the left of feature  $y$  from stream  $Y$  (i.e., if  $x.end < y.start$ ), then  $x$  does not overlap  $y$ , nor any  $Y$  feature following  $y$ , and vice versa. During the merge pass, we will add a feature to its stream’s window only if there is a possibility of overlap with remaining features in the opposite stream, and we will remove a feature from its window when such overlaps are no longer possible.

Consider Figure 2(a), in which we scan feature  $y$  against window  $W_X$ , and consider the first (leftmost) feature in  $W_X$ . Clearly, it does not overlap  $y$ ; moreover, because  $Y$  is sorted, it cannot overlap any remaining  $y$ . We can therefore remove it from  $W_X$ . We remove the third feature for the same reason. The second feature overlaps  $y$ , and we report the overlap. However, because we cannot rule out the possibility of further overlaps, we leave this feature in  $W_X$ . Finally, because  $y$  is not to the left of  $X$ ’s current position, more overlaps are possible, so we add  $y$  to  $W_Y$ . Figure 2b shows the result.

---

<sup>2</sup> Indeed, if sorting is required, the reference implementation calls the standard Unix sort.

<sup>3</sup> How the sentinel is represented is implementation dependent. The key point is that the sentinel’s start coordinate compares greater than any real coordinate.



**Figure 2.** The merge pass. Feature streams X and Y are sorted in order of increasing start position. A sentinel (S) marks the end of each stream. Each stream has a current feature ( $x$  and  $y$ ) and a window ( $W_X$ ,  $W_Y$ ). Features already processed are drawn with dotted lines. In each round, the stream with the smaller current position is chosen; its current feature is scanned against the opposite stream's window, and the stream is advanced to the next feature. The panels show snapshots of successive rounds of a merge already under way. **(a)** First, we scan feature  $y$  against window  $W_X$ , which contains the first three features of X. During the scan, the first and third features will be removed from  $W_X$ , and the overlap with the second will be reported.  $y$  will then be added to  $W_Y$ . Finally, stream Y will be advanced to the next feature, ending the round. **(b)** In this round,  $x$  will be scanned against  $W_Y$ . Both features overlap  $x$ , so both remain in  $W_Y$ . However, we omit adding  $x$  to  $W_X$ , since it is to the left of Y's current position, and therefore cannot overlap any remaining features. **(c)** In this round,  $x$  is scanned against  $W_Y$ , leaving  $W_Y$  empty **(d)**. Note that  $y$  is not scanned against  $W_X$  until we reach X's sentinel. As well, note that  $W_Y$  remains empty for the rest of the merge pass.

**Pseudo code.** The following pseudo code states the algorithm more precisely. A number of details are omitted to focus on the heart of the algorithm. In particular, we assume  $X$  and  $Y$  are already sorted, and we defer the generalization to overlap by at least  $k$  to Section 6. Line numbers appear in the left margin for later reference.

```

def fjoin(X,Y):
1   Wx = []
2   Wy = []
3   x = X.next()
4   y = Y.next()
5   while not (x is sentinel and y is sentinel):
6       if x.start <= y.start:
7           scan(x, Wx, y, Wy)
8           x = X.next()
9       else:
10          scan(y, Wy, x, Wx)
11          y = Y.next()

def scan(f, Wf, g, Wg):
12   for g2 in Wg:
13       if leftOf(g2,f):
14           remove g2 from Wg
15       else if overlaps(g2,f):
16           output
17   # end for-loop
18   if not leftOf(f, g):
19       append f to Wf

def overlaps(a, b):
19   v = min(a.end,b.end) - max(a.start,b.start) + 1
20   return (v >= 1)

def leftOf(a, b):
21   return (a.end < b.start)

```

#### 4. CORRECTNESS

In this section, we show that `fjoin` is correct, i.e., it outputs a feature pair  $(x,y)$  if and only if  $x$  and  $y$  overlap.

*Only-if:* Clearly, to reach the output statement in line 16, the features must overlap, because of the condition imposed in line 15.

*If:* We need to show that if  $x$  and  $y$  overlap, then `fjoin` outputs the pair. First, observe that at some point during the while loop (lines 5-11), `scan(f, ...)` is called (either line 7 or 10) for every feature,  $f$ . Now assume there is an overlapping pair  $(x,y)$ . Suppose  $x.start \leq y.start$ . Because of the condition in line 6, `scan(x, ...)` will be called before `scan(y, ...)`. Let that call be: `scan(x,Wx,y',Wy)`, where  $x.start \leq y'.start \leq y.start$ . Because we assume  $x$  overlaps  $y$ ,  $x$  cannot be left of  $y'$  (line 17), so  $x$  will be added to  $W_x$  (line 18). Now,  $x$  remains in  $W_x$  until the call to `scan(y, ...)`. To see this, assume  $x$  is removed (line 14) prior to calling `scan(y, ...)`. Then there must be some  $y''$ , where  $y'.start \leq y''.start \leq y.start$ , where  $x$  is left of  $y''$  (line 13). But then,  $x$  could not overlap  $y$ , a contradiction. Therefore,  $x$  is a member of  $W_x$  when we call `scan(y, ...)`.

During this call, the for loop (lines 12-16) compares  $y$  with every feature in  $W_X$ . The overlap of  $x$  and  $y$  will be discovered and reported (lines 15 and 16). A symmetric argument holds for the case where  $x.start > y.start$ .

Therefore, `fjoin` outputs a pair  $(x,y)$  if and only if  $x$  and  $y$  overlap.

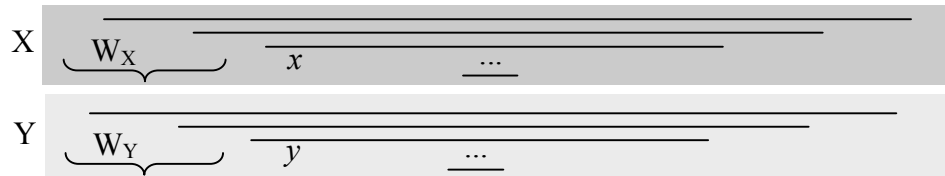
## 5. PERFORMANCE

The theoretical performance of this algorithm is easy to analyze. Sorting is  $n\log(n)$ , so the cost of sorting the inputs is no more than  $|X|\log(|X|) + |Y|\log(|Y|)$ . In practice, we sort the files once and then run many analyses (and some data sets are already sorted), so the actual sorting cost is amortized.

The loop in `fjoin` iterates over all features in both lists, and for each feature, iterates over the opposite stream's window. If  $a_X$  is the average size of  $W_X$  and  $a_Y$  is the average size of  $W_Y$ , then the total time to run `fjoin` is no more than:  $|X|\log(|X|) + |Y|\log(|Y|) + a_Y|X| + a_X|Y|$ . Since  $a_X$  and  $a_Y$  are (typically small) constants, `fjoin` is  $O(n\log(n))$ , and it requires  $O(1)$  space. If the feature sets are already sorted, then `fjoin` is  $O(n)$ .

In comparing exons from the NCBI and Ensembl mouse build 36 gene models, the empirical average window size was 1.5, and we can expect such values to be the rule. Consider the point immediately following a scan, say, of  $y$  against  $W_X$ . At this point,  $W_X$  contains precisely those  $x$ 's that (a) have been seen, and (b) can possibly overlap a  $y$  following the current position. Because of the sort order, we know we have seen all  $x$ 's where  $x.start \leq y.start$ . Therefore,  $W_X$  contains all  $x$ 's where  $x.start \leq y.start$  and  $x.end \geq y.start$ , i.e., the  $x$ 's that cover position  $y.start$ . In other words,  $W_X$  contains the  $x$ 's that are "stacked up" at position  $y.start$ . Thus,  $a_X$  equals the average stacking depth of features in  $X$  at the start coordinates in  $Y$ . While genomic features certainly do overlap one another, they do not stack very deeply, so we can expect small average window sizes.

Note that in the worst case, `fjoin`'s performance can degrade to  $O(n^2)$ . Consider the hypothetical (and highly unlikely) scenario depicted in Figure 3. Because every feature overlaps every other, the windows continue to grow; each scan compares the next feature against everything seen so far in the other stream. Although `fjoin` is  $n^2$  in this instance, it is only because there are  $n^2$  overlaps in the data. *In general, `fjoin`'s time and space requirements vary in proportion to the amount of overlap inherent in the data.*



**Figure 3.** Worst case. In the worst case, `fjoin` is  $O(n^2)$ . Here, every feature overlaps every other. The windows continue to grow, and each new feature is scanned against every feature seen so far in the opposite stream. Note, however, that  $n^2$  iterations are needed simply to enumerate the answer. The amount of work that `fjoin` does varies in proportion to the amount of overlap in the data.

## 6. EXTENSIONS

Simple extensions allow us to generalize fjoin in several useful directions.

Minimum overlap, maximum separation. We have so far considered “overlaps” to mean “at least one base”. However, it is useful to allow users to specify a minimum amount for two features to qualify as “overlapping”. For example, we may wish to limit qualifying pairs to those that overlap by at least 100 bases. The essential change is to the `overlaps` and `leftOf` functions (and the corresponding calls in the body of the `scan` procedure). Here, `k` is the minimum overlap amount specified by the user:

```
def overlaps(a, b, k):
    v = min(a.end, b.end) - max(a.start, b.start) + 1
    return v >= k

def leftOf(a, b, k):
    return (a.end < (b.start + k - 1))
```

We can also find feature pairs that are *separated* by no more than a given amount by passing a negative `k`. For example, to find SNPs within 10 kb of any gene, we’d specify `k = -10000`. Note that for given inputs, the average window size is inversely proportional to `k`. For example, as `k` becomes more negative, it becomes “easier” for two features to overlap, and “harder” to be `leftOf` a given position `p`.

Minimum overlap fraction. A further extension allows the minimum overlap to be specified, not as a fixed number, but as a percentage of the length of the features, e.g., to require an overlap of at least 80%. An additional parameter specifies whether the percentage is relative to the longer or the shorter of each pair. (As special cases, specifying 100% of the longer means perfect matches, while 100% of the shorter means containment.) For this extension, the minimum overlap amount (`k`) is recalculated by the `scan` procedure for each call to `overlaps` (details omitted). For determining when to remove features from windows, we cannot use `k`, since it changes. However, `k` is a positive percentage of a positive length, so we may conservatively pass 0. The revised `scan` procedure follows:

```
def scan(f, Wf, g, Wg):
    for g2 in Wg:
        if leftOf(g2, f, 0):
            remove g2 from Wg
        else if overlaps(f, g2, computeK(f, g2)):
            output
    #
    if not leftOf(f, g, 0):
        append f to Wf
```

Continuous intervals. Another extension handles intervals in continuous coordinate systems, for example, in genetic or RH maps. In a continuous coordinate system, coordinates are represented as floating-point numbers, and the overlap between two intervals is defined as  $V(x,y) = \min(x.end, y.end) - \max(x.start, y.start)$ . We can handle continuous intervals by changing the `overlaps` and `leftOf` functions:

```
def overlaps(a, b, k):
    v = min(a.end, b.end) - max(a.start, b.start)
    return v >= k

def leftOf(a, b, k):
    return (a.end < (b.start + k))
```

The reference implementation is suitably parameterized and handles both discrete and continuous intervals.

Handling chromosome and strand. Actual genomics feature sets almost always include chromosome and strand along with start and end coordinates. Generally, we don't consider features on different chromosomes as overlapping, regardless of their coordinates, and we may or may not accept features from opposite strands, depending on need. Of course, one can always partition the input by chromosome and strand, then run fjoin separately against each subset. The reference implementation handles combined (unpartitioned) data sets. The data must be globally sorted on start position, thus intermingling features from different chromosomes/strands; internally, it maintains a separate window for each chromosome/strand. These are stored in a hash table, so finding the correct window for a feature is a constant-time operation.

Handling  $n$  inputs. Although not currently implemented, fjoin is clearly extensible to more than two inputs. In each round, we would advance the stream with the minimum current position, and compare against the windows of all the other streams.

## 7. IMPLEMENTATION

An implementation of fjoin is available for download from:

<ftp://ftp.informatics.jax.org/pub/fjoin>. This is a stand-alone Python program that implements the basic algorithm as well as most of the extensions described in the previous section. The program requires version 2.3 or later of the Python interpreter. Python is available at <http://www.python.org>.

## 8. SUMMARY

This paper has presented fjoin, a new algorithm for efficiently finding proximity-based pairs of features (e.g., overlapping features), given two feature sets. fjoin is a remarkably simple procedure, yet achieves  $O(n \log(n))$  performance ( $O(n)$ , if the inputs are sorted), using  $O(1)$  space. We have described the basic algorithm, shown its correctness, analyzed its performance, and discussed several useful extensions. An implementation of fjoin can be downloaded from <ftp://ftp.informatics.jax.org/pub/fjoin>.

## ACKNOWLEDGEMENTS

Thanks to Carol Bult, Joel Graber, Jim Kadin, and Ben King for reading drafts of this paper and providing numerous helpful comments and constructive criticisms. Thanks especially to Jim for suggesting a refinement that simplified the algorithm and for outlining the correctness proof.



## REFERENCES

- Bentley, J.L. 1977. Algorithms for Klee's rectangle problem. Unpublished. Computer Sciences Department, Carnegie-Mellon University, Pittsburgh.
- dbSNP. 2006. Online database resource at: <http://www.ncbi.nlm.nih.gov/SNP>
- Edelsbrunner, H. 1980. Dynamic rectangle intersection searching. Institute for Information Processing Report 47, Technical University of Graz, Graz, Austria.
- Ensembl. 2006. Online database resource at: <http://www.ensembl.org>
- Eppig, J.T., Bult, C.J., Kadin, J.A., Richardson, J.E., Blake, J.A., and the members of the Mouse Genome Database Group. 2005. The Mouse Genome Database (MGD): from genes to mice—a community resource for mouse biology. *Nucleic Acids Res* 2005; 33: D471-D475.
- Florea, L., Hartzell, G., Zhang, Z., Rubin, G., and Miller, W. 1998. A computer program for aligning a cDNA sequence with a genomic DNA sequence. *Genome Research* 8, 967-974.
- GBrowse. 2006. Open-source genome map browser at: <http://www.gmod.org/ggb/>
- GFF3. 2004. Generic Feature Format Version 3. Online documentation available at: <http://song.sourceforge.net/gff3-jan04.shtml>
- Guttman, A. 1984. R-trees: a dynamic index structure for spatial searching. *Proceedings of the ACM SIGMOD Conference*, Boston, pp. 44-57.
- Hill, D.P., Begley, D.A., Finger, J.H., Hayamizu, T.F., McCright, I.J., Smith, C.M., Beal, J.S., Corbani, L.E., Blake, J.A., Eppig, J.T., Kadin, J.A., Richardson, J.E., and Ringwald, M. 2004. The mouse Gene Expression Database (GXD): updates and enhancements. *Nucleic Acids Res* 32: D568-D571.
- Kent, W.J. 2002. BLAT -- The BLAST-Like Alignment Tool. *Genome Research* 4: 656-664.
- Korth, H. and Silberschatz, A. 1986. Database System Concepts. McGraw-Hill, New York, p. 315.
- Krupke, D.M., Näf, D., Vincent M.J., Allio, T., Mikaelian, I., Sundberg, J.P., Bult, C.J., and Eppig, J.T. 2005. The Mouse Tumor Biology Database: integrated access to mouse cancer biology data. *Exp Lung Res* 31: 1-12.
- Lapp, H., Mungall, C., Cain, S., and Stein, L. 2003. Optimizing Genome Interval Overlap Queries Using an R-Tree Index. *ISMB Abstract*, Brisbane, Australia.

MGI. 2006. Online database resource at: <http://www.informatics.jax.org>

Mouse Genome Sequencing Consortium (MGSC) and Mouse Genome Analysis Group. 2002. Initial sequencing and comparative analysis of the mouse genome. *Nature* 420:520-562.

Pearson, W.R. 2000. Flexible sequence similarity searching with the FASTA3 program package *Methods Mol. Biol.* 132:185-219.

PSL. 2006. PSL Alignment Format. Online documentation available at: [http://bioperl.org/wiki/PSL\\_alignment\\_format](http://bioperl.org/wiki/PSL_alignment_format)

Python. 2006. Open-source, interpreted, object-oriented language. <http://www.python.org>

Samet, H. 1990. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, New York.

UCSC. 2006. Online genome browser at: <http://genome.ucsc.edu/>