

–Supplementary Material–
libfbi: A C++ Implementation for Fast Box
Intersection and Application to Sparse Mass
Spectrometry Data

Marc Kirchner^{1,2,3,†}, Buote Xu^{1,2,†},
Hanno Steen^{1,2,3}, Judith Steen^{1,4}

¹Proteomics Center, Children’s Hospital Boston, Boston, MA, USA

²Department of Pathology, Children’s Hospital Boston, Boston, MA, USA

³Department of Pathology, Harvard Medical School, Boston, MA, USA

⁴Department of Neurobiology, Harvard Medical School

and T. M. Kirby Neurobiology Center, Children’s Hospital, Boston, MA, USA

February 7, 2011

1 Algorithms

The following sections give a brief overview over the design considerations underlying the `libfbi` implementation. The principal approach to solving the box intersection problem is to make use of a combination of a *segment* and a *range tree* [4]. The segment tree solves the batched stabbing problem; it stores intervals and allows for efficient point queries. The range tree solves the batched range search problem; it stores points and enables efficient interval queries. This combination allows to solve the interval intersection problem in a single dimension. We can derive the general box intersection approach based on two observations: (i) For two D -dimensional boxes to intersect, the boxes need to intersect in *all* D dimensions. Hence, with axes-parallel boxes, each dimension delivers an exclusion criterion and only boxes whose intervals intersect for all $d \in D$ can be accepted. This allows sequential probing of dimensions: given the set of overlapping intervals in dimension d , we recursively solve the interval intersection problems in dimensions $d+1, \dots, D$. (ii) Given two sets of intervals A and B , all intersecting intervals are given by the union of the results of two separate queries: first, build a segment tree for B viewed as intervals, query with C viewed as points, then build a range tree for B viewed as points and query with C viewed as intervals. Note, that the latter step is equivalent to building a segment tree for C and querying it with B . This duality between

segment and range trees allows us to avoid the construction of range trees and to solve the problem with two calls to a segment tree instead.

All remaining algorithmic changes are optimizations: with all intervals and queries available, there is no need to explicitly construct the segment trees as they will only be traversed once (in post-order) and we can limit space requirements to $O(|B| + |C|)$. In order to efficiently determine good pivot elements for the divide and conquer step of the segment tree, we make use of the *ApproxMedian* algorithm. Finally, when we reach the last dimension or when the problem becomes small enough (i.e. $|B|$ or $|C|$ is smaller than θ), we switch to a brute-force scanning method *OneWayScan* that avoids the comparatively large hidden constants associated with segment tree construction and performs linearly in the number of overlapping boxes between B and C .

1.1 The HYBRID algorithm

We give a brief overview over the components of *Hybrid*, see [4] for a more in-depth explanation and derivations.

Given two sets of boxes B and C , the *Hybrid* algorithm determines the set of intersecting boxes between B and C . The function takes two sets of boxes B and C , a working dimension d , the current working range in dimension d and the box intersection problem dimensionality D . For the (implicit) segment tree construction, we reduce the box information for B and C in the working dimension d to a set of lower endpoints P and a set of intervals I , respectively (the second call to *Hybrid* reverses the roles of B and C). Providing lower limits of a dimension is necessary to implement the segment tree divide and conquer strategy. The *Hybrid* algorithm then proceeds as follows (see Algorithm 1):

- 1-2:** Base of the recursion.
- 3-5:** First hybridization: always use a brute force scan in the last dimension.
- 6-8:** Second hybridization: switch to scanning if the number of boxes falls below a predefined cutoff θ .
- 9-11:** Determine the set I_m of intervals (for dimension d) that span the current segment and recursively call *Hybrid* for the next dimension $d+1$. Repeat, with I_m and P interchanged.
- 12:** Use the *ApproxMedian* function to calculate m , which divides the segment $[l, u]$ into $[l, m]$ and $[m, u]$. Note that we deviate from the original algorithm [4] here: Zomorodian and Edelsbrunner use the median of *points* which does not guarantee strictly decreasing set sizes, allowing infinite recursions in the degenerate case (e.g. if all points are the same so are all intervals between the points and $+\infty$; consequently, all splits will create an identical copy of the previous problem). Switching to using the median of the endpoints of the intervals yields a behavior that is more natural w.r.t. the underlying *segment tree* and circumvents these problems.

Algorithm 1: Hybrid(P, I, l, u, d, D)

Data: A set of boxes, interpreted as points in dimension d : P ; A set of boxes, interpreted as intervals in the current working dimension d : I ; the working range $[u, l]$ in dimension d , and the overall number of box dimensions D .

Result: An adjacency list containing all box intersections.

```
1 begin
2   if  $I = \emptyset$  or  $P = \emptyset$  or  $u \leq l$  then
3      $A \leftarrow \emptyset$  return  $A$ ;
4   else if  $d = D$  then
5      $A \leftarrow \text{OneWayScan}(P, I, d, D)$ ;
6     return  $A$ ;
7   else if  $|I| < \theta$  or  $|P| < \theta$  then
8      $A \leftarrow \text{OneWayScan}(P, I, d, D)$ ;
9     return  $A$ ;
10   $I_m \leftarrow \{\mathbf{i} \in I \mid [l, u] \subseteq i_d\}$ ;
11  Hybrid( $P, I_m, -\infty, +\infty, d + 1$ );
12  Hybrid( $I_m, P, -\infty, +\infty, d + 1$ );
13   $m \leftarrow \text{ApproxMedian}(I, h, d)$ ;
14   $P_l \leftarrow \{\mathbf{p} \in P \mid p_d < m\}$ ;
15   $I_l \leftarrow \{\mathbf{i} \in I \setminus I_m \mid i_d \cap [l, m] \neq \emptyset\}$ ;
16  Hybrid( $P_l, I_l, l, m, d, D$ );
17   $P_r \leftarrow \{\mathbf{p} \in P \mid p_d \geq m\}$ ;
18   $I_r \leftarrow \{\mathbf{i} \in I \setminus I_m \mid i_d \cap [m, u] \neq \emptyset\}$ ;
19  Hybrid( $P_r, I_r, m, u, d, D$ );
20 end
```

13-15 and 16-18: Set up the point and interval sets (in dimension d) for the left and right subsegments. Call *Hybrid* on these subsegments. This is the divide and conquer step.

Because the algorithm follows the idea that checking for a d -dimensional intersection between two boxes is equivalent to checking if all their d one-dimensional intervals overlap, it follows that the order in which the dimensions are processed can have a significant impact on evaluation speed (this is the ultimate reason why `libfbi` allows adjusting evaluation dimension ordering).

1.2 The APPROXMEDIAN algorithm

ApproxMedian (cf. Algorithm 2) is a fast approximate median finding algorithm used in the divide and conquer partitioning of *Hybrid*. The algorithm takes a set of boxes I and a dimension d (it will only consider the box intervals in dimension d), as well as a tree height parameter h . *ApproxMedian* then constructs a ternary

Algorithm 2: `ApproxMedian(I, h, d)`

Data: A set of boxes I , the maximum tree height h , and a dimension d

Result: An estimate for the median in dimension d

```
1 begin
2   if  $h = 0$  then
3      $\mathbf{i} \leftarrow$  randomly choose a box from  $I$ ;
4      $m \leftarrow$  randomly choose one of the endpoints of  $i_d$ ;
5     return  $m$ ;
6   else
7     return MedianOfThree (
8       ApproxMedian ( $I, h - 1, d$ ),
9       ApproxMedian ( $I, h - 1, d$ ),
10      ApproxMedian ( $I, h - 1, d$ ));
11 end
```

tree of height h with random points as leaves and evaluates it in a bottom-up fashion, recursively reporting the median of children nodes until it reaches the root. It then returns the median estimate m . The optimal height h is a free parameter that depends on the number of boxes in the sets and needs to be optimized. We have achieved satisfactory results using $h = 1.8 \cdot \log_{10}(|I|)$. The original description and analysis of the approach is given in [1], the original application to fast box intersection is in [4]. `MedianOfThree` is a simple function that picks the median from a set of three values.

1.3 The ONEWAYSCAN algorithm

OneWayScan (cf. Algorithm 3) is a brute force scanning approach to solve the box intersection problem in dimensions d, \dots, D . It takes two sets of boxes P and I , a current (starting) dimension d and the stopping dimension D (this is in general the dimensionality of the box intersection problem).

For the purpose of *Hybrid*, note that for two boxes $\mathbf{i} \in I, \mathbf{p} \in P$, intersections in all dimensions up to (but not including) the current dimension d have already been confirmed.

OneWayScan views one set (P in our case) as points, the other (I) as intervals in dimension d . After sorting, intersections can be determined in linear time: we maintain a set of active intervals and traverse over the sorted points. For a point p_d , we first add all intervals i_d which may contain p_d , i.e. start before (or at) p_d . We also remove all intervals that end before p_d , yielding the set of active intervals, each of which gives rise to a valid solution (p_d, i_d) for dimension d . This procedure is repeated for dimensions $d + 1, \dots, D$.

Algorithm 3: OneWayScan(P, I, d)

Data: Two sets of boxes P, I , a starting dimension d , and a stopping dimension D

Result: An adjacency list A with pairs of boxes whose intervals intersect in all dimensions d, \dots, D .

```
1 begin
2   sort  $P$  and  $I$  by their lower interval endpoints in dimension  $d$ ;
3    $A \leftarrow \emptyset$ ;
4    $C \leftarrow \emptyset$ ;
5   for  $p \in P$  do
6      $C \leftarrow C \cup \{\mathbf{i} \in I \mid i_{d,lower} \leq p\}$ ;
7      $C \leftarrow C \setminus \{\mathbf{i} \in I \mid i_{d,upper} \leq p\}$ ;
8     for  $c \in C$  do
9        $intersect \leftarrow true$ ;
10      for  $j \in \{d+1, \dots, D\}$  do
11        if  $c_j \cap p_j = \emptyset$  then
12           $intersect \leftarrow false$ ;
13      if  $intersect$  then
14         $A \leftarrow A \cup (c, p)$ ;
15   return  $A$ ;
16 end
```

2 Run Time Comparison

Among a set of potential alternatives, we chose to compare `libfbi` adjacency list determination runtimes with a kd-tree-based range query implementation. While kd-tree range queries and box intersection are two conceptually different problems and while their comparison should thus be regarded as somewhat semi-quantitative, kd-tree-based range queries have already been applied to mass spectrometry data analysis [3], and hence determining comparative query efficiency is of interest.

We compare two generic, C++ header-only libraries: `libfbi` and Savarese Software Research Cooperation’s `libssrckdtree`. We have used this kd-tree implementation extensively in LC/MS data processing over the last two years and consider it to be a well-maintained and efficient library.

For the benchmark, we chose the following setup: for each of the centroid sets \mathcal{L}_i of the 9 LC/MS experiments available in this study (see below and table 1), we determined a hierarchical partitioning $L_{i,1} \subset L_{i,2} \subset \dots \subset L_{i,11} \subseteq \mathcal{L}_i$, yielding data sets with up to $1.8 \cdot 10^6$ centroids. On each of the datasets, we determine the runtime by (i) constructing the respective data structure; (ii) issuing XIC range queries (i.e. attempting to find all centroids that belong to the same XIC); and (iii) clearing the data structure. The benchmark code is

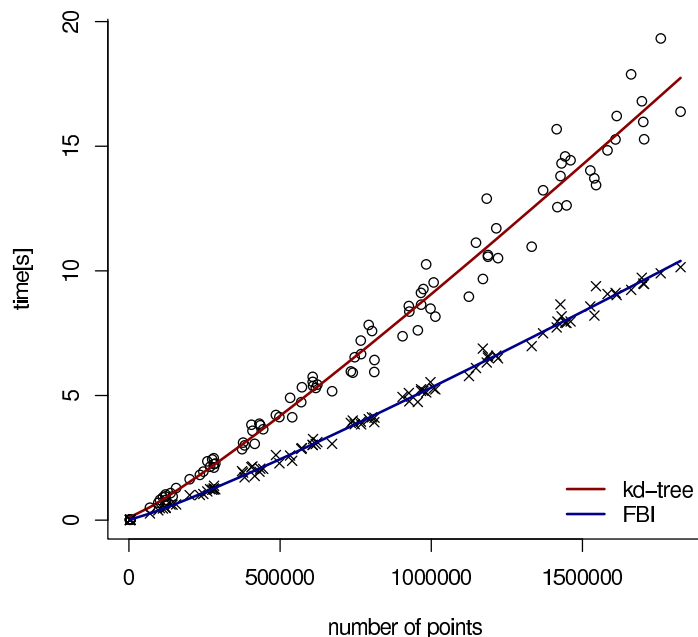


Figure 1: Runtime comparison between `libfbi` (dark blue) and a kd-tree implementation (dark red).

available in the `libfbi` example directory, along with all necessary header files to compile the benchmark executables. Compilation optimization for `g++` was set to `-O3`.

The runtime results are shown in figure 1. It comes as a slight surprise that the fast box intersection is running faster than the kd-tree across all datasets; however, we refrain from drawing extensive conclusions, acknowledging the fact that `libfbi` may be somewhat more tailored to the set intersection problem tackled in the given context (i.e. amortization of tree construction and single-sweep intersection determination vs. providing a generic storage container). Hence, figure 1 merely illustrates that the `libfbi` implementation is clearly able to rival existing kd-tree implementations in LC/MS range search scenarios. In summary, the availability of an efficient box intersection procedure such as `libfbi` should motivate researchers and vendors to retire existing approximations and include measurement uncertainty information in their feature extraction routines.

3 Mass Spectrometry Data Acquisition

HeLa S3 cells were grown in DMEM, supplemented with fetal bovine serum and penicillin/streptomycin. Cells were harvested using trypsin, washed with PBS and resuspended in 1x LDS sample buffer. The solution was boiled at

Fraction	Size
01	1696230
02	1824515
03	1701226
04	1609508
05	1544696
06	1758877
07	1646068
08	1460669
09	997322

Table 1: Fractions of the HeLa S3 data set, with number of centroids extracted from each fraction.

95 °C for 5 min, reduced, alkylated and run in a 1D SDS-PAGE gel. The gel was cut into fractions and in-gel digestion was performed using trypsin. Extracted peptides were acquired on a Thermo LTQ/Orbitrap instrument hyphenated to an Eksigent nanoLC 2D system (60min gradient, m/z 400-2000). Thermo .raw files were converted into MzXML using the ProteoWizard toolset (<http://proteowizard.sourceforge.net>). Centroids were extracted using an in-house program, implementing three-point Gaussian fits as detailed in the supplementary material of [2]. Table 1 lists the sizes (number of extracted centroids) for the different fractions of the data set.

References

- [1] Clarkson, K. L., Eppstein, D., Miller, G. L., Sturtevant, C., and Teng, S-H. (1996). Approximating center points with iterative radon points. *International Journal of Computational Geometry and Applications*, **6**, 357–377.
- [2] Cox, J. and Mann, M. (2008). Maxquant enables high peptide identification rates, individualized p.p.b.-range mass accuracies and proteome-wide protein quantification. *Nature Biotechnology*, **26**(12), 1367–1372.
- [3] Khan, Z., Bloom, J. S., Garcia, B. A., Singh, M., and Kruglyak, L. (2009). Protein quantification across hundreds of experimental conditions. *Proceedings of the National Academy of Sciences*, **106**(37), 15544–15548.
- [4] Zomorodian, A. and Edelsbrunner, H. (2000). Fast software for box intersections. In *SCG '00: Proceedings of the sixteenth annual symposium on Computational geometry*, pages 129–138, New York, NY, USA. ACM.