

2008

## Application of Blast-Based Techniques For Musical Information Retrieval

Fedor Aleksandrovich Korsakov  
*University of Northern Iowa*

*Let us know how access to this document benefits you*

Copyright ©2008 Fedor Aleksandrovich Korsakov

Follow this and additional works at: <https://scholarworks.uni.edu/hpt>

---

### Recommended Citation

Korsakov, Fedor Aleksandrovich, "Application of Blast-Based Techniques For Musical Information Retrieval" (2008). *Honors Program Theses*. 800.

<https://scholarworks.uni.edu/hpt/800>

This Open Access Honors Program Thesis is brought to you for free and open access by the Student Work at UNI ScholarWorks. It has been accepted for inclusion in Honors Program Theses by an authorized administrator of UNI ScholarWorks. For more information, please contact [scholarworks@uni.edu](mailto:scholarworks@uni.edu).

**Offensive Materials Statement:** Materials located in UNI ScholarWorks come from a broad range of sources and time periods. Some of these materials may contain offensive stereotypes, ideas, visuals, or language.

APPLICATION OF BLAST-BASED TECHNIQUES  
FOR MUSICAL INFORMATION RETRIEVAL

A Thesis

Submitted

in Partial Fulfillment

of the Requirements for the Designation

University Honors with Distinction

Fedor Aleksandrovich Korsakov

University of Northern Iowa

May 2008


This Study by: Fedor Korsakov

Entitled: Application of BLAST-based techniques for Musical Information Retrieval

has been approved as meeting the thesis or project requirement for the Designation  
University Honors with Distinction

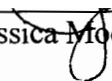
30/12/08

Date

  
/ Dr. Kevin O'Kane, Honors Thesis/Project Advisor

5/9/08

Date

  
Jessica Moon, Director, University Honors Program

# Application of BLAST-based Techniques for Musical Information Retrieval

Fedor KORSAKOV  
(student)  
Department of Computer Science  
University of Northern Iowa  
Cedar Falls, IA 50614, USA  
korsakov@uni.edu

## Abstract

Content retrieval in musical collections has been dependent on textual metadata (e.g. ID3 tags) which can present problems when the title of a piece is forgotten, misspelled, or when the search revolves around the similarity of sound. Content-based MIR (musical information retrieval) could offer an alternative. BLAST (basic local alignment search tool), an algorithm widely used in bioinformatics to search for sequences of aminoacids within longer sequences, seeks similarities and homologies, which makes it interesting for MIR, because musical information can be expected to be imprecise, and because homologies can allow to draw connections between musical pieces. Increased availability of digital music necessitates MIR methods which would allow to search a polyphonic sound collection with polyphonic queries to retrieve individual files, and a question can be raised how viable is BLAST-based retrieval for this kind of data. This paper discusses an implementation of such a system.

# 1. INTRODUCTION

Personal computers are being increasingly used for multimedia purposes. The emergence of new content distribution models has contributed to the growth of collections of digital music. One of the effects of this growth is the need for the enhancement of content retrieval mechanisms, which historically have been dependent on textual metadata associated with files. In the specific case of musical data, a common approach is to incorporate tags (such as ID3) into the files. While this method is adequate for textual search, there are drawbacks in using data representation significantly different from data itself. Textual search, for all of its simplicity, may have difficulties with situations where the title of a musical piece is forgotten, misspelled, or typed in a different language, or where the search revolves around the similarity of the sound (for example, looking for a remix that incorporates a classical piece) rather than the title. While tags can be edited to compensate for some of these problems, it is obvious that devising means to perform MIR (musical information retrieval) on the basis of content could offer a promising alternative solution. Furthermore, such an approach could lead to innovative user interfaces which would allow searches with acoustic queries which are hummed, spoken, or played on an instrument.

A conceptually alike challenge is common in bioinformatics, where it is frequently necessary to perform a search for a sequence of aminoacids within a longer sequence (e.g. genome). BLAST (basic local alignment search tool) is an algorithm widely used to address this need, and allows not only to look for the exact match, but also to find similar sequences. The emphasis on speed rather than accuracy makes BLAST-based retrieval interesting for MIR, since musical information, especially user queries, can be expected to be inherently imprecise.

# 2. LITERATURE REVIEW

A substantial body of information exists on the subject of MIR. Mongeau-Sankoff algorithm [8] is a seminal work which provides an effective similarity measure for pieces of music, but requires the information to be presented in a symbolic form. While it has been successfully used for MIR [1], the conversion of musical files into sheet music is beyond the scope of this research. Kline and Glinert [3] as well as Miura and Shioya [6] all agree that while pitch contours may be used for the purposes of fast retrieval, the accuracy is very poor. The latter research suggests using pitch spectrum (histogram of notes per bar) as a means to describe musical information, and even though the technique was developed for sheet music, the idea behind it may potentially be reapplied in the context of this research. While a fairly large number of works deal with monophonic (query by humming) or symbolic queries on symbolic databases, Yang's research [11] presents interest due to its focus on polyphonic queries on a polyphonic database. Yang uses spectral indexing and implies that this technique is rather unexplored. STFT (Short-Time Fourier Transform) is used to generate spectrograms, and then the resulting data is processed to determine characteristic sequences. It would appear obvious that due to difficulties of automated music transcription, spectrograms are a promising way to address the problem of finding a suitable means of representation for the music.

The suitability of various similarity measures to the problems of MIR has been assessed and discussed by Uitdenbogerd and Zobel [10]. Their research shows that local alignment is a superior similarity measure, even for short queries; however it is sensitive to normalization, as longer pieces will usually have more alignment matches, and the problem of normalization has not been fully addressed in the article. The approach I propose revolves around detection of regions that resemble the query in the spectrograms of the musical collection, and while this does not directly address normalization issue, it moves the emphasis towards finding a single high-ranking match within the

piece as opposed to assessing the relevance of the piece on the basis of the number of matches over its entire duration.

### 3. METHODOLOGY

The problem consisted of finding relevant files in a collection of MP3 files. 18 excerpts (2 - 48 seconds duration) of various musical pieces served as queries. Results were returned as lists of files. Effectiveness was assessed through presence of the relevant file among the top matches, the relevant file being the piece from which the excerpt originated. While most musical pieces were unique, some were present as multiple recordings (i.e. same piece played by various orchestras). In those situations, the top match with the musical piece in question was considered to be the relevant file regardless of whether it was the specific recording from which the query originated.

#### 3.1 Environment

The system has been implemented on Mac OS X 10.4, which facilitated access to a variety of open source libraries (FFTW [2], libpng [4] and libsndfile [5]) and utilities (SoX [9]) and led to a piece of software that can be easily ported to Linux in the future. The collection consisted of 106 MP3 files, containing music by Mozart, Haydn, Borodin, Bach, Beethoven, and other classical composers, as well as the more modern pieces by bands Solstice, Naglfar, Agalloch and Dimmu Borgir.

#### 3.2 Preprocessor

Out of the existing BLAST applications, MegaBLAST [7] was chosen to be used in the search system due to its efficiency in sequence alignment search among slightly different, relatively long sequences. However, in order to be used with MegaBLAST, the collection had to undergo initial processing. The MP3 files were decompressed and converted to monophonic WAV format with SoX. Single-precision real-to-complex STFT was done on the resulting data, producing an array of complex values. Complex moduli of these results were normalized and transposed (since the search was going to occur in the time domain). The next step involved conversion of the two-dimensionally arranged spectral data into sequential data on which MegaBLAST could operate. Each value in the array of normalized STFT results was converted into 2 nucleotide characters (each – out of four characters). Thus in the produced aggregate of sequences, each sequence corresponded to the intensities of a specific frequency over time, expressed as integers between 0 and 15 through the following formula, where  $V$  is the resulting value, and  $v$  is the normalized complex modulus:

$$V = \lfloor 16 \cdot \sqrt{\left( \frac{\sin\left(\frac{\pi \cdot v}{2}\right) + \sqrt{v}}{2} \right)^2} \rfloor$$

The aggregate was converted into FASTA-formatted text which was added to the musical database through the use of formatdb. The queries (based on the excerpts from musical files) had undergone similar preprocessing and were stored as separate files. FASTA headers contained additional information about the files, i.e. bit rate, as well as length and width of STFT array.

### 3.3 Postprocessor

The queries were submitted to a Ruby script that wrapped MegaBLAST in order to do additional analysis of the program's output. For each file where matches occurred, the matches' weighted scores were placed into a hash table where the keys were based on the starting time of the match divided by the duration of the query. Even though this approach potentially ignores closely positioned matches that fall outside of the interval of interest (one query duration long), the underlying assumption was that matches of significance would occur in a region corresponding to the query in size and not be dispersed further than query's duration. This provided a simple and efficient way of partitioning the search space into regions that could be later ranked, based on the combined score of the matches they contained. The combined score was acquired through the following formula, where  $S$  is the region score,  $D$  is the distance between the frequency of the sequence in the query and the actual frequency of the match and, and  $B$  is the MegaBLAST score the match received.

$$S = \sum_{i=0}^n \left( \frac{1}{(D_i + 1)} \cdot \left( \frac{10}{(B_i + 10)} \right) \right)$$

Since MegaBLAST tends to return matches that are fairly close, the variation in  $B$  can be viewed as largely acceptable. Thus the combined score emphasized frequency distance score ( $D$ ) and downplayed MegaBLAST score ( $B$ ). The overall result was that the regions were ranked by their scores, then files were ranked by the score of their highest-ranking region, and the ranked list of files was displayed to the user.

## 4. FINDINGS

On average, in a search, 99 files out of 106 were returned as containing a match. In 56% of the cases, the relevant file was in top 10 results, in 67% of the cases it was in the top 20, and in all cases it could be found in the top 50% of the output, thus it is possible to say that the system demonstrated basic feasibility of usage of MegaBLAST for MIR purposes. However, the likelihood of a successful search depended on several factors. The queries varied between random and systematically selected. The most successful searches were performed with short queries that sounded in a manner that was typical and unique for the pieces in question. Pieces by Mozart tended to yield large numbers of high-ranking unrelated results, while such pieces as "For All Tid" by Dimmu Borgir, with a monophonic solo, were recognized with a remarkable precision from short (4-5 seconds) queries. The precision of return did not significantly increase with the increase of the query size, instead more unrelated results were retrieved. The best results were returned with queries 4-10 seconds long. Thus, it is reasonable to conclude that a successful search is reliant on a query that is concise and characteristic of the musical piece in question, and that monophonic music may have a retrieval advantage over polyphonic music.

There were some additional findings related to the implementation. Variations of the number of nucleotide characters per STFT array value were attempted. It was discovered that 1 character (i.e. a range of 4 possible values) per value is clearly inadequate, as it leads to searches without conclusive results, whereas the use of more than 2 characters would create a database larger than the initial collection by more than 4 times, and therefore 2 characters per value were chosen.

## 5. DISCUSSION

While MegaBLAST can be viewed as interesting for the purposes of MIR, it is geared towards one-dimensional data, and its usage with music (two-dimensional data) introduces the need for substantial processing of the results. An algorithm specifically constructed for similarity searches in two-dimensional data could be more suitable to the problem, especially since the need to do computationally intensive preprocessing and to allocate a substantial amount of space (approximately 1.3 GB for the database used in this research) towards the storage of preprocessed data presents additional issues. However, with adequate postprocessing of the matches, MegaBLAST usage can yield useful results. Furthermore, a behavior that involves returning not only the queried musical piece, but also acoustically related pieces can be of interest to the user, and there are scenarios (e.g. Internet radio station) in which it may, in fact, be desirable.

Despite being capable of basic functionality, the system exhibits a number of weaknesses that should be addressed in future research. High return of polyphonic pieces of limited relevance presents a major issue. It may be possible to alleviate this problem by weighting scores on the basis of their frequency. A look at the STFT-generated spectrograms shows presence of artifacts and non-useful noise in the upper range of frequencies. Giving higher scores to the matches that occur closer to the mid-range of human hearing should reduce the impact of this problem. Another limitation is the fact that by ranking files on the basis of single best match, it is theoretically possible to overlook files with multiple good “good” matches but a mediocre “best” match. This demonstrates the importance of normalization, as mentioned by Uitdenbogerd and Zobel. With that said, in the majority of cases, relevant pieces still appeared among the top-scoring files. The issue that might be challenging to address is the reliance on characteristic queries. It may be possible to reduce the severity of this problem by performing a separate search that would look for the presence of query alone, rather than for the presence of regions resembling the query, but in its current state the system is better suited for retrieval of files with similar sounding quality rather than precise matches. Another research direction to consider is the usage of BLAST instead of MegaBLAST, which would increase the number of detected potential matches and may positively affect the results.

## SOURCES CITED

- [1] C. Gomez, S. Abad-Mota and E. Ruckhaus, “An analysis of the Mongeau-Sankoff algorithm for music information retrieval,” In *Proc. ISMIR 2007*, Austria, 2007, p. 109.
- [2] FFTW, <http://www.fftw.org/>
- [3] R. Kline and E. Glinert, “Approximate Matching Algorithms for Music Information Retrieval Using Vocal Input,” *MM’03*, 2003.
- [4] Libpng, <http://www.libpng.org/pub/png/libpng.html>
- [5] Libsndfile, <http://www.mega-nerd.com/libsndfile/>
- [6] T. Miura and I. Shioya, “Similarity among Melodies for Music Information Retrieval,” *CIKM’03*, USA, 2003.
- [7] MegaBLAST, <http://www.ncbi.nlm.nih.gov/blast/megablast.shtml>



- [8] M. Mongeau and D. Sankoff, "Comparison of Musical Sequences," *Computer and the Humanities*, vol. 24, 1990, pp. 161–175.
- [9] SoX, <http://sox.sourceforge.net/>
- [10] A. Uitdenbogerd and J. Zobel, "Melodic Matching Techniques for Large Music Databases," *ACM Multimedia '99, USA*, 1999, pp. 57-66.
- [11] C. Yang, "Efficient Acoustic Index for Musical Retrieval with Various Degrees of Similarity," *Multimedia'02, France*, 2002, pp. 584-591.

## APPENDICES

### A. Search Results

Query	Duration (s)	Rank
ECO_Symphony_No.40_-_I._Molto_allegro_excerpt3.txt	1	10
Solstice-excerpt.txt	2	33
Agalloch_-_Ashes_Against_The_Grain_-_05_-_Not_Unlike_The_Waves_excerpt3.txt	3	10
ForAllTid_excerpt.txt	4	1
Agalloch_-_Ashes_Against_The_Grain_-_04_-_Fire_Above,_Ice_Below_excerpt2.txt	5	1
Agalloch_-_Ashes_Against_The_Grain_-_04_-_Fire_Above,_Ice_Below_excerpt1.txt	5	4
swarm_of_plagues_excerpt.txt	5	7
ECO_Symphony_No.40_-_I._Molto_allegro_excerpt2.txt	5	8
Mozart-S25Gm_excerpt.txt	5	26
RCO_Symphony_No.21_in_A,_KV_134_-_iii._Menuetto_excerpt3.txt	6	2
Solstice-excerpt2.txt	6	45
Agalloch_-_Ashes_Against_The_Grain_-_05_-_Not_Unlike_The_Waves_excerpt1.txt	9	11
RCO_Symphony_No_38_in_D_major,_KV_504_Prague_-_III._Finale._Presto_excerpt1.txt	9	33
Agalloch_-_Ashes_Against_The_Grain_-_05_-_Not_Unlike_The_Waves_excerpt2.txt	10	7
RCO_Symphony_No.21_in_A,_KV_134_-_iii._Menuetto_excerpt2.txt	17	10
ECO_Symphony_No.40_-_I._Molto_allegro_excerpt1.txt	24	12
RCO_Symphony_No.21_in_A,_KV_134_-_iii._Menuetto_excerpt1.txt	35	28
ECO_Symphony_No.40_-_I._Molto_allegro_excerpt5.txt	48	44

### B. Preprocessor Code

```
#include <stdio.h>
#include <sys/types.h>
#include <dirent.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <fftw3.h>
#include <png.h>
#include <sndfile.h>
#define MUS_DIR "/music"
#define WAV_DIR "/wav"
#define IMG_DIR "/png"
```

```

#define DAT_DIR "/txt"
#define DBF_DIR "/db"
#define CONVERTER "sox"
#define XTD "wav"
#define XTO "mp3"
#define FNAME_LENGTH 256
#define CL_LENGTH 516
#define FN_SUCCESS 0
#define FN_FAILURE 1
#define FN_NORESUL 2
#define FRAME_SIZE 2
#define IMG_BLOCK_H 10000
#define IMG_BLOCK_W 64
#define S_F 10
#define CHUNK 4096
#define ADV 2000

int convert(struct dirent *);

int make_img_d(char *, float *, long, char *);
int make_img_d_block(char *, float *, long, int);

/* song name, in_data, data size, bitrate, output file */
int make_fasta(char *, float *, long, int, FILE *);

/* helper stuff */
long simplify(float *, float *, int, long);
int make_img_block(char *, float *, long, int);
int make_img(char *, float *, long, char *);

/* make them song-specific! */
/* float g_min = 100.0; */
float g_max = 0.0;

int spectrum_w = CHUNK/2 + 1;
char appdir[FNAME_LENGTH]="\0";

/* NOTES:
The frequencies have inverse proportional distribution.
Output is normalized!!! (26/01)
*/

main(argc, argv, envp)

    int argc;           /* Number of args */
    char *argv[];      /* Argument ptr array */
    char *envp[];      /* Environment ptr array */
{
    struct dirent *df;
    DIR *dr;
    char t_cf[FNAME_LENGTH + 4] = "";
    char cf[FNAME_LENGTH + 4] = "";
    SNDFILE *sf;
    SF_INFO sfinfo;

    time_t t1 = time(NULL);

    int a = 0;
    while (envp[a] != NULL)

```

```

{
    if (strncmp(envp[a], "PWD=", 4) == 0)
    {
        strncpy(appdir, strchr(envp[a], '/'), FNAME_LENGTH);
        printf ("appdir:%s\n", appdir);
        break;
    }
    a = a++;
}

char temp[FNAME_LENGTH] = "\0";
snprintf(temp, FNAME_LENGTH, "%s%s", appdir, MUS_DIR);
printf ("opendir:%s\n", temp);
dr = opendir(temp);

FILE *dbf;
temp[0] = "\0";
snprintf(temp, FNAME_LENGTH, "%s/db/collection.txt", appdir);

dbf = fopen(temp, "wb");
if (!dbf)
{
    printf("Error opening database file %s.\n", temp);
    return FN_FAILURE;
}

/* there is a directory...*/
if(dr != NULL) {
    /* get filenames from directory and for each of them...*/
    while((df = readdir(dr)) != NULL)
    {
        /*convert each suitable file */
        if (convert(df) == FN_SUCCESS)
        {
            /* construct path to converted file */

            strcat(t_cf, df->d_name);
            char f_cf[FNAME_LENGTH] = "\0";
            strncpy(f_cf, t_cf, strlen(t_cf)-3);

            snprintf(cf, FNAME_LENGTH, "%s%s/%s%s", appdir, WAV_DIR, f_cf, XTD);

            printf("Opening %s\n", cf);

            /* prepare room for converted sound file info */
            memset (&sfinfo, 0, sizeof (sfinfo));

            /* open converted file */
            if ((sf = sf_open (cf, SFM_READ, &sfinfo)) != NULL)
            {
                /* display file info */
                printf ("Sample Rate : %d\n", sfinfo.samplerate);
                if (sfinfo.frames > 0x7FFFFFFF)
                    printf ("Frames      : unknown\n" );
                else
                    printf ("Frames      : %ld\n", (long) sfinfo.frames);
                printf ("Channels   : %d\n", sfinfo.channels) ;
                printf ("Format    : 0x%08X\n", sfinfo.format) ;
                printf ("Sections  : %d\n", sfinfo.sections) ;
                printf ("Seekable  : %s\n", (sfinfo.seekable ? "TRUE" : "FALSE"));
            }
        }
    }
}

```

```

/* make room for sound data */
float *data_ptr = calloc(sfinfo.frames, sizeof(float));

/* suppose this does not work */
if (data_ptr == NULL)
{
    /* print error and exit. */
    fprintf(stderr, "calloc failure\n");
    exit(EXIT_FAILURE);
}

/* everything is going well, so read file into our array */
if (sf_readf_float(sf, data_ptr, sfinfo.frames) < (sfinfo.frames))
{
    /* should never happen! */
    printf ("Underread\n") ;
}
else
{
    printf ("Data read into memory OK: %ld\n", sfinfo.frames);
}

/* preparation work is done!!! */
/* break out the transform into a separate function later on! */
/* FFTW STUFF */

/* define necessary variables */
/* this is a real (float) to complex transform */
fftwf_complex *out;
float *in, *result;
fftwf_plan p;

/* humans hear in the range of 15 - 20000 Hz
   this means unsimplified sample is adequate
   and chunks should not be no smaller than 2940 frames
*/

/* allocate input and output arrays with fftw_malloc */

/* in is just chunk-sized */
in = (float*) fftw_malloc(sizeof(float) * CHUNK);

printf ("Input array allocated.\n");
out = (fftwf_complex*) fftw_malloc(sizeof(fftwf_complex) *
spectrum_w);

printf ("Output array allocated.\n");

/* create plan */
/* consider other flags!
http://www.fftw.org/fftw3\_doc/One\_002dDimensional-DFTs-of-Real-Data.html#One\_002dDimensional-DFTs-of-Real-Data
http://www.geosci.usyd.edu.au/users/jboyden/vad/index.html#chapter-fourier
*/
p = fftwf_plan_dft_r2c_1d(CHUNK, in, out, FFTW_ESTIMATE);
printf ("Plan created.\n");

/* intermediate result */
/* allocate array for result */
/* adjusting for flex ADV offset */

```

```

ADV) * spectrum_w);

result = (float*) malloc(sizeof(float) * ((sfinfo.frames-CHUNK+ADV) /
printf ("Result array allocated.\n");

/* loop through data! */
/* we need n-1 of starting points */

int f;
/* -1 because after starting point there is chunkful of bits to load
into in*/
for(f=0; f < ((sfinfo.frames-CHUNK+ADV) / ADV - 1); f++)
{
    /* populate in array by reading CHUNK into it*/
    int a;
    for (a=0; a < CHUNK; a++)
    {
        /*
        in[a]=left_data_ptr[f*ADV+a]; */
        in[a]=data_ptr[f*ADV+a];
    }

    /* execute plan */
    fftwf_execute(p);
    /* printf ("Plan executed %d: %d.\n",f,a); */

    /* we got result */
    int b;
    for (b=0; b < spectrum_w; b++)
    {
        /* complex modulus = sqrt(x^2 + y^2)
        where x is real and y is imaginary components of the
output
        */

        /* compute complex modulus and store result */
        result[f * spectrum_w + b] = sqrt(pow(out[b][0], 2) +
pow(out[b][1], 2));

        if (result[f * spectrum_w + b] > g_max)
        {
            g_max = result[ f * spectrum_w + b];
        }
    }

    printf ("STFT complete.\n");
    printf ("Max = %f.\n", g_max);

    if(make_fasta(cf, result, ((sfinfo.frames-CHUNK+ADV) / ADV) *
spectrum_w, sfinfo.samplerate, dbf) == FN_SUCCESS)
    {
        /* printf("HUGE SUCCESS\n"); */
    }
    else
    {
        printf("ERROR FORMATTING FOR FASTA");
    }

    /* PICTURES */
    if(make_img_d(cf, result, ((sfinfo.frames-CHUNK+ADV) / ADV) *
spectrum_w, "m") == FN_SUCCESS)

```

```

        {
            printf("HUGE SUCCESS\n");
        }
        else
        {
            printf("ERROR DRAWING SPECTRA");
        }

        /* destroy plan and arrays */
        fftwf_destroy_plan(p);
        fftwf_free(in);
        fftwf_free(out);
        g_max=0.0;

        /* cleanup main stuff */
        free(data_ptr);

        free(result);
    }
    else
    {
        printf("Error opening %s\n", cf);
    }

    sf_close (sf);

    cf[0] = '\0';
    t_cf[0] = '\0';
}

char cl[CL_LENGTH]='\0';

/* quick fix, needs to be set in PATH */
char prepend[FNAME_LENGTH]="/usr/local/blast-2.2.17/bin/";

snprintf(cl, CL_LENGTH, "%sformatdb -i ./db/collection.txt -p F -o F", prepend);
system(cl);

closedir(dr);
fclose(dbf);
}

printf("Entire process took %d\n",time(NULL)-t1);
return EXIT_SUCCESS;
}

/* song name, in_data, data size, bitrate */
int make_fasta(char *sname, float *data, long s, int brate, FILE *dbf)
{
    char fname[FNAME_LENGTH + 4] = "\0";
    char pname[FNAME_LENGTH + 4] = "\0";
    char bdesc[32] = "\0";

    /*    snprintf(pname, FNAME_LENGTH, "%s%s", appdir, DAT_DIR); */
    /* printf("%s\n", pname); */

    strncpy(fname, strrchr(sname, '/'), strlen(strrchr(sname, '/'))-4);

```

```

int y=0;
while (fname[y] != '\0')
{
    if (fname[y] == ' ')
    {
        fname[y] = '_';
    }
    y=y+1;
}

/*  snprintf(pname, FNAME_LENGTH, "%s%s%s", appdir, DAT_DIR, fname); */
snprintf(pname, FNAME_LENGTH, "%s%s%s.txt", appdir, DAT_DIR, fname);
/*  printf("%s\n", pname); */

/*  snprintf(pname, FNAME_LENGTH, "%s.txt", pname); */
int i;
snprintf(bdesc, 26, "r:%06d|w:%05d|l:%06d", brate, spectrum_w, s/spectrum_w);
/*  print header */
/*fprintf(dbf, ">%s|s\n", fname,bdesc); */
int line_length = 32; /*x2*/
/* 2 bit! */
char code[4];

code[0]='A';
code[1]='C';
code[2]='G';
code[3]='T';

FILE *out;

if (strcasestr(fname, "excerpt") == NULL)
    out = dbf;
else
{
    printf("%s\n", pname);
    out = fopen(pname, "wb");
}

fprintf(out, ">%s|s\n", pname,bdesc);
for (i = 0; i < spectrum_w-1; i = i + 1)
{
    int j = 0;
    while (j< s/spectrum_w)
    {
        fprintf(out, "%c", code[ ((int) (16 * pow( 0.5*(sin(1.57 *
data[i+j*spectrum_w] / g_max) + sqrt(data[i+j*spectrum_w] / g_max)),1.0/2))) >> 2]);
        fprintf(out, "%c", code[ ((int) (16 * pow( 0.5*(sin(1.57 *
data[i+j*spectrum_w] / g_max) + sqrt(data[i+j*spectrum_w] / g_max)),1.0/2))) & 3]);

        j = j + 1;
        if (j%line_length==0)
        {
            fprintf(out, "\n");
        }
    }
    fprintf(out, "\n");
}
if (out != dbf)
    fclose(out);
return FN_SUCCESS;

```

```

}

/* generates image set of spectra
  mfname - mus file name
  data - array of doubles
  s - complete length of data
  chan - channel suffix */
int make_img_d(char *mfname, float *data, long s, char *chan)
{
    /* make img file name out of mus file name */
    char imgf[FNAME_LENGTH] = "\0";
    char imgf2[FNAME_LENGTH] = "\0";
    /* for all data given */
    int i;
    for (i = 0; i < s; i=i+(IMG_BLOCK_H * spectrum_w))
    {
        strncpy(imgf, strrchr(mfname, '/'), strlen(strrchr(mfname, '/'))-4);
        sprintf(imgf2, FNAME_LENGTH, "%s%s%s%09d%s.png", appdir, IMG_DIR, imgf, i, chan);

        /* write img file */
        make_img_d_block(imgf2, data, s, i);
        imgf[0]='\0';
        imgf2[0]='\0';
    }
    return 0;
}

/* generates vertical image of spectrum with specified name, for specified data and
  specified height

  spfname - filename to use
  data - pointer to datablock of doubles
  s - total length of data
  offset - starting point

  */
int make_img_d_block(char *spfname, float *data, long s, int offset)
{
    FILE *fp;
    png_structp png_ptr;
    png_infop info_ptr;
    png_infop end_info;

    int act_height;

    /* determine actual height */
    if ((s - offset) / spectrum_w < IMG_BLOCK_H)
    {
        act_height = (s - offset)/spectrum_w;
    }
    else
    {
        act_height = IMG_BLOCK_H;
    }

    /* open file */
    fp = fopen(spfname, "wb");

```



```

if (!fp)
{
    printf("Error opening image %s\n", spfname);
    return FN_FAILURE;
}

/* allocate memory for image data */
png_ptr = png_create_write_struct(PNG_LIBPNG_VER_STRING,
                                png_voidp_NULL,
                                png_error_ptr_NULL,
                                png_error_ptr_NULL);

if (!png_ptr)
{
    fclose(fp);
    printf("Error allocating memory.\n");
    return FN_FAILURE;
}

/* allocate memory for image info */
info_ptr = png_create_info_struct(png_ptr);
if (info_ptr == NULL)
{
    fclose(fp);
    png_destroy_write_struct(&png_ptr,
                            (png_infopp)NULL);
    printf("Error allocating memory.\n");
    return FN_FAILURE;
}

/* error handling */
if (setjmp(png_jmpbuf(png_ptr)))
{
    png_destroy_write_struct(&png_ptr, &info_ptr);
    fclose(fp);
    /* i.e. problem reading file (???) */
    return FN_FAILURE;
}

/* we are using standard C streams */
png_init_io(png_ptr, fp);

/* setting file info */
printf("Dim: %ld x %ld.\n", spectrum_w, act_height);

png_set_IHDR(png_ptr,
             info_ptr,
             spectrum_w,
             act_height,
             4, /* 16 initially, but it is too much*/
             PNG_COLOR_TYPE_GRAY,
             PNG_INTERLACE_NONE,
             PNG_COMPRESSION_TYPE_DEFAULT,
             PNG_FILTER_TYPE_DEFAULT);

png_write_info(png_ptr, info_ptr);

/* we have an array of doubles */
int i;
char r[spectrum_w/2];

for (i = offset; i < (offset + act_height * spectrum_w); i = i + spectrum_w)

```

```

    {
        int j;
        for (j=0; j < spectrum_w/2; j++)
        {
            char x = (uint8_t) 16 * pow( 0.5*(sin(1.57 * data[i + 2*j] / g_max) +
sqrt(data[i + 2*j] / g_max)),1.0/2);
            char y = (uint8_t) 16 * pow( 0.5*(sin(1.57 * data[i + 2*j+1] / g_max) +
sqrt(data[i + 2*j+1] / g_max)),1.0/2);

                x<<=4;
                r[j] = x|y;
            }
            png_bytep row_pointer = (png_bytep)r;
            png_write_row(png_ptr, row_pointer);
        }
        /* printf("2 - OKAY\n"); */

        /* after writing */
        /* write the rest of the file */
        png_write_end(png_ptr, info_ptr);
        /* cleanup and free memory */
        png_destroy_write_struct(&png_ptr, &info_ptr);

        /* close file */
        fclose(fp);
        return FN_SUCCESS;
    }

int convert(struct dirent *f)
{
    char cl[CL_LENGTH]="";
    char ofname[FNAME_LENGTH]="";
    int dot;

    printf("%s\n", f->d_name);

    if (dot = strstr(f->d_name,XTD) != NULL)
    {
        /* assemble shell command for sox */
        strcat(ofname,f->d_name,strlen(f->d_name)-3);

        /*      snprintf(cl, CL_LENGTH, "%s \"%s/%s\" \"%s/%s%s\"", CONVERTER, MUS_DIR, f-
>d_name, WAV_DIR, ofname, XTD); */
        snprintf(cl, CL_LENGTH, "%s \"%s%s/%s\" -c 1
\"%s%s/%s%s\"", CONVERTER, appdir, MUS_DIR, f->d_name, appdir, WAV_DIR, ofname, XTD);

        printf(">>> %s\n", cl);
        system(cl);
        return FN_SUCCESS;

        /*      system("%s %s/%s %s/%s%s", CONVERTER, MUS_DIR, dp->d_name, WAV_DIR, ofname,
XTD); */
    }
    else
    {
        return FN_NORESUL;
    }
}

/* helper/testing function

```

```

    generates image set
    mfname - mus file name
    data - array of floats
    s - complete length of data
    chan - channel suffix */
int make_img(char *mfname, float *data, long s, char *chan)
{
    /* make img file name out of mus file name */
    char imgf[FNAME_LENGTH + 4] = "";
    char imgf2[FNAME_LENGTH + 4] = "";
    /* for all data given */
    int i;
    for (i = 0; i < s; i=i+IMG_BLOCK_H)
    {
        /* remove unnecessary extension */
        strncat(imgf, mfname, strlen(mfname)-4);
        /* prepend dirname to filename */
        strncpy(imgf, IMG_DIR, 5);
        /* append seq num and extension to filename */
        sprintf(imgf, "%s%09d%s%s", imgf, i, chan, ".png");

        /*printf("%s\n",imgf); */

        /* write img file */
        make_img_block(imgf, data, s, i);
        imgf[0]='\0';
    }
    return 0;
}

/* helper/testing function
   generates vertical image with specified name, for specified data and specified height

   spfname - filename to use
   data - pointer to datablock
   s - total length of data
   offset - starting point

*/
int make_img_block(char *spfname, float *data, long s, int offset)
{
    FILE *fp;
    png_structp png_ptr;
    png_infop info_ptr;
    png_infop end_info;

    int act_height;

    /* determine actual height */
    if (s - offset < IMG_BLOCK_H)
    {
        act_height = s - offset;
    }
    else
    {
        act_height = IMG_BLOCK_H;
    }
}

```

```

/* open file */
fp = fopen(spfname, "wb");
if (!fp)
{
    printf("Error opening image %s\n", spfname);
    return FN_FAILURE;
}

/* allocate memory for image data */
png_ptr = png_create_write_struct(PNG_LIBPNG_VER_STRING,
                                  png_voidp_NULL,
                                  png_error_ptr_NULL,
                                  png_error_ptr_NULL);

if (!png_ptr)
{
    fclose(fp);
    printf("Error allocating memory.\n");
    return FN_FAILURE;
}

/* allocate memory for image info */
info_ptr = png_create_info_struct(png_ptr);
if (info_ptr == NULL)
{
    fclose(fp);
    png_destroy_write_struct(&png_ptr,
                             (png_infopp)NULL);
    printf("Error allocating memory.\n");
    return FN_FAILURE;
}

/* error handling */
if (setjmp(png_jmpbuf(png_ptr)))
{
    png_destroy_write_struct(&png_ptr, &info_ptr);
    fclose(fp);
    /* i.e. problem reading file (???) */
    return FN_FAILURE;
}

/* we are using standard C streams */
png_init_io(png_ptr, fp);

/* setting file info */
/* IMG_BLOCK_DIM is a magic number */
png_set_IHDR(png_ptr,
             info_ptr,
             IMG_BLOCK_W,
             act_height,
             16,
             PNG_COLOR_TYPE_GRAY,
             PNG_INTERLACE_NONE,
             PNG_COMPRESSION_TYPE_DEFAULT,
             PNG_FILTER_TYPE_DEFAULT);

png_write_info(png_ptr, info_ptr);

/* for now, let's presume an array is made of floats */
/* therefore, each frame translates into a line up to IMG_BLOCK_W px */
int i;
int b;

```

```

png_uint_16 r[IMG_BLOCK_W];
for (i = offset; (i < offset + act_height); i++)
{
    b = IMG_BLOCK_W / 2 * (data[i]+1.0); /* length of line */

    int j;
    for (j=0; j<IMG_BLOCK_W; j++) {
        if (b > j)
        {
            r[j] = (png_uint_16) 65535;
        }
        else
        {
            r[j] = (png_uint_16) 0;
        }
    }
    png_bytep row_pointer = (png_bytep)r;
    png_write_row(png_ptr, row_pointer);
}
/* printf("2 - OKAY\n"); */

/* after writing */
/* write the rest of the file */
png_write_end(png_ptr, info_ptr);
/* cleanup and free memory */
png_destroy_write_struct(&png_ptr, &info_ptr);

/* close file */
fclose(fp);
return FN_SUCCESS;
}

/* helper function
reduces size of dataset by averaging it */
long simplify(float *in, float *out, int confactor, long l_in)
{
    long i;
    long j;
    float x = 0;

    for(i = 0; i*confactor < l_in; i++)
    {
        j=0;
        while((j < confactor) && (i*confactor+j < l_in ))
        {
            x = x+in[i*confactor + j];
            j++;
        }
        x = x/(j+1);
        out[i] = x;
        x = 0;
    }
    printf("%ld\n", i);

    return i;
}

```

## C. Postprocessor Code

```

#!/usr/bin/env ruby

require 'optparse'
require 'ostruct'

# Fedor Korsakov
# Undergrad research final component
# 2008/04/26
# Analysis of MegaBLAST results

class Blaster
  attr_accessor :f

  def initialize()
    @f = ""
    #@targets = Array.new
  end #initialize

  def parse(args)

    opts = OptionParser.new do |opts|
      opts.banner = "Usage: binsearch.rb [options]"

      opts.on("-f FILE", "--file FILE", String, "Query file") do |ar|
        @f = ar
      end

      #opts.on("-t ARRAY", "--targets ARRAY", Array, "Array of target files") do |
ar|
        # ar.each do |el|
        #   @targets << el
        # end
        #end

      opts.on_tail("-h", "--help", "Show this message") do
        puts opts
        exit
      end
    end

    if args.empty?
      puts opts
      exit
    end
    begin
      opts.parse!(args)
    rescue OptionParser::InvalidOption => e
      puts e
      puts opts
      exit
    rescue OptionParser::MissingArgument => e
      puts e
      puts opts
      exit
    rescue OptionParser::InvalidArgument => e
      puts e
      puts opts
      exit
    end
  end #parse
end

```

```

def bsearch()
  cmd_string = "megablast -d db/collection.txt -i txt/#{@f} -D 0 -g T"
  query_length = 0
  File.open("txt/#{@f}", 'r') do |qfile|
    query_length = qfile.readline.slice(/(l:\d+)/).slice(/(\d+)/).to_i
  end
  puts "FILE          = #{@f}"
  puts "QUERY LENGTH = #{query_length}"
  megablast_results = IO.popen(cmd_string)

  # make array to hold scores, it'll be an array of structs
  megablast_results_parse = Array.new
  file_tree = Hash.new

  # read a line
  # e.g. '/Users/fedorkorsakov/Desktop/RESEARCH/code/current/thesis/txt/Solstice_
  _New_Dark_Age_-_04_-_Hammer_Of_Damnation.txt|r:044100|w:02049|l:011011'=='+1_' (5168231
  20227 5168260 20256) 0

  megablast_results.each_line do |line|
    match = OpenStruct.new
    match.filename = line[/(\.\.txt)/]
    match.length = line[/(\d+)/].slice(/(\d+)/).to_i #l:
    match.bitrate = line[/(\d+)/].slice(/(\d+)/).to_i #selfexplanatory
    match.width = line[/(\d+)/].slice(/(\d+)/).to_i #w:
    match.start = line[/(\d+\s)/].slice(/(\d+)/).to_i #offset in the file
    match.q_start = line[/(\d+\s\d+\s)/].slice(/(\s\d+)/).to_i #offset in the
query
    match.score = ((match.start / match.length).to_i - (match.q_start /
query_length).to_i).abs
    match.blastscore = line[/\d+\)\s\d+\/].slice(/(\s\d+)/).to_i #blast score

    # store results for now... GET RID LATER?
    megablast_results_parse = megablast_results_parse.push(match)

    #so we create a hash of filenames, each value corresponds to a struct that
hold the score and a list of matches
    # if added match scored well enough (i.e. low), we increase filescore

    if file_tree.key?(match.filename) == false
      file_tree[match.filename] = OpenStruct.new
      file_tree[match.filename].score = 0.0
      file_tree[match.filename].mlist = Array.new #matchlist
      file_tree[match.filename].distr = Hash.new(0.0) #distribution of matches
      file_tree[match.filename].sortd = Array.new #sorted distribution
      file_tree[match.filename].dstsc = 0.0 #highest density of
matches
      file_tree[match.filename].dsloc = 0.0 #highest density of
matches location
      file_tree[match.filename].brate = match.bitrate #bitrate
      file_tree[match.filename].length = match.length #f_length
      file_tree[match.filename].width = match.width #f_length
    end

    file_tree[match.filename].mlist = file_tree[match.filename].mlist.push(match)

    file_tree[match.filename].score = file_tree[match.filename].score +
1.0/match.length

    file_tree[match.filename].distr[((match.start %

```

```

match.length)/query_length).to_i] = file_tree[match.filename].distr[((match.start %
match.length)/query_length).to_i] + (1.0/(match.score + 1))*(10.0/(match.blastscore + 10))

    #end
end

# all output is read!
# now we have array of matches megablast found, and hash of files where matches
happen

results = Array.new
file_tree.each_pair do |key, value|
  wmatch = OpenStruct.new
  wmatch.name = key
  wmatch.score = value.score
  results = results.push(wmatch)
  #puts "#{key} SCORE=#{value.score}"
end

results.sort! {|x,y| x.score <=> y.score}

# results.each {|x| puts "#{x.score} #{x.name}"}

puts ""
puts ""

file_tree.each_pair do |key, value|
  value.sortd = value.distr.to_a
  value.sortd.sort! {|x,y| y[1] <=> x[1]}
  value.dstsc = value.sortd[0][1]
  #puts value.dstsc
  value.dsloc = value.sortd[0][0]
  #puts value.dsloc
end

results2 = Array.new
file_tree.each_pair do |key, value|
  wmatch = OpenStruct.new
  wmatch.name = key
  wmatch.score = value.dstsc
  wmatch.loc = value.dsloc
  results2 = results2.push(wmatch)
  #puts "#{key} SCORE=#{value.score}"
end

results2.sort! {|x,y| x.score <=> y.score}

results2.each {|x| puts "#{x.score}@ #{ x.loc * query_length *
file_tree[x.name].width / file_tree[x.name].brate } SECONDS IN #{x.name}"}

#best_match = results2.length
#results2.each do |match|
#  if @targets.include?(match.name)
#    puts match.name
#    best_match = results2.length - results2.index(match)
#  end
#end

#puts "Target found in position #{best_match}"
#puts "\t#{results2[results2.length - best_match].name}"

```



```

    #megablast_results_parse.each do |m|
      #m.score = ((m.start / m.length).to_i - (m.q_start / query_length).to_i).abs
      # puts "name = #{m.filename}, length = #{m.length}, bitrate = #{m.bitrate},
width = #{m.width}, offset = #{m.start}, q_offset = #{m.q_start} score = #{m.score}"
      #end

      #megablast_results_parse.sort! {|x,y| y.score <=> x.score }
      #puts
"-----"
      #puts @f
      #megablast_results_parse.reverse_each do |m|
        # puts "name = #{m.filename}, length = #{m.length}, bitrate = #{m.bitrate},
width = #{m.width}, offset = #{m.start}, q_offset = #{m.q_start} score = #{m.score} AT
#{(m.start % m.length) * m.width / m.bitrate } S"
        #end
      end #bsearch
    end #Binsort

if __FILE__ == $0
  bs=Blaster.new()
  bs.parse(ARGV)
  t1 = Time.now
  bs.bsearch()
  t2 = Time.now
  puts ""
  t=t2-t1
  puts "It took #{t} seconds."
end

```