GPS!

(Going Places… Sonically!)

Saif Choudhury

Kirstin Cummings

Brian Rulifson

GPS Sonification

We see in the near future the introduction of handheld communication devices that will incorporate various forms of technology to assist the user in navigating their everyday life, one of which will certainly be Global Positioning System data. With the ready access to a log of this data in our communication devices we see an opportunity to provide a method for both personal musical expression as well as a method for a form of social interaction through the sonification of this data. This time-delayed interaction should serve to illuminate patterns in daily movements amongst self-selected social groups through predictable, rule-based sonification schemes.

Current technology allows for limited use of real-time music making potential as the combination of GPS unit, laptop, and real-time data transfer is an unwieldy package. For this reason we focused on time-delayed music generation and for a single, non-interacting source while creating a framework extensible to additional data sets. This reduced the amount of logic coding necessary to relate action to meaningful sound, but at the same time began a framework that could be extended to multiple data streams through selected "dipping" methodologies to integrate the various streams.

While we explored data sets from sources as diverse as snowmobile paths in Alaska to albatross migration data to London taxi dispatching data, we focused on something closer to home: the Bay Area commute. Because we were able to focus on a limited geography and a relatively small set of commutation potentials, we were able to explore the effects of various types of commutes on our sonification routines and how we could make them more and less general.

Overview

Our project has three main phases of operation, analogous to the “real” musical process: Composition, Arrangement, and Performance. They’re implemented in our project in the following manner:

Composition – the collection of the data with the GPS unit
Arrangement – the processing in C and in Pd
Performance – the audio output from Pd, with the real-time user controls

Our data is “composed” by the user interacting with the world, and having his/her motion recorded as data. The following data is “arranged” through a combination of analysis in C, musical processing in Pd, and user input. The piece is then “performed” by Pd, with improvisational elements coming from a user, interfacing with Pd via controls connected to the Atmel board. Like the “real” musical process, you can first have control over writing the score with your movements, and then be able to arrange it as music. Your control does not end there, however. During the performance you can change the interpretation using the dials, so that even though the score is set, you can re-experience

the piece with different scales, tempos, voices, etc. This gives you a chance to explore your journey in ways you cannot do with simple numbers printed off a GPS sensor.

Musical/Interface Implementation

Much of music relies on the concept of progression. In tonal harmony, music can be viewed as a series of tonal elements that returns to a tonal center. In modern music, that tonal center, the tonic note, or the "key" of the whole "piece" is not necessarily required to be fixed and can be varied. However, as modern as we get, we still use some of the rules of tonality in music, because years later, tonal technique still provides us with a pleasant musical experience.

A common characteristic of most of Western music is the chord progression. That is to say, within a song, the perceived "tonic" note can move, following a pattern. A common way for the tonic to move is along the Circle of Fifths. That is, a song that starts out in C may move to G, then on to D, then on to A...and so on. Our philosophy is that harmony, more than melody, is the most effective medium for conveying the idea of motion in data through music. We chose to concentrate not on how position in real life maps to position along the musical scale, but rather how position in real life maps to position in a tonal progression.

Mapping motion to harmonic progression gives a listener a musical sense of motion. True, some sense of motion can be interpreted from notes and melody, but aside from basic motives, a listener can gather only so much information from the bare notes of a melodic line, and a much deeper experience from the variations in harmony, rather than melody. Using modular arithmetic to determine notes does not detract from the expression of motion in the music, because it's not very meaningful to encode motion into the scale itself. Were absolute position to be mapped directly to absolute notes, the pitches in the music would change so infrequently that one would hear notes several minutes long, as if one were listening to the test of the Emergency Broadcast System.

However, to cause the root, or the current tonic note, to move, we allow data to be converted into melody, but *information* to be encoded as harmony. The motion of chords in music is vastly more informative than the motion of individual notes, according to our philosophy. It allows the end result to not only sound like music, but sound like interesting, dynamic music.

Our musical instrument has both a graphical and physical user interface. The graphical user interface (see the *main* patch in the file *pd_patch_images.pdf*) includes indicators for tempo, volume, current tonic note, and scale choice, as well as a progress bar that shows how much of the musical piece has been played. Also in the graphical user interface are buttons and toggle switches for mute, pause, play, rewind, and MIDI activation.

The physical user interface is an acrylic enclosure containing the Atmel board and housing four dials and one button. The dials map to tempo, volume, tonic note, and scale choice, and the button maps to pause. The enclosure has a sloped face with ports in the back for power and MIDI cables, making it look very much like a modern, slick electronic musical instrument with a resemblance to a dashboard.

Tempo and Volume are each represented by a vertical slider. The sliders are connected directly to the tempo and volume dials on the physical user interface. Scale Choice is a vertical "dial" (in Pd nomenclature; it actually looks like an array of squares) indicating which of the six available keys is activated. It is also connected to a dial on the physical interface, and this dial is used as the selector. Tonic Note is visually displayed using a Pd dial, and the current tonic note is incremented and decremented when the user turns the corresponding dial on the physical interface clockwise and counterclockwise, respectively. The pause button is the only physical button; the others, corresponding to mute, play, rewind, and MIDI on/off, are graphical.

Technical Implementation

The "analyze.c" program parses the GPS data files it receives into a file format that the Pd patches can recognize. It also does many of the calculations and large-scale data gathering that would be awkward in the Pd patch itself. Once the GPS data has been gathered, the analyze program turns that information into a composition that will be read in and manipulated in real time with the pd patch and GPS board. The program takes several parameters in addition to the GPS file name. You can ask it to reverse the data to see what the data sounds like backwards, as if you were traveling back in time, or starting on the other end of a commute. You can also specify the number of seconds you would like in the song produced by the program. The program then returns only as many points in the output file as it would take for the Pd patch to create a song that particular length.

The analyze program keeps track of the total number of points it will output so that the Pd patch can time the piece accordingly. It returns the maximum values for all three spatial dimensions so that the pd patch can scale the piece to fit with the overall range of the data. This way the song's peaks and dips will be relative to each other, not to some absolute value. This sort of relative scaling keeps the piece interesting to listen to for small trips and not overly complicated or extreme for large distances. The GPS unit does not necessarily take samples at regular intervals. To avoid having points given to the Pd patch that are separated by anywhere between 1 second and 5 minutes, analyze takes each data point and then interpolates between the points to make sure that the samples are evenly spaced. This guarantees that the music you hear is in the right temporal order, and spaced correctly within the song. The program then calculates the velocity, angle, acceleration and other such useful variables to be used by the musical algorithms in the Pd patch. These values, along with the scaling information, are printed to a file which is now the composition for the Pd patch to read.

Implementing a musical philosophy required some algorithmic acrobatics. The following mappings were chosen to convert data into musical information:

Variable	Variable Symbol	Mapped to:
North latitude	*X*	Notes for Voice 1
Change in N. latitude	*Dx*	Rhythm for Voice 1
East latitude	*Y*	Notes for Voice 2
Change in E. latitude	*Dy*	Rhythm for Voice 2
Elevation	*Z*	Note volume
Change in elevation	*Dz*	Harmonic progression

Variable x and y are manipulated symmetrically, as are dx and dy. This can be seen in the Pd patched *voice_x* and *voice_y*. In *voice_x*, a numerical value from the data stream for x is multiplied by 10,000, then converted to an integer, and then fed to the *scalechoice* patch, where the resulting integer is reduced modulo 13 to choose from one of 13 MIDI note values. The scale patch then sends the note back to *scalechoice*, which sends the note through *voice_x* to the subpatch *send_audio_out*.

The subpatch *voice_x* also receives the data stream dx, and divides those values by the static value max_dx, the value of maximum magnitude that dx achieves from the data; this value is calculated by the analysis program and is sent to Pd through the qlist. The adjusted values of dx, which are now between 0 and 1, are multiplied by 40, converted to integers, and reduced modulo 4. These values control a spigot for the note stream coming from *voice_x*; depending on the value of the adjusted dx, a new note may or may not be generated by *voice_x*, causing a pattern of rhythmic variety in the musical line.

The exact same processes are true for the variables y and dy, the constant max_dy, and the patch *voice_y*. The difference is that *voice_x*, generated by latitude data, is sent to the left audio channel, while *voice_y*, generated by longitude data, is sent to the right audio channel. This makes it easier to tell what is controlling the sound you hear.

Elevation is treated in a different manner. The variable z is read by the subpatch *send_audio_out*, and is used to alter the volume of the computed audio. The data stream z is multiplied by 10,000, converted to an integer, then reduced modulo 128 and divided by 128, to produce values between 0 and 1. These values are used as a multiplicative factor when computing the audio, so as to create the impression of varying intensity in the music. To ensure smoothness in the audio, a simple ramp envelope is used when a new amplitude value is calculated.

The subpatch *key_struct* performs many tasks relating to the harmonic progression of the music. Within *key_struct*, change in elevation is mapped to harmonic progression as follows: dz is divided by max_delev, the maximum change in elevation, to generate values between -1 and 1. These values are multiplied by 6 and made positive integers. A control structure then checks the values; if the value is equal to 4, the current key is incremented; i.e., the tonic is advanced through the Circle of Fifths.

Also, within *key_struct*, the bass voice values are computed. The initial tonic note is set to MIDI note 60, or middle C. As the data is computed, the tonic note structure will receive instructions to increment the tonic note. Basically, a counter keeps track of what the current tonic note is, and the value is converted to a MIDI note number by the following formula:

$$k = 7r \pmod{12} + 60$$

where k is the current MIDI note number for the tonic note, and r is the value of the counter, which is initially set to 0. This algorithm ensures that as *key_struct* receives instructions to increment the key, the current tonic note is substituted with the next note on the Circle of Fifths. This leads to the following pattern:

r	k	MIDI note #	Note name
0	0	60	C
1	7	67	G
2	2	62	D
3	9	69	A
4	4	64	E
5	11	71	B
6	6	66	F# (Gb)
7	1	61	C# (Db)
8	8	68	Ab (G#)
9	3	63	Eb
10	10	70	Bb
11	5	65	F
12	0	60	C

As can be seen from the chart, as the counter variable r is incremented (or decremented, with the real-time user controls), the tonal center shifts along the Circle of Fifths, and is broadcast throughout the Pd patch as the variable *tonicnote*. The various scale-definition patches use this value to adjust the actual value of the MIDI note numbers they send out (while the relative MIDI note numbers remain the same).

The bass voice, included as a harmonic reference for the ear, is simply the tonic note calculated down two octaves, and then sent to both channels. Its intensity is governed by z just like the other two voices. It serves as a reference pitch to remind the user of the current tonal center and show very clearly how the tonal center is moving.

In addition to sending the audio out using the digital-to-analog converter in Pd, the audio can also be sent out to a MIDI synthesizer. Voice 1 is sent to channel 1, Voice 2 is sent to channel 3, and the bass voice is sent to channel 2. The velocity of the notes are modified in the same manner as the volume of the audio. The user is given the flexibility of choosing the actual MIDI instruments on the device itself, to correlate with the music in the way that the user sees fit.

The implementation of the algorithms really shows in the music. After listening to examples of the output, a listener can gradually identify the path of the user through

the music. The latitude and longitude are mapped to the note choice, which varies in an interesting manner. The changes in latitude and longitude, which relate to the east-west and north-south speed, alter the rhythms in such a way that when the user is moving faster, the notes change more frequently, and when the user is moving slower, the notes change less frequently. They are also sent to different channels so you can distinguish between them. When the user is going up and over hills, the music moves much faster through the Circle of Fifths than usual. It's possible that with some training, a user could mentally recreate the path by listening to the music.

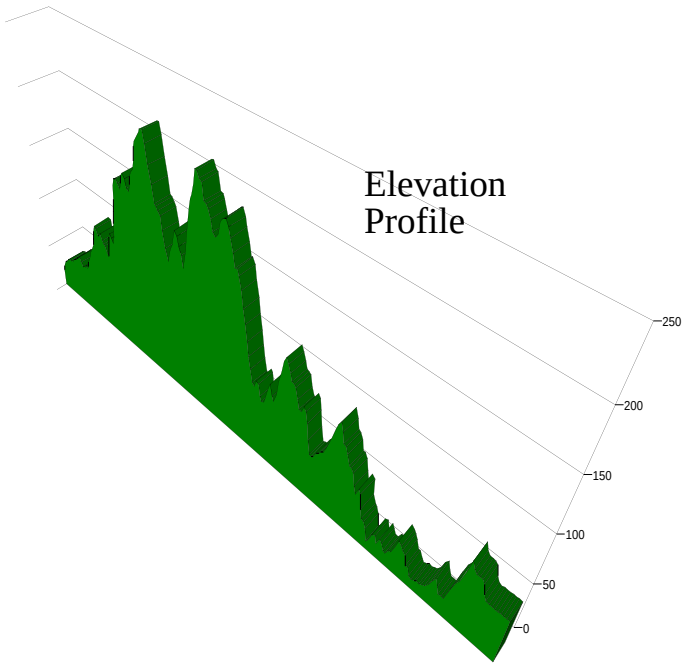
For more information, see the file *pd_patch_images.pdf*, which contains the Pd patch and all associated subpatches used in the project.

Appendix A: Eightfold Path

Our project addressed the eight concerns for good design that were brought up in class as depicted in this diagram: http://www-ccrma/courses/250a/projects/briankirstensaif.gif

One difference is the representation of *display*. We expanded on our initial ideas of how to incorporate display and feedback significantly, through the use of the graphical user interface and the enclosure for the physical user interface. The provide much more substantial visual feedback than indicated in the diagram.

Appendix B: Data Visualization



Appendix C: GPS Data-Reformatting C Code

```
/* File: analyze.c
 * ---------------
 * GPS Music Group
 * This program takes a gps data file as its input.  It also allows
 * you to send it parameters for reversing the file and for the total
 * length of the file it will output (by specifying the number of
 * seconds the ultimate piece of music will be).  The output is a file
 * containing all the relevant information needed for producing music
 * with the gps board.
 */

#include <math.h>       // abs, sqrt, mod, etc
#include "scanner.h"    // read token commands
#include <assert.h>     // for assert statements (similar to 'error')
#include <stdlib.h>     // for malloc
#include <string.h>     // for strcasecmp
#include <limits.h>     // for PATH_MAX
#include <ctype.h>      // for isascii


//in delimiters: took out the period, - and / from the basic list of
//delimiters, and added T, N, W, E, S
#define DELIMITERS " \t\n\r,!\"(){};:?\\*^%$#@&[]=<>~`|+<>TNEWS"
#define MAX_WORD_LEN 100
#define MAX_POINTS_IN_LIST 10000
#define MAX_TIME_DIFFERENCE 300
#define NUM_BASS_NOTES 8
#define NUM_DRUM_NOTES 8

/* Type: struct gpspoint and gpslist
 * ---------------------------------
 * The gpslist structure is just a simple fixed-size array of strings packaged with
 * its current length (i.e. number of entries in use). Since you always need
 * the two together, it's convenient and tidy to put them in one structure.
 * Gpslist also remembers the maximum values for latitude, longitude,
 * and elevation so it can write these to a file for the pd patch.
 * The gpspoint structure stores all the possibly useful information
 * gleaned from the gps data file.  Uses all floats with the
 * exception of time and date so we will not need to type-cast later
 * on when doing calculations.
 */
struct gpslist {
  gpspoint points[MAX_POINTS_IN_LIST];
  int count;
  float maxdlat, maxdlong, maxdelev;
};

typedef struct gpspoint {
  float x, y, z, dlat, delev, dlong;
  float velolat, velolong, acclat, accelev, acclong, veloelev;
  float xyveloc, angle, dist, dt;
  char *date;
  int t;
} gpspoint;

/* Functions */
static void ReadInGPS(const char *infilename, const char *argument1, const char
*argument2);
static void ExtractPointsIntoList(Scanner s, struct gpslist *list);
static void PrintList(struct gpslist *list, bool isReverse, int usedSamples);
static void FreeList(struct gpslist *list);
static char *CopyString(const char *s);
static void ProcessAllValues(struct gpslist *list);
static float LatVeloc (struct gpslist *list, int n1, int n2);
static float LongVeloc (struct gpslist *list, int n1, int n2);
static float ElevVeloc (struct gpslist *list, int n1, int n2);
```

```
static float LatAccel (struct gpslist *list, int n1, int n2);
static float LongAccel (struct gpslist *list, int n1, int n2);
static float ElevAccel (struct gpslist *list, int n1, int n2);
static float Speed(struct gpslist *list, int n1, int n2);
static float Angle(struct gpslist *list, int n1, int n2);
static float Distance(struct gpslist *list, int n1, int n2);
static float DLatitude(struct gpslist *list, int n1, int n2);
static float DLongitude(struct gpslist *list, int n1, int n2);
static float DElevation(struct gpslist *list, int n1, int n2);
static int DTime(struct gpslist *list, int n1, int n2);
static void InterpolateAndPrint(struct gpslist *list, bool isReverse, int usedSamples);
static float GetInterValue(float n1, float n2, float scalar);


/* Main */

int main(int argc, char *argv[])
{
    if (argc < 2) {
        printf("\nThis program takes at least one argument which is a filename\n");
               printf("containing the gps data you wish to analyze. Please try
again!\n\n");
    } else {
        ReadInGPS(argv[1], argv[2], argv[3]);
    }
    return 0;
}


/* ReadInGPS
 * ---------------
 * We try to create a Scanner and if the call returns null, that
 * means the file didn't exist or couldn't be opened (usually because
 * of protection) and we just return.  Then we parse the file for all
 * the data and send that info to be interpolated so that the
 * time samples are equal, and processed (calculating velocity
 * and such) then send it to be printed.
 */

static void ReadInGPS(const char *infilename, const char
*argument1, const char *argument2)
{
    Scanner s = NULL;
    bool isReverse = 0;
    int desiredTime = 0;
    int usedSamples = 0;
    struct gpslist list;
    char point[MAX_WORD_LEN];

    //looks just at each line (for the header)
    s = NewScannerFromFilename(infilename, DELIMITERS, true);
    assert(s != NULL);
    ReadNextToken(s, point, MAX_WORD_LEN);
    SkipUntil(s, "D");
    ReadNextToken(s, point, MAX_WORD_LEN);
    SkipUntil(s, "t");

    if (s != NULL) {
      ExtractPointsIntoList(s, &list);
      FreeScanner(s);
    }

    if(argument1 != NULL) {
        if(argument1[0] == 'r' || (argument2 != NULL && argument2[0] == 'r'))
               isReverse = 1;
         if(argument1 != NULL && argument1[0] != 'r')
               desiredTime = strtod(argument1, (char**)NULL);
        if(argument2 != NULL && argument2[0] != 'r')
               desiredTime = strtod(argument2, (char**)NULL);
    }
```

```
    usedSamples = (int) (1000.0*(desiredTime/160.0));

    InterpolateAndPrint(&list, isReverse, usedSamples);

    FreeList(&list);
}

/* InterpolateAndPrint
 * -------------------
 * Interpolates between all the points so that the spaces between
 * samples are even (1 second to be exact).
 */

static void InterpolateAndPrint(struct gpslist *list, bool isReverse, int
usedSamples)
{
    int i, j, startt, scalar_num, t1, t2;
    struct gpslist newlist;
    float pos_scalar, dt;

    assert(list != NULL);

    newlist.count = 0;
    startt = list->points[0].t;

    for (i = 0; i < list->count-2; i++) {
      t1 = list->points[i].t;
      t2 = list->points[i+1].t;
      dt = (float) DTime(list, i+1, i);
      if (dt > MAX_TIME_DIFFERENCE) {
        list->count = i;
        break;
      }
      scalar_num = 0;
      for (j = t1-startt; j < t2-startt; j++) {
        newlist.count++;
        pos_scalar = scalar_num/dt;
        newlist.points[j].date = CopyString(list->points[i].date);
        newlist.points[j].t = (int)
           GetInterValue((float)(list->points[i].t), (float)(list->points[i+1].t),
pos_scalar);
        newlist.points[j].x =
           GetInterValue(list->points[i].x, list->points[i+1].x, pos_scalar);
        newlist.points[j].y =
           GetInterValue(list->points[i].y, list->points[i+1].y, pos_scalar);
        newlist.points[j].z =
           GetInterValue(list->points[i].z, list->points[i+1].z, pos_scalar);
        scalar_num++;

     }
   }

    ProcessAllValues(&newlist);

    PrintList(&newlist, isReverse, usedSamples);

    FreeList(&newlist);
}


/* GetInterValue
 * -------------
 * This is just a helper function to return whatever value lies
 * between the two points' values.
 */

static float GetInterValue(float n1, float n2, float scalar)
{
    return (n1 + scalar * (n2 - n1));
}
```

```
/* ExtractPointsIntoList
 * --------------------
 * This uses the Scanner to extract the words from the file, by looping
 * calling ReadNextToken until it returns false (which indicates end of
 * file).
 * We use the stack for the storage for the point (again since the stack
 * is cheap and quick), but if we need to store the string permanently
 * in the list, we have to make a new heap copy, since the stack buffer
 * will be overwritten each time we read a new token. This copy is made
 * when adding the point to the list.  It is used only for "date".
 */
static void ExtractPointsIntoList(Scanner s, struct gpslist *list)
{
    char point[MAX_WORD_LEN];
    int i;
    float hour, minute, second;

    i = 0;

    while (ReadNextToken(s, point, MAX_WORD_LEN)) { //t
      if (strcmp(point, "n") == 0) break; //end of file
         ReadNextToken(s, point, MAX_WORD_LEN); //d

        ReadNextToken(s, point, MAX_WORD_LEN);//reads in the Latitude
        list->points[i].x = strtod(point, (char**)NULL);

        ReadNextToken(s, point, MAX_WORD_LEN); //reads the Longitude
        list->points[i].y = strtod(point, (char**)NULL);

        ReadNextToken(s, point, MAX_WORD_LEN); //reads the date
        list->points[i].date = CopyString(point);

        ReadNextToken(s, point, MAX_WORD_LEN); //reads the hours
        hour = strtod(point, (char**)NULL);
        ReadNextToken(s, point, MAX_WORD_LEN); //reads the minutes
         minute = strtod(point, (char**)NULL);
        ReadNextToken(s, point, MAX_WORD_LEN); //reads the seconds
         second = strtod(point, (char**)NULL);

        // forcing the time to be in seconds...  otherwise it's a
        // fraction of a day (24 hours).

        list->points[i].t = (hour*60*60 + minute*60.0 + second);

        ReadNextToken(s, point, MAX_WORD_LEN); //reads the Altitude
        list->points[i].z = strtod(point, (char**)NULL);

        ReadNextToken(s, point, MAX_WORD_LEN); //reads last number (?)

        i++;
    }

    list->count = i;

}

/* PrintList
 * ---------
 * Prints the gps list for the gps pd patch to read from.  Do not
 * change the names without changing the pd patch as well.
 */
static void PrintList(struct gpslist *list, bool isReverse, int usedSamples)
{
    int i, start, end, inc, npoints, counter;
    gpspoint p;
    int skipped_samples = 0;

    // determine what the skipped samples interval should be in order to
    // get as close as possible to the requested number of used samples.
    if(usedSamples != 0) {
```

```
        skipped_samples = ((float)(list->count)) / (float)usedSamples;
    }

    // make sure that if not initialized by user, or if user put in
    // too high a sample usage, that it increments by 1 (avoids infinite
    // loops)
    if(skipped_samples == 0) skipped_samples = 1;

    //first line to tell how long in sec the sample is, ie how many points
    npoints = ceil(((float)list->count)/((float)skipped_samples));

    printf("npoints %d; maxdlat %f; maxdlong %f; maxdelev %f;\n",
       npoints, list->maxdlat, list->maxdlong, list->maxdelev);

    //are we reversing the data?  if so, change the read direction
    if (!isReverse) {
        start = 0;
        end = list->count;
        inc = skipped_samples;
    } else {
        start = list->count - 1;
        end = -1;
        inc = -skipped_samples;
    }

    i = start;
    counter = 0;

    while (i >= 0 && i < list->count) {
        printf("1 pointnum %d; ", counter); //indexes from 0

        p = list->points[i];
        // leading 1 for delay, but really it's just a placeholder
        printf("north %f; ", p.x);
        printf("east %f; \n", p.y);
        printf("date %s; ", p.date);
        printf("time %d; ", p.t);
        printf("dt %f; ", p.dt);
        printf("elevation %f; ", p.z);
        printf("dlat %f; ", p.dlat);
        printf("delev %f; ", p.delev);
        printf("velolat %f; ", p.velolat);
        printf("velolong %f; ", p.velolong);
        printf("acclat %f; ", p.acclat);
        printf("acclong %f; ", p.acclong);
        printf("accelev %f; ", p.accelev);
        printf("dlong %f; ", p.dlong);
        printf("veloelev %f; ", p.veloelev);
        printf("xyveloc %f; ", p.xyveloc);
        printf("angle %f; ", p.angle);
        printf("xydist %f; \n", p.dist);
        i += inc;
        counter++;
    }

}



/* ProcessAllValues
 * ----------------
 * Uses the algorithms for converting the data into
 * angle, velocity, acceleration, etc. and the stores the values.
 */

static void ProcessAllValues(struct gpslist *list)
{
        int i;

        //initialize max for y, x, and z
        list->maxdelev = 0;
```

```
        list->maxdlong = 0;
        list->maxdlat = 0;

        for (i = 0 ; i < list->count ; i++) {
                list->points[i].dist = Distance(list, i, i-1);
                list->points[i].xyveloc = Speed(list, i, i-1);
                list->points[i].angle = Angle(list, i, i-1);

                list->points[i].dlat = DLatitude(list, i, i-1);
                if (fabs(list->points[i].dlat) > list->maxdlat)
                        list->maxdlat = fabs(list->points[i].dlat);

                list->points[i].delev = DElevation(list, i, i-1);
                if (fabs(list->points[i].delev) > list->maxdelev)
                        list->maxdelev = fabs(list->points[i].delev);

                list->points[i].velolat = LatVeloc(list, i, i-1);
                list->points[i].velolong = LongVeloc(list, i, i-1);
                list->points[i].acclat = LatAccel(list, i, i-1);
                list->points[i].acclong = LongAccel(list, i, i-1);
                list->points[i].accelev = ElevAccel(list, i, i-1);

                list->points[i].dlong = DLongitude(list, i, i-1);
                if (fabs(list->points[i].dlong) > list->maxdlong)
                        list->maxdlong = fabs(list->points[i].dlong);

                list->points[i].veloelev = ElevVeloc(list, i, i-1);
                list->points[i].dt = DTime(list, i, i-1);

        }

}


/* LOTS OF LITTLE HELPER FUNCTIONS */


/* Changes in Dimension Functions */

static float DLatitude(struct gpslist *list, int n1, int n2)
{
        if (n1 < 0 || n2 < 0) return 0;
        return(list->points[n1].x - list->points[n2].x);
}

static float DLongitude(struct gpslist *list, int n1, int n2)
{
        if (n1 < 0 || n2 < 0) return 0;
        return(list->points[n1].y - list->points[n2].y);
}

static float DElevation(struct gpslist *list, int n1, int n2)
{
        if (n1 < 0 || n2 < 0) return 0;
        return(list->points[n1].z - list->points[n2].z);
}

static int DTime(struct gpslist *list, int n1, int n2)
{
        if (n1 < 0 || n2 < 0) return 0;
        return(list->points[n1].t - list->points[n2].t);
}


/* Speed
 * -----
 * In X, Y plane, not just latitude and longitude
 */

static float Speed(struct gpslist *list, int n1, int n2)
{
```

```
        float speed;

        if (n1 < 0 || n2 < 0) return 0;
        speed = Distance(list, n1, n2) /
        (list->points[n1].t - list->points[n2].t);

        return speed;
}


/* Distance
 * --------
 * In X, Y plane.  Not just latitude or longitude.
 */

static float Distance(struct gpslist *list, int n1, int n2)
{
        float dist;

        if (n2 < 0 || n1 < 0) return 0;
        dist = sqrt(
        (list->points[n1].x - list->points[n2].x) *
        (list->points[n1].x - list->points[n2].x)
        + (list->points[n1].y - list->points[n2].y) *
        (list->points[n1].y - list->points[n2].y));

        return dist;
}



/* Angle
 * -----
 * In X, Y plane
 */

static float Angle(struct gpslist *list, int n1, int n2)
{
        double angle;

        if (n2 < 0 || n1 < 0) return 0;
        if (Distance(list, n1, n2) == 0) return 0;
        angle = asin((double)((list->points[n1].x - list->points[n2].x))
        / Distance(list, n1, n2));

        if (list->points[n1].y < 0) angle = 180 - angle;
        if (list->points[n1].x < 0) angle = angle + 360;

        return angle;
}


/* Velocity Functions */

static float LatVeloc(struct gpslist *list, int n1, int n2)
{
        if (n1 < 0 || n2 < 0) return 0; // can't take previous sample
        if (DTime(list, n1, n2) == 0) return 0;
        return(DLatitude(list, n1, n2)/DTime(list, n1, n2));
}
static float LongVeloc(struct gpslist *list, int n1, int n2)
{
        if (n1 < 0 || n2 < 0) return 0; // can't take previous sample
        if (DTime(list, n1, n2) == 0) return 0;
        return(DLongitude(list, n1, n2)/DTime(list, n1, n2));

}
static float ElevVeloc(struct gpslist *list, int n1, int n2)
{
        if (n1 < 0 || n2 < 0) return 0; // can't take previous sample
        if (DTime(list, n1, n2) == 0) return 0;
```

```
        return(DElevation(list, n1, n2)/DTime(list, n1, n2));
}


/* Acceleration Functions */

static float LatAccel(struct gpslist *list, int n1, int n2)
{
        if (n1 < 0 || n2 < 0) return 0;
        if (DTime(list, n1, n2) == 0) return 0;
        return ((LatVeloc(list, n1, n1-1) - LatVeloc(list, n2, n2-1)) /
          DTime(list, n1, n2));
}

static float LongAccel(struct gpslist *list, int n1, int n2)
{
        if (n1 < 0 || n2 < 0) return 0;
        if (DTime(list, n1, n2) == 0) return 0;
        return ((LongVeloc(list, n1, n1-1) - LongVeloc(list, n2, n2-1)) /
         DTime(list, n1, n2));
}

static float ElevAccel(struct gpslist *list, int n1, int n2)
{
        if (n1 < 0 || n2 < 0) return 0;
        if (DTime(list, n1, n2) == 0) return 0;
        return ((ElevVeloc(list, n1, n1-1) - ElevVeloc(list, n2, n2-1)) /
         DTime(list, n1, n2));
}



/* FreeList
 * ---------
 * Frees all the dynamically-allocated strings in the gps list.  This
 * should just be "date".
 */
static void FreeList(struct gpslist *list)
{
    int i;

    for (i = 0; i < list->count; i++) {
         free(list->points[i].date);
    }
    free(list);
}

/* CopyString
 * ----------
 */

static char *CopyString(const char *s)
{
  char *copy = malloc(strlen(s) + 1);
  assert(copy != NULL);
  strcpy(copy, s);
  return copy;
}
```

Appendix D: Atmel code in C for User Interface

```
//*************************************
//*************************************
//
//  GPS MUSIC
//  For avrlib and avrmini development board.
//
//  File:     gps-board.c
//  Author:   gps music group
//
//
//*************************************
// This program takes the potentiometer sensor data from
// the gps control board and passes the information
// to the pd patch in the form of midi messages.
//
//*************************************

#include <io.h>
#include <progmem.h>
#include "global.h"
#include "uart.h"
#include "midi.h"
#include "timer.h"
#include <interrupt.h>
#include <sig-avr.h>
#include "a2d.h"
#include "uart.h"


int main(void)
{

  // General Initializations
  timerInit();
  uartInit();
  midiInit();

  // A to D initializations
  a2dInit();
  a2dSetReference(0x01);

  sei();                    // enable interrupts
  outb(DDRA, 0x00);
  outb(DDRD, 0x00);
  outb(PORTA, 0xFF);

  u08 pa0, pa1, oldpa2, pa2, pa3, pa4, pa5;      //temp storage

        // loop forever
        while(1) {

          timerPause(10);
          pa0 = a2dConvert8bit(0)/2;      // read the a/d on PortA pin 1
          midiNoteOnOut(pa0, pa0,  0);

          timerPause(10);
          pa1 = a2dConvert8bit(1)/2;      // read the a/d on PortA pin 2
          midiNoteOnOut(pa1, pa1, 1);

          timerPause(10);
          pa2 = a2dConvert8bit(2)/2;      // read the a/d on PortA pin 3

           // check whether the sensor assigned to chord changes
           // is going up or down or staying the same.  if it's going
           // up send it 0 - converted to -1 in pd patch
           // if it's going down, send out 2 - converted to 1 in the
```

```
          // pd patch.  This is because midi cannot send a negative
          // value, so the pd patch subtracts one from the msg.

          if (pa2 > oldpa2) midiNoteOnOut(0, 0, 2);
          else if (pa2 < oldpa2) midiNoteOnOut(2, 2, 2);
          else midiNoteOnOut(1, 1, 2);
          oldpa2 = pa2;

          timerPause(10);
          pa3 = a2dConvert8bit(3)/2;      // read the a/d on PortA pin 4
          midiNoteOnOut(pa3, pa3, 3);

          timerPause(10);
          pa4 = a2dConvert8bit(4)/2;      // read the a/d on PortA pin 5
          midiNoteOnOut(pa4, pa4, 4);

          timerPause(10);
         pa5 = a2dConvert8bit(5)/2;      // read the a/d on PortA pin 6
         midiNoteOnOut(pa5, pa5, 5);

          timerPause(10);
          // check for whether the pause button is pressed.
          if(! bit_is_set(PIND, 2)) {
                midiNoteOnOut(1, 1, 6);
          } else { midiNoteOffOut(0, 0, 6); }

        }

        return 0;
}
```