Tutorial and Examples on Pure Data Externals: Real-time Audio Signal Processing and Analysis

Sakari Tervo and Jukka Pätynen
Department of Media Technology
Aalto University School of Science and Technology

1 Introduction

Pure Data (PD) is a graphical programming environment suitable for quickly developing processing methods for audio, image and video. While PD contains a wide collection of building blocks, complex processing with built-in objects and graphical connections is not sensible. However, PD contains the possibility of writing new objects with C/C++ programming language. Such objects are called *externals*. In this document, we present a tutorial for writing an PD external for real-time multi-channel audio analysis.

The implementation of the examples shown in this tutorial can be found from `www.tml.tkk.fi/~tervos/PDExternals.zip`. All of the source code is also shown here, however some details may vary from original.

Regarding the principles on programming PD externals, Johannes M. Zmölnig has written a *how-to* tutorial discussing the basic functions and structure of an external [2]. Few elementary coding examples are provided. Understanding the contents of those examples is recommended. This document functions as an extension for the original tutorial.

Three examples of real time signal processing applications are given. First of the three applications is performing Mel-Cepstral Coefficient (MFCC) calculation. These coefficients are useful in many speech processing applications, e.g., speech recognition.

The second one calculates sound intensity vectors from a grid of six microphones. In short, sound intensity is a quantity which describes the instantaneous

direction and magnitude of propagating sound. For instance, sound intensity analysis can be used for tracing acoustic reflections in a space. Therefore such approach is often beneficial in studying rooms where the acoustics has a great importance, such as concert halls. The current implementation performs the analysis in 3-D for estimating both azimuth and elevation.

The third application computes 1 dimensional histogram. That is, the occurrence of numbers in a data vector with respect to their value.

The theoretical basis of functions implemented here are given in section 3. The implementation (the source code) as well as comments of it are given in section 4.

2 Compiling

Compiling of the third-party externals requires an environment setup for the required libraries and header files. The downloadable packages include the source code and the required files for easy compiling. It should be noted that the externals files should be located in the PD search path in order to being found in PD.

Modification and recompiling the provided externals can be done with the following instructions. For Windows 32-bits the executables of the externals in their most recent form are provided. For Linux 32-bits the compilation is not provided, that is, they must be compiled in order to use them in PD. The instructions for the compilation in Linux follow in the next subsection.

An example where all the three applications are in use is shown in `pd-external-example.pd`. If the compilation is done correctly that example should run.

2.1 Linux

Linux installations provide required libraries when PD is installed with modern package management methods. Firstly open the `Makefile` from the package [1]. Then modify line 22 so that PDPATH has the path of your PD executable.

```
PDPATH=<!path of your pd!>
```

Then compile each external with these commands

```
> make clean
> make
> pd <!your pd file name!>.pd
```

The last row starts up PD with your external.

[1] The authors wish to thank M.Sc. Raine Kajastila for providing the Makefile.

2.2 Win32

While compiling the externals in MS Visual Studio family packages is possible, a Mingw32 environment is used in these examples for simplicity. Comprehensive description of the Mingw32 and Msys setup in Windows systems is written by Alberto Zin [4]. Requirements for successful setup consist of PD libraries and header files, installed Mingw32 environment and Msys command interpreter. Here a Makefile is provided for the automated compilation of individual externals. The available Makefiles should be adapted for suitable target location.

In a correctly installed Msys environment an external can be compiled with command

```
> make -f Makefile.win
```

3 Theory

3.1 Mel Cepstral Coefficients

Mel Cepstral Coefficients are calculated from short time windows with the following procedure. Firstly, the Discrete Fourier Transform (DFT) is applied for the signal x_n:

$$X_w = \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i}{N} kn} \qquad k = 0, \ldots, K-1 \tag{1}$$

where $k = 0 \ldots K-1$ are the discrete frequencies and n is the discretized time index. The signal in the discrete frequency domain is then divided into frequency bands that are of equivalent length in a so-called mel-scale frequencies. The conversions from linear to mel-scale and back are defined as:

$$f = 700(e^{m/1127.01048-1)}, \tag{2}$$

and

$$m = 1127.01048 \log\{1 + (f/2)/700\}, \tag{3}$$

where f is the frequency and m is the mel-frequency. Then, the energies on each frequency band are calculated and weighted with triangular filter in the frequency domain:

$$E_k = \sum_{k=k_1}^{k_2} \log_{10}\{|X_k|^2\} W_{\text{tri}}^k, \tag{4}$$

where W_{tri}^{k} is the triangular filter. The final MFCCs are given as the Discrete Cosine Transform (DCT) of the energy signal E_k

$$\text{MFCC}_w = \sum_{k=0}^{K-1} E_k \cos\left[\frac{\pi}{K}\left(n+\frac{1}{2}\right)w\right] \qquad w = 0, \ldots, K-1 \tag{5}$$

where w is another "frequency" index. It should be noted that the DCT can be implemented through DFT. This is possible since DFT includes the components needed for DCT calculations. The conversion from DFT to DCT can be done by multiplying the real and imaginary part of the DFT with proper cosine and sine terms:

$$\text{D}_k = \cos(-k\pi)\Re\{X_k\} - \sin(-k\pi)\Im\{X_k\}, \tag{6}$$

where $\Re\{\cdot\}$ and $\Im\{\cdot\}$ are the real and imaginary part of a complex number, respectively.

For more information on MFCCs the reader is referred to e.g. [3].

3.2 Sound intensity vector calculation and mean direction estimation

On a certain axis x, the sound intensity is given in the frequency domain as

$$I_x(\omega) = \Re\{P^*(\omega)U_x(\omega)\}, \tag{7}$$

where $P(\omega)$ and $U_x(\omega)$ are the frequency presentations of the sound pressure and of the particle velocity with angular frequency ω [1]. In addition, $\Re\{\cdot\}$ is the real part of a complex number and $(\cdot)^*$ denotes the complex conjugate.

The pressure in the middle of the array, shown in Fig. 1, can be estimated as the average pressure of the microphones [1]:

$$P(\omega) \approx \frac{1}{6}\sum_{n=1}^{6} P_n(\omega). \tag{8}$$

In the frequency domain the particle velocity is estimated for x-axis as:

$$U_{\text{x}}(\omega) \approx \frac{-j}{\omega\rho_0 d}[P_1(\omega) - P_2(\omega)], \tag{9}$$

where d is the distance between the two receivers, j is the imaginary unit, and for example with the speed of sound $c = 343$ m/s the median density of the air is $\rho_0 = 1.204$ kg/m^3.

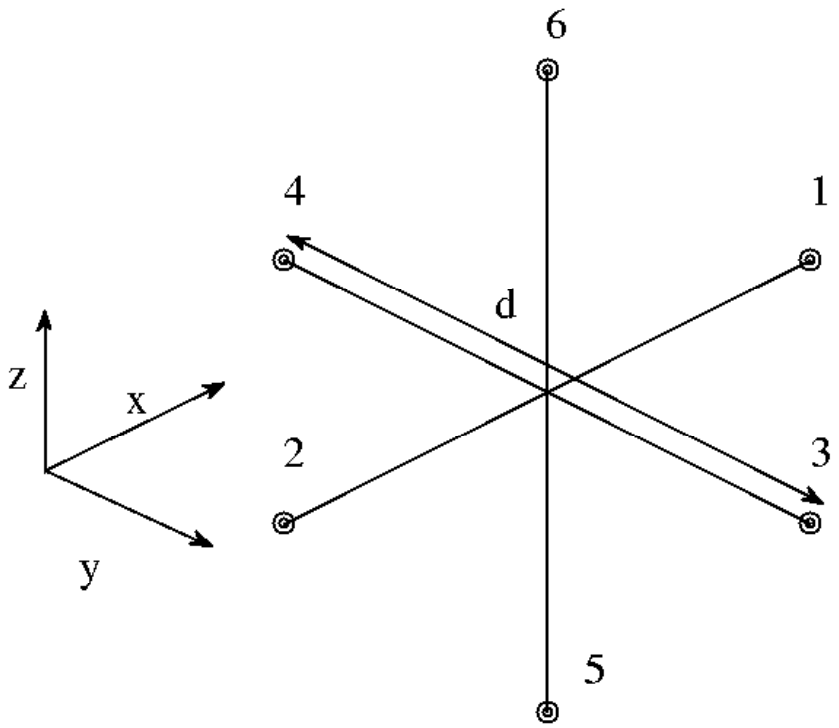

Figure 1: 3-D microphone array suitable for sound intensity estimation. Spacing d is equal between two microphones on a single axis.

The sound intensity in (7) is estimated with the approximations in (8) and (9). For obtaining the y and z -component of the sound intensity, the microphones 1 - 2, are replaced in (9) with microphones 3 - 4, and 5 - 6, respectively. The sound intensity is then noted with $\boldsymbol{I}(\omega) = [I_x(\omega), I_y(\omega), I_z(\omega)]$.

The conversion from Cartesian (x,y,z) to spherical coordinate (r, θ, phi) system can be done using basic trigonometric equations. The azimuth angle is calculated as:

$$\theta_k = \arctan_2\{I_y(k)/I_x(k)\} \tag{10}$$

where $\arctan_2\{\cdot\}$ is the four quadrant inverse tangent. The elevation angle is given as:

$$\phi_k = \arccos\left\{\frac{I_z(k)}{\sqrt{I_x(k)^2 + I_y(k)^2 + I_z(k)^2}}\right\} \tag{11}$$

where $\arccos\{\cdot\}$ is the inverse cosine.

The mean direction of the sound intensity vectors can be estimated as

$$\hat{\boldsymbol{I}} = \frac{S}{\|S\|}, \tag{12}$$

where

$$S = \sum_{k=k_1}^{k_2} \boldsymbol{I}(k) \tag{13}$$

is the sum of all the individual vectors over the selected discretized frequency band from k_1 to k_2. The conversion to spherical coordinate system can be done similarly as previously. Another possibility is to calculate the mean direction directly using the sound intensity vectors in the spherical coordinate system. Then the mean direction is given as:

$$\hat{\theta} = \arg\left\{ \sum_{k=k_1}^{k_2} e^{j\theta_k} \right\}, \tag{14}$$

and

$$\hat{\phi} = \arg\left\{ \sum_{k=k_1}^{k_2} e^{j\phi_k} \right\}, \tag{15}$$

where $\arg\{\cdot\}$ is the argument of a complex number. The latter option for the direction estimation is used in the implementation.

3.3 Histogram

One dimensional histogram is the number of events within the given limits. It is defined as

$$h_n = \sum_{n=0}^{N-1} \delta(x_n), \tag{16}$$

where $\delta(x_n) = 1$ if the current data value is between the limits, and 0 otherwise.

4 Implementation as PD Externals

4.1 Mel-Cepstral Coefficients

```
#include <stdio.h>
#include <math.h>
#include "m_pd.h"

static t_class *MFCC_tilde_class;

typedef struct _MFCC_tilde {
  t_object x_obj;
  t_sample f_MFCC;
  t_sample f;
  int numOfBands;
} t_MFCC_tilde;

const double PI = 3.1415926538;
```

The above code defines the dataspace used by MFCC-object. Only few variables are required, as the object does not need to store data in runtime.

The following setup method is similar as presented in the PD external tutorial [2].

```
void MFCC_tilde_setup(void) {
  MFCC_tilde_class = class_new(gensym("MFCC~"),
        (t_newmethod)MFCC_tilde_new,
        0, sizeof(t_MFCC_tilde),
        CLASS_DEFAULT,
        A_DEFFLOAT, 0);

  class_addmethod(MFCC_tilde_class,
        (t_method)MFCC_tilde_dsp, gensym("dsp"), 0);
  CLASS_MAINSIGNALIN(MFCC_tilde_class, t_MFCC_tilde, f);
}
```

The following constructor method has the optional creation argument for desired number of MFCC bands. Default is 50. Separate signal inlets are required for real and imaginary parts from FFT.

```
void *MFCC_tilde_new(t_floatarg f)
{
  t_MFCC_tilde *x = (t_MFCC_tilde *)pd_new(MFCC_tilde_class);
  x->numOfBands = (f)?f:50; /* defaults to 50 */

  post("Number of MFCC bands: %d", x->numOfBands);

  inlet_new(&x->x_obj, &x->x_obj.ob_pd, &s_signal, &s_signal);
  outlet_new(&x->x_obj, &s_signal);

  return (void *)x;
}
```

The following part implements the DFT as defined in (1).

```
void dft(double *x, double *y, double *r, double *i, int N)
{
   int   n, k;
```

```
   double c, s, p, x0, y100;

   for(k = 0; k < N ; k++)
   {
      x0 = 0;
      y100 = 0;

      for(n = 0; n < N; n++)
      {
         p  = -2*PI*k*n/N;
         c  = cos(p);
         s  = sin(p);
         /*x0 = x0 + c*x[n] - s*y[n];
         y0 = y0 + c*y[n] + s*x[n];*/
         x0 = x0 + c*x[n];
         y100 = y100 + s*x[n];

      }
      r[k] = x0;
      i[k] = y100;
   }
}
```

Here the Mel-Cepstral Coefficients are computed as given in (2) – (6). Mel-frequency mapping is performed first. Second is the computation of energies on the mel-band with triangular weighting. Source code for DCT is included. DCT could also be implemented in a separate function.

```
void MFCC(double *MFCCoeffs, double *xr, double *xi, int N, t_MFCC_tilde *x)
{

    /* Variables and constants */
    int numOfBands = x->numOfBands;
    double fs = 48000;
    double mel_f, linear_f, tri;
    double fftlen = N;

    int i, k;
    double d1 = 0;
    double d2 = 1127.01048*log(1+(fs/2)/700);
    int fft_ind[numOfBands];
    double P[numOfBands*2], wr, wi, FCr[numOfBands*2],FCi[numOfBands*2];


    /* frequency mappings, see Wikipedia, Mel Scale */
    for(i = 0; i < numOfBands+2; i++)
    {
        mel_f    = d1 + i*(d2-d1)/(floor(numOfBands+2-1)); /* mel */
        linear_f = 700*(exp(mel_f/1127.01048)-1);/* Hz */
        fft_ind[i] = (int)floor(fftlen*linear_f/fs+0.5);

    }

    /* Spectral power */
    for(i = 0;i < numOfBands;i++)
    {
```

```
        P[i] = 0;
        for(k = fft_ind[i]; k < fft_ind[i+2];k++)
        {
        /* Create the triangluar filter in frequency domain */
            if( k < fft_ind[i+1])
            {
                tri = 1+((double)k-fft_ind[i+1])*1/
        ((double)fft_ind[i+1] - (double)fft_ind[i]);
            }
            else
            {
                tri = 1 + ((double)k-fft_ind[i+1])*(-1)/
        ((double)fft_ind[i+2] - (double)fft_ind[i+1]);
            }

            P[i] = P[i] + (pow(xr[k],2) + pow(xi[k],2))*tri;
        }
        P[i] = log10(P[i] + 0.00001);
        /* for dct */
        P[2*numOfBands-(i+1)] = P[i];


    }

    /* DCT */
    dft(P, P, FCr, FCi, numOfBands*2);

    for(i = 0; i < numOfBands; i++)
    {
        /* DCT weights */
        wr = (cos(-(i)*PI/(2*numOfBands))/pow(2*numOfBands,.5));
        wi = (sin(-(i)*PI/(2*numOfBands))/pow(2*numOfBands,.5));

        if( i == 0)
        {
            wr = wr / pow(2,.5);
            wi = wi / pow(2,.5);
        }
        /* only the real part is interesting since input is real */

        MFCCoeffs[i] = FCr[i] * wr  - wi * FCi[i];
    }

}

t_int *MFCC_tilde_perform(t_int *w)
{

  t_MFCC_tilde  *x  = (t_MFCC_tilde *)(w[1]);
  t_sample  *in1 = (t_sample *)(w[2]);
  t_sample  *in2 = (t_sample *)(w[3]);

  t_sample  *out = (t_sample *)(w[4]);
  int        n   = (int)(w[5]);

  double xr[n], xi[n];
  int numOfBands = x->numOfBands;
```

```
  double MFCCoeffs[n];

  int i;
  for (i = 0;i < n;i++)
  {
    /* input */
    xr[i] = (double) in1[i];
    xi[i] = (double) in2[i];
    MFCCoeffs[i] = 0.0;

  }

    MFCC(MFCCoeffs, xr, xi, n, x);

    for (i = 0;i < numOfBands;i++)
    {
    out[i] = (t_sample) MFCCoeffs[i];
    }

    /* fill the rest of the block beyond 25 MFCC bands with zeros */
    for (i = numOfBands;i < n;i++)
    {
    out[i] = (t_sample) 0.0;
    }

    out[0] = (t_sample) 0.0; /* remove the first coefficient */

  return (w+6);

}

void MFCC_tilde_dsp(t_MFCC_tilde *x, t_signal **sp)
{
    dsp_add(MFCC_tilde_perform, 5, x,
          sp[0]->s_vec, sp[1]->s_vec, sp[2]->s_vec, sp[0]->s_n);
}
```

Full source code for `MFCC` is included in the code packages.

4.2 Sound intensity vector calculation

The class `calcIntensityVector~` and the dataspace (internal variables) are declared as follows:

```
static t_class *calcIntensityVectors_tilde_class;
typedef struct _calcIntensityVectors_tilde {
  t_object x_obj;
  double f;
  double dist;
  int FootballFlag;
  t_outlet *f_az_out;
  t_outlet *f_el_out;
} t_calcIntensityVectors_tilde;

const double PI = 3.1415926538;
```

t_object `x_obj` is required as the principal object container. The variable double `dist` is for saving the distance between the opposing elements in the microphone array. The variable int `FootballFlag` contains the option of using an alternate microphone setup. The variable name originates from one of the array types in use called *the football*. Pointer `*f_az_out` handles the vector output.

```
void calcIntensityVectors_tilde_setup(void) {
  calcIntensityVectors_tilde_class = class_new(gensym("calcIntensityVectors~"),
        (t_newmethod)calcIntensityVectors_tilde_new,
        0, sizeof(t_calcIntensityVectors_tilde),
        CLASS_DEFAULT,
        A_GIMME, 0);

  class_addmethod(calcIntensityVectors_tilde_class,
        (t_method)calcIntensityVectors_tilde_dsp, gensym("dsp"), 0);
  CLASS_MAINSIGNALIN(calcIntensityVectors_tilde_class,
t_calcIntensityVectors_tilde, f);
}
```

The above code prepares the object for accepting undefined number of creation arguments using `A_GIMME` definition. In addition, the DSP method is attached to the class. The last row indicates that the inlets accept audio signal vectors instead of atoms as control messages. However, such declaration only provides a single signal inlet. Additional inlets are created in the class constructor with a set of `inlet_new`-commands in the following code segment.

```
void *calcIntensityVectors_tilde_new(t_symbol *s, int argc, t_atom *argv)
{
  t_calcIntensityVectors_tilde *x = (t_calcIntensityVectors_tilde
*)pd_new(calcIntensityVectors_tilde_class);

  x->FootballFlag = 1;
  x->dist = 0.01;
  if(argc>1) {
    x->FootballFlag = atom_getint(argv+1);
    x->dist = atom_getfloat(argv);
  }

  if(argc==1) {
    x->dist = atom_getfloat(argv);
  }

  if(x->FootballFlag > 0) {
     post("Using microphone order for football [+x -y -x +y]");
   }
  else {
    post("Using microphone order for GRAS [+x -x +y -y]");
  }

  post("distance from argument %f", x->dist);

  inlet_new(&x->x_obj, &x->x_obj.ob_pd, &s_signal, &s_signal);
  inlet_new(&x->x_obj, &x->x_obj.ob_pd, &s_signal, &s_signal);
```

```
  inlet_new(&x->x_obj, &x->x_obj.ob_pd, &s_signal, &s_signal);
  inlet_new(&x->x_obj, &x->x_obj.ob_pd, &s_signal, &s_signal);
  inlet_new(&x->x_obj, &x->x_obj.ob_pd, &s_signal, &s_signal);
  inlet_new(&x->x_obj, &x->x_obj.ob_pd, &s_signal, &s_signal);
  inlet_new(&x->x_obj, &x->x_obj.ob_pd, &s_signal, &s_signal);
  inlet_new(&x->x_obj, &x->x_obj.ob_pd, &s_signal, &s_signal);
  inlet_new(&x->x_obj, &x->x_obj.ob_pd, &s_signal, &s_signal);
  inlet_new(&x->x_obj, &x->x_obj.ob_pd, &s_signal, &s_signal);
  inlet_new(&x->x_obj, &x->x_obj.ob_pd, &s_signal, &s_signal);

  inlet_new(&x->x_obj, &x->x_obj.ob_pd, &s_signal, &s_signal); /* decisionIn
vector*/

  outlet_new(&x->x_obj, &s_signal); /* original outlet */
  outlet_new(&x->x_obj, &s_signal);

  x->f_az_out = outlet_new(&x->x_obj, &s_float); /* outlets the float azimuth*/
  x->f_el_out = outlet_new(&x->x_obj, &s_float); /* float elevation */

  return (void *)x;
}
```

In the beginning of the constructor above, the creation arguments are read and the microphone element order as well as the size of the array are defined and printed for the user. On the last lines signal and float outlets are created. Last created signal inlet controls the reliability of the intensity vector estimation. In this example the reliability measure was calculated from the spectral content of the signal. A simple adaptive threshold is used to control the frequency bins that are used in the calculations. Basically, if the analyzed audio signal contains only noise, estimation is not performed.

The actual method for processing the signal vectors is assigned in a `_dsp`-method with pointer arguments of the object dataspace and the signal array. `dsp_add`-method binds the processing method with the signal paths. The number in the second argument denotes the total number of pointers sent to the `calcIntensityVectors_tilde_perform` method. In this example the number of pointers is fairly high due to the requirements of passing the real and imaginary parts of six microphone signal FFTs separately. In addition to the mandatory object dataspace, 13 inlet signal pointers (12 FFT + 1 Decision value), two signal output pointers (estimated azimuth and elevation) and an integer pointer are assigned.

```
void calcIntensityVectors_tilde_dsp(t_calcIntensityVectors_tilde *x, t_signal
**sp)
{
  dsp_add(calcIntensityVectors_tilde_perform, 17, x,
          sp[0]->s_vec, sp[1]->s_vec, sp[2]->s_vec, sp[3]->s_vec, sp[4]->s_vec,
sp[5]->s_vec,
          sp[6]->s_vec, sp[7]->s_vec, sp[8]->s_vec, sp[9]->s_vec, sp[10]->s_vec,
sp[11]->s_vec,
```

```
          sp[12]->s_vec, sp[13]->s_vec, sp[14]->s_vec, sp[0]->s_n);
}
```

The following code fragment shows the DSP wrapper containing the handling of the inlet and outlet signals. Six first inlets receive the real part of FFTs from the microphones. The next six inlets are for the imaginary part.

```
t_int *calcIntensityVectors_tilde_perform(t_int *w)
{
  t_calcIntensityVectors_tilde *x  = (t_calcIntensityVectors_tilde *)(w[1]);
  t_sample  *r1 = (t_sample *)(w[2]);
  t_sample  *i1 = (t_sample *)(w[3]);
  t_sample  *r2 = (t_sample *)(w[4]);
  t_sample  *i2 = (t_sample *)(w[5]);
  t_sample  *r3 = (t_sample *)(w[6]);
  t_sample  *i3 = (t_sample *)(w[7]);
  t_sample  *r4 = (t_sample *)(w[8]);
  t_sample  *i4 = (t_sample *)(w[9]);
  t_sample  *r5 = (t_sample *)(w[10]);
  t_sample  *i5 = (t_sample *)(w[11]);
  t_sample  *r6 = (t_sample *)(w[12]);
  t_sample  *i6 = (t_sample *)(w[13]);

  /* decisionVector input */
  t_sample  *decisionIn = (t_sample *)(w[14]);


  t_sample  *outaz = (t_sample *)(w[15]);
  t_sample  *outel = (t_sample *)(w[16]);
  int        n   = (int)(w[17]);

  double input1[n];
  double xr[n*6];
  double xi[n*6];
  double I[n*3];
  double az[n], el[n];

  double outAZ, outEL;

  double xc[n], yc[n], zc[n];
  double decisionVector[n];

  int i;

  for (i = 0;i < n;i++)
  {
  /* input */
  xr[0*n + i] = (double) r1[i];
  xr[1*n + i] = (double) r2[i];
  xr[2*n + i] = (double) r3[i];
  xr[3*n + i] = (double) r4[i];
  xr[4*n + i] = (double) r5[i];
  xr[5*n + i] = (double) r6[i];
  xi[0*n + i] = (double) i1[i];
  xi[1*n + i] = (double) i2[i];
  xi[2*n + i] = (double) i3[i];
  xi[3*n + i] = (double) i4[i];
```

```
  xi[4*n + i] = (double) i5[i];
  xi[5*n + i] = (double) i6[i];

  decisionVector[i] = (double) decisionIn[i];

  /* output */
  I[0*n + i] = 0;
  I[1*n + i] = 0;
  I[2*n + i] = 0;

  az[i] = 0;
  el[i] = 0;
  }

  calcIntensityVector(x, xr, xi, I, n);

  for(i = 0; i < n/2; i++)
  {
      xc[i] = I[0*n + i];
      yc[i] = I[1*n + i];
      zc[i] = I[2*n + i];
  }

  cart2sph(decisionVector, az, el, xc, yc, zc, n);

  for (i = 0;i < n;i++)
  {
    outaz[i] = (t_sample) az[i];
    outel[i] = (t_sample) el[i];
  }

  outAZ = (double) az[0];
  outEL = (double) el[0];

  outlet_float(x->f_az_out, outAZ*180.0/PI);
  outlet_float(x->f_el_out, outEL*180.0/PI);

  return (w+18);
}
```

The intensity vector estimation is performed in a for-loop iterating through the FFT bins. Real and imaginary parts from the FFTs are assigned and output variables are initialized before passing to the `calcIntensityVector` subroutine. Axes of the Cartesian coordinates are separated from the subroutine output before converting into spherical coordinates in subroutine `cart2sph`.

As described in the tutorial by Zmoelnig [2], the return argument for the method must be the dataspace pointer plus the number of assigned arguments in `dsp_add` method plus one, so in this case `w+(17+1)`.

The principal part of the estimation process is performed in the method `calcIntensityVector` which is listed below.

```
void calcIntensityVector(t_calcIntensityVectors_tilde *x, double *xr, double
```

```
*xi, double *I, int N)
{
    double rho0 = 1.2;
    double dist = x->dist; /* 0.01; */
    int i,j,ii,c,m1,m2;
    double omega;
    double fs = 48000.0; /* 4*48000 */
    double nfft = N;

    /* particle velocity, real, imag */ \eqref{eq:U}
    double uxr[N],uyr[N],uzr[N], uyi[N], uxi[N], uzi[N] ;
    /* pressure average, real, imag */
    double pr[N],pi[N];

    int micorder[6] = {0,1,2,3,4,5};

    if(x->FootballFlag > 0)
    {
       micorder[0] = 0;
       micorder[1] = 2;
       micorder[2] = 1;
       micorder[3] = 3;
       micorder[4] = 4;
       micorder[5] = 5;

    }

    for(i = 0;i < N/2; i++ )
    {
        omega = (2 * PI * fs / nfft) * (i);
        /* multiplying with sqrt(-1) changes the order */
        m1 = micorder[0]; m2 = micorder[1];
        /* X-coordinate particle velocity */
        uxi[i] = -(xr[m1*N+i] - xr[m2*N+i])/(omega*rho0*dist);
        uxr[i] = (xi[m1*N+i] - xi[m2*N+i])/(omega*rho0*dist);
        m1 = micorder[2]; m2 = micorder[3];
        /* Y-coordinate particle velocity */
        uyi[i] = -(xr[m1*N+i] - xr[m2*N+i])/(omega*rho0*dist);
        uyr[i] = (xi[m1*N+i] - xi[m2*N+i])/(omega*rho0*dist);

        m1 = micorder[4]; m2 = micorder[5];
        /* z-coordinate particle velocity */
        uzi[i] = -(xr[m1*N+i] - xr[m2*N+i])/(omega*rho0*dist);
        uzr[i] = (xi[m1*N+i] - xi[m2*N+i])/(omega*rho0*dist);

        /* average pressure */
        pr[i] = (xr[0*N+i] + xr[1*N+i] + xr[2*N+i] + xr[3*N+i] + xr[4*N+i] +
xr[5*N+i])/6;
        pi[i] = (xi[0*N+i] + xi[1*N+i] + xi[2*N+i] + xi[3*N+i] + xi[4*N+i] +
xi[5*N+i])/6;

        c = 0;
        I[c*N + i] = pr[i]*uxr[i] - (-1*pi[i]*uxi[i]);
        c = 1;
        I[c*N + i] = pr[i]*uyr[i] - (-1*pi[i]*uyi[i]);
        c = 2;
        I[c*N + i] = pr[i]*uzr[i] - (-1*pi[i]*uzi[i]);
```

```
    }
}
```

The coordinate transform is performed with the following method. It should be noted that each FFT analysis bin is processed separately. However, here the `decisionVector` variable comes to play, as the angles are calculated only within the globally defined frequency interval (200-4000 Hz) and at those bins where the signal exceeds the defined threshold.

```
void cart2sph(double *decisionVector, double *az, double *el, double *x, double
*y, double *z, int N)
{
    int i;
    double AZr = 0;
    double AZi = 0;
    double AZ;

    double ELr = 0;
    double ELi = 0;
    double EL;

    double sumN = 0;

    int lowBin = (int) floor(Flow/(FS/N));
    int highBin = (int) floor(Fhigh/(FS/N));


    for(i = 1; i < N; i++)
    {
      az[i] = atan2(y[i],x[i]);
      el[i] = acos(z[i] / sqrt(pow(x[i],2)+pow(y[i],2)+pow(z[i],2)));

      if(decisionVector[i] == 1 && i>lowBin && i<highBin) {

        AZr = AZr + cos(az[i]);
        AZi = AZi + sin(az[i]);

        ELr = ELr + cos(el[i]);
        ELi = ELi + sin(el[i]);

        sumN = sumN + 1;
      }
    }
    AZ = atan2(AZi/(sumN/2),AZr/(sumN/2));
    EL = atan2(ELi/(sumN/2),ELr/(sumN/2));
    az[0] = AZ;
    el[0] = EL;
    /* post("Azimuth angle of the spherical mean is %f", AZ*180.0/PI); */
}
```

Complete source code for the 3-D intensity vector estimation is included in the package.

4.3 Histogram

The intensity vector estimation gives the estimates in vector form for each FFT bin. Such format is not very convenient for further use, such as visualization. For this reason, a real-time histogram object is presented. While the example is usable for any histogram purpose, the particular code contains some features for improving the intensity vector analysis. For instance, the calculated bins are limited to usable frequency range.

```
#define FS 48000
#define Flow 100
#define Fhigh 1000

static t_class *hist_tilde_class;

typedef struct _hist_tilde {
  t_object x_obj;
  t_float counter;
  t_float bufferLenIn;
  int noBins;             /* number of histogram bins */
  double minHist;         /* histogram minimum value */
  double maxHist;         /* histogram maximum value */
  t_outlet *f_out;
  t_outlet *bufferLenOut; /* outlet, length for histogram calculation */
  t_float bufferLength;
  int bufNo;
  t_float *histogram;     /* the histogram */
  t_sample f;
} t_hist_tilde;
```

The object dataspace contains the necessary variables for reading and updating the histogram table, and inlet and outlet pointers.

```
void hist_tilde_setup(void) {
  hist_tilde_class = class_new(gensym("hist~"), (t_newmethod) hist_tilde_new,
(t_method) hist_tilde_free,
        sizeof(t_hist_tilde),CLASS_DEFAULT,A_DEFFLOAT,0);

  /* reset histogram with bang */
  class_addbang(hist_tilde_class,resetHistogram);

  class_addmethod(hist_tilde_class, (t_method)printInfo, gensym("info"), 0);

  class_addmethod(hist_tilde_class,
        (t_method)hist_tilde_dsp, gensym("dsp"), 0);
  CLASS_MAINSIGNALIN(hist_tilde_class, t_hist_tilde, f);
}
```

In the method binding, a received `bang` message is defined to reset the whole histogram table with `class_addbang`. An important difference to the previous cases here is the use of destructor method `hist_tilde_free`. It is required to call a memory function for releasing the memory space which is reserved when

creating the object. The memory space is required for storing the histogram table over the life of the object. DSP method is bound similarly as in the previous examples.

```
static void hist_tilde_free(t_hist_tilde *x)
{
  /* free reserved memory when destrying object */
freebytes(x->histogram, (int)(x->bufferLength) * x->noBins*sizeof(t_float));
}
```

Method `freebytes` handles the memory release by pointing to the variable and its size.

```
void *hist_tilde_new(t_floatarg f)
{
  t_hist_tilde *x = (t_hist_tilde *)pd_new(hist_tilde_class);

  t_float *histogram;
  t_float noBins;
  t_float minHist = 0.;
  t_float maxHist = 1.;

  /* optional creation argument defines number of histogram bins */
  noBins = (f)?f:100.;

  x->counter = 0;
  x->bufNo = 0;
  x->minHist = (double) minHist;
  x->maxHist = (double) maxHist;


  x->noBins = (int) noBins;
  x->bufferLength = 20; /* frames */

  post("Histogram bins %d",x->noBins);
  post("Histogram limits %f %f", x->minHist, x->maxHist);

  /* reserve memory for storing the updating histogram */
  x->histogram = (t_float *)getbytes(x->bufferLength * x->noBins * sizeof(t_float));
  histogram = x->histogram;

  resetHistogram(x);

  inlet_new(&x->x_obj, &x->x_obj.ob_pd, &s_signal, &s_signal);
  outlet_new(&x->x_obj, &s_signal);
  x->bufferLenOut = outlet_new(&x->x_obj, &s_float);

  floatinlet_new(&x->x_obj, &x->bufferLenIn);


  return (void *)x;
}
```

The constructor method above received the optional creation argument which defines the number of histogram bins. The number defaults to 100.

Also, the histogram is calculated over the previous 20 frames, defined in `x->bufferLength`. After that, the memory reservation is performed, giving the `x->histogram` variable bins $\times$ frames bytes of own space.

```
static void resetHistogram(t_hist_tilde *x)
{
  /* re-initialize histogram memory space */
int i = x->noBins*(int)x->bufferLength;
t_float *histogram = x->histogram;

while (i--) {
    *histogram++= 0.;
  }

x->counter = 0;
}
```

Initialization of the histogram is accomplished by a simple loop going through the reserved memory space for the table.

```
t_int *hist_tilde_perform(t_int *w)
{
/* inputs and outputs  */
  t_hist_tilde *x   = (t_hist_tilde *)(w[1]);
  t_sample  *in1    = (t_sample *)(w[2]);
  t_sample  *in2    = (t_sample *)(w[3]); /* obsolete (adaptive MFCC) */
  t_sample  *out    = (t_sample *)(w[4]);
  int         n     = (int)(w[5]);

  int i = x->noBins;
  int j;
  int returnBin;

  int bufNo = x->bufNo;

  double temp1[n];
  double outputSum[n];

  /* init */

  t_float histSum = 0.00000;
  t_float histMax = 0.00001;
  t_float tempSum = 0.;

  t_float *histogram = x->histogram;
  t_int returnBins[x->noBins];
  t_int currentBin;

  for(i=0; i<x->noBins; i++) {
 returnBins[i] = 0.;
  }

  int lowBin = (int) floor(Flow/(FS/n));
  int highBin = (int) floor(Fhigh/(FS/n));
```

```
 for(i=0; i<n; i++) {
   if(i>lowBin && i<highBin) {
     /* only count selected interval from intensity vector analysis bins */
     /* then increase the correct bin count by one */
     temp1[i] = (double) in1[i];
     currentBin = calcHistBin(temp1[i], x);
     returnBins[currentBin]++;
     }
 }

 /* replace current buffer */
 for(i=0; i<x->noBins; i++) {
histogram[x->noBins*bufNo + i] = returnBins[i];
 }

 /* calculate sum for each bin over frames*/
 for(i=0; i < x->noBins; i++) {
   tempSum = 0.;
   for(j=0; j < x->bufferLength; j++) {

       tempSum = tempSum + histogram[j*x->noBins + i];

     }

if(tempSum > histMax) histMax = tempSum;
outputSum[i] = tempSum;
histSum = histSum + tempSum;
 }


 /* scale histogram output */
 for(i=0; i < n; i++) {
   if(i<x->noBins) {
     out[i] = (t_sample) (10.0 * outputSum[i] / histSum);
}
else {
       out[i] = (t_sample) (0.0);
}
 }


 /* advance or reset current histogram buffer position */
 if (x->bufNo < x->bufferLength -1) {
   x->bufNo++;
 }
 else {
   x->bufNo = 0;
 }

 /* outlet histogram length */
 outlet_float(x->bufferLenOut, x->bufferLength);

 return (w+6);
}
```

The above code contains the main working method for handling the histogram. It should be noted that extra inlet exists in this slightly reduced example. The

phases after the initialization segment are the following: calculate the correct histogram bins for the individual values in the inlet vector; replace the current position bins in the histogram table; calculate the cumulative sum of the bins in each frame and scale output.

The following method is used to determine the correct histogram bin for a value. The implementation checks whether the value against the bin limits. Alternatively the correct bin could be determined by calculating the division of the value by the histogram bins and scaling the outcome.

```
int calcHistBin(double value, t_hist_tilde *x)
{
  /* check into which bin a value belongs */
double a;
double step;
int i;
int returnBin;
returnBin = 0;
i = 0;
a = x->minHist;
step = (x->maxHist - x->minHist) / x->noBins;

while (a<value && i<x->noBins) {
returnBin = i+1;
a = a + step;
i++;
}
return returnBin-1;
}
```

5 Conclusions

Three examples of PD externals were shown. First of them is calculating Mel Cepstral Coefficients, the second one sound intensity vectors and the third one a histogram.

The visualization of the output of these externals is possible with PD GEM-package.

Comments, corrections, questions, and feedback can be sent to `sakari.tervo@tml.hut.fi` or to `jukka.patynen@tml.hut.fi`, but they are not necessarily answered or taken into account.

Copyleft

References

[1] Juha Merimaa, *Analysis, Synthesis, and Perception of Spatial Sound: Binaural Localization Modeling and Multichannel Loudspeaker Reproduction*. 2006, Helsinki University of Technology.

[2] Johannes M. Zmoelnig, *HOWTO write an External for puredata*. http://pdstatic.iem.at/externals-HOWTO/, 2010.

[3] Wikipedia *Mel-frequency cepstrum*. http://en.wikipedia.org/wiki/Mel-frequency_cepstrum, as of June 2010.

[4] *MinGW tutorial*. http://sites.google.com/site/albertozin2/mingwtutorial, 2010.