# Read AES SOFA files in Pure Data

Author: Edoardo Mortara, Università degli Studi di Milano

*Abstract*—**In this paper I will talk about 'readsofa', a Pure Data External for reading AES SOFA files. In the first part of the paper its use will be described, then I will give a brief description on how to make externals for Pure Data, analysing the basics and some of the main difficulties.**

## I. Introduction

There are currently no Pure Data objects capable of reading AES SOFA file [3]. Pure Data externals that use HRTF usually have an internal predefined HRTF database (like 'earplug$\sim$'). The advantages of using a custom HRTF database are many, and this paper was born precisely to meet these needs.

The goal of my external is to provide a simple interface for reading a given SOFA file in some X-Y-Z coordinates (Cartesian coordinates) to be specified.

## II. SOFA

SOFA is a file format for storing spatially oriented acoustic data like head-related transfer functions (HRTFs) and binaural or directional room impulse responses (BRIRs, DRIRs). SOFA has been standardized by the Audio Engineering Society (AES) as AES69-2015.
A head-related transfer function (HRTF), also sometimes known as the anatomical transfer function (ATF), is a response that characterizes how an ear receives a sound from a point in space. Linear systems analysis defines the transfer function as the complex ratio between the output signal spectrum and the input signal spectrum as a function of frequency.
The transfer function H(f) of any linear time-invariant system at frequency f is:

$$H(f) = Output(f)/Input(f)$$

One method used to obtain the HRTF from a given source location is therefore to measure the head-related impulse response (HRIR), h(t), at the ear drum for the impulse $\Delta(t)$ placed at the source. The HRTF H(f) (Figure 1) is the Fourier transform of the HRIR h(t).

## III. 'readsofa' external

The 'readsofa' external is structured as follows. There are four inlets, the first one accepts the name of the AES SOFA file to be open (as a symbol), the other three accept the X-Y-Z coordinates (Cartesian coordinates) in which to open the HRTF database related to the AES SOFA file (Figure 2 and 3). The object 'readsofa' produces output only when it receives a Bang in the first inlet.

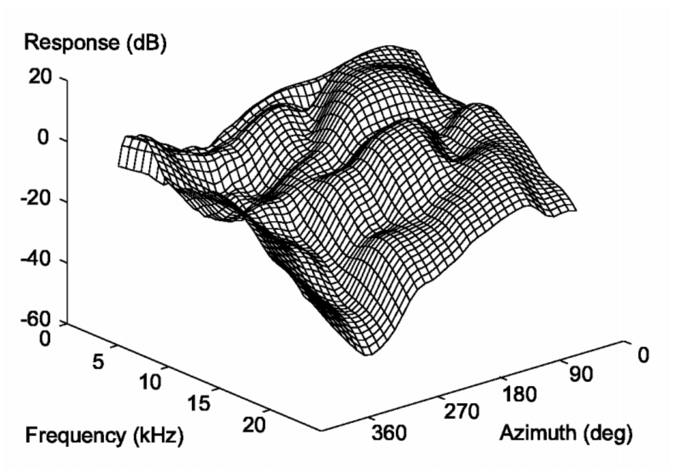

Fig. 1. 'HRTF visualized'

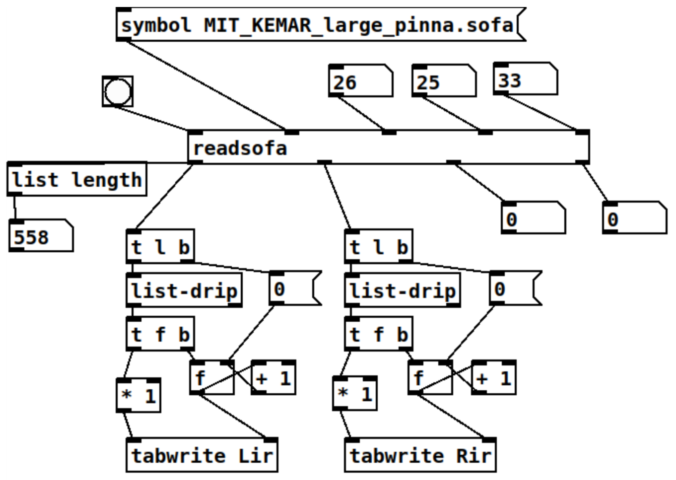

Fig. 2. 'readsofa-help'

The external output the left and right Impulse Response (IR) stored in the required position, and the left and right delay (ITD [8]). The first two output are float lists and with a simple patch [7] they can be wrote in an array (Figure 4).

If the user prefers polar coordinates, they can be translated into X-Y-Z coordinates whit the patch illustrated in Figure 5.
The external I made uses a lightweight AES SOFA API for C language called 'libmysofa' [4]; it allows to read AES SOFA files (AES69-2015 standard) in a simple and fast way.
The following code illustrate how 'readsofa' uses 'libmysofa':

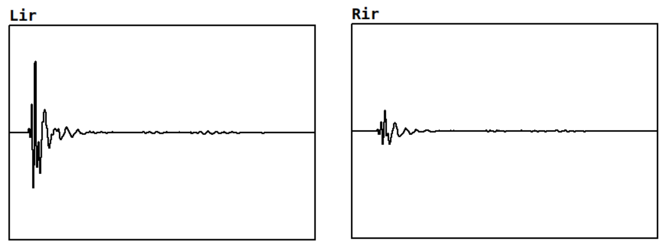

Fig. 3. 'Left and Right IR arrays'

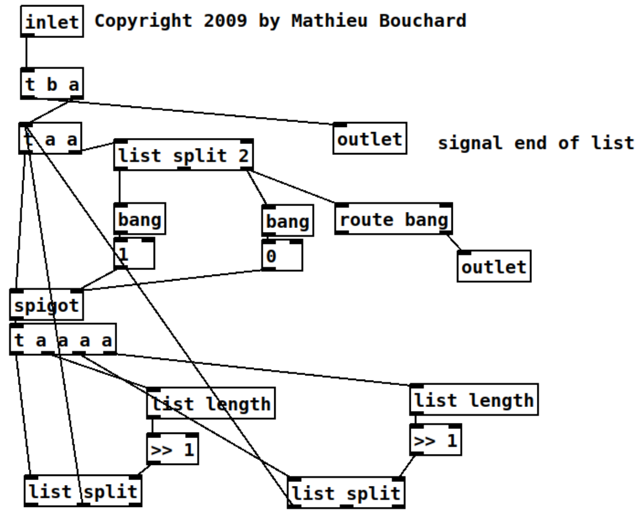

Fig. 4. 'list-drip'

```
void readsofa_bang(t_readsofa *x)
{
    if(x->hrtf == NULL)
        x->hrtf = mysofa_open((x->path)->s_name,
    48000, &x->filter_length, &x->err);
    error((x->path)->s_name);

    float leftIR[x->filter_length]; // [-1. till 1]
    float rightIR[x->filter_length];
    float leftDelay = 0;  // unit is sec.
    float rightDelay = 0; // unit is sec.

    if (x->hrtf != NULL)
    {
        mysofa_getfilter_float(x->hrtf, (float)x->x,
     (float)x->y, (float)x->z, leftIR, rightIR, &
    leftDelay, &rightDelay);
    }

    int l = x->filter_length;
    t_atom argv[l];
    for (size_t i = 0; i < l; i++)
    {
        SETFLOAT(argv + i, leftIR[i]);
    }
    outlet_list(x->l_out, gensym("list"), l, argv);

    for (size_t i = 0; i < l; i++)
    {
        SETFLOAT(argv + i, rightIR[i]);
    }
    outlet_list(x->r_out, gensym("list"), l, argv);
```

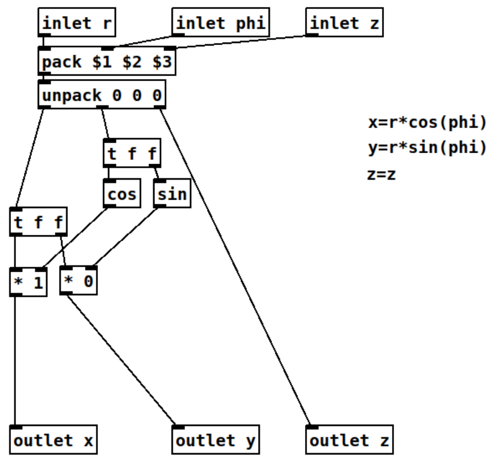

Fig. 5. 'pol2cart'

```
    outlet_float(x->ld_out, leftDelay);
    outlet_float(x->rd_out, rightDelay);
}
```

The function

```
mysofa_getfilter_float(x->hrtf, (float)x->x, (float)
    x->y, (float)x->z, leftIR, rightIR, &leftDelay,
    &rightDelay);
```

allows to read the HRTF file ('$x->hrtf$'),opened with '$mysofa\_open()$', in some X-Y-Z coordinates, save the left and right IR in two float arrays ('leftIR' and 'rightIR') and the right and left delay in two float pointers ('leftDelay' and 'rightDelay').

## IV. Results

This external is a simple interface to SOFA files; this is, the user can read the left and right IR as a float list from an AES SOFA file. This interface and this form of freedom towards HRTF databases was not present among the various Pure Data externals.

The results obtained are good: IR values read are the same as those obtained to other APIs for reading SOFA files [5] [6].
The project can still be improved: using the 'libmysofa' API [4] it is possible to parallelize the reading of the same SOFA file using the function '$mysofa\_open\_cached(\ldots)$'.
On the other hand, I decided to avoid multithreading, because the user can extract in Pure Data all the required IR using Figure 2, and later use them with a normal 'tabread' or 'tabread4~' access.

## V. How to write an external

In this section we will analyse the basis for writing an external for Pure Data in Linux.
The necessary set-up is as follows:
- Pure Data

- C editor (Sublime Text, Atom, VSC...)
- Linux Terminal

Now download *github.com/pure-data/pd-lib-builder* and follow the instructions. This will be the builder for our externals. It's really simple to use; here is a 'MakeFile' example:

```
lib.name = mylib
class.sources = class.c
include /home/user/Pd/pd-lib-builder/Makefile.
    pdlibbuilder
```

In the first line we choose the name of our library, then in the second we select which file .c must be compiled. The third line is the path of the builder we downloaded before.
Now all we have to do is open the terminal in the 'MakeFile' folder and write 'make'. In this project I used other libraries, so we have to specify them using the flag 'ldlibs': " make ldlibs='-lz -lmysofa' ".
Let's focus on the .c files that defines our external. I recommend reading [1] for a full overview.

Here is a simple 'HelloWorld' external example:

```
#include "m_pd.h"
static t_class *helloworld_class;

typedef struct _helloworld {
  t_object  x_obj;
} t_helloworld;

void helloworld_bang(t_helloworld *x)
{
  post("Hello world !!");
}

void *helloworld_new(void)
{
  t_helloworld *x = (t_helloworld *)pd_new(
    helloworld_class);

  return (void *)x;
}

void helloworld_setup(void) {
  helloworld_class = class_new(gensym("helloworld"),
        (t_newmethod)helloworld_new,
        0, sizeof(t_helloworld),
        CLASS_DEFAULT, 0);
  class_addbang(helloworld_class, helloworld_bang);
}
```

First of all, we must include Pure Data interface $m\_pd.h$, because it contains all the basic tools for building an external. Remember that our external is a class, and in C this is implemented using structs and the Pure Data type "t_ class". There are a lot of types in 'm_ pd.h' and I suggest to read '*github.com/pure-data/externals-howto\#pd-types*' for a better understanding.
In this example our class is called "helloworld" and contains only a 't_ object' called 'x_ obj', which is used to store internal object-properties like the graphical presentation of the object or data about inlets and outlets.
The 'helloworld_ setup' function creates the class 'helloworld' with the constructor 'helloworld_ new', then assign to the class the 'bang' function 'helloworld_ bang'. Every time our class instance in Pure Data will receive a bang, it will execute that function.
Every constructor needs to create and return an instance of our class: $t\_helloworld * x = (t\_helloworld*)pd\_new(helloworld\_class);$ . This instance will be used in every function we will associate to our class. From 'x' we can access our class variables and outlets ('$x- > variable\_name$').

Here is a more advanced example:

```
#include "m_pd.h"

static t_class *counter_class;

typedef struct _counter
{
    t_object x_obj;
    t_int i_count;
    t_float step;
    t_int i_down, i_up;
    t_outlet *f_out, *b_out;
} t_counter;

void counter_bang(t_counter *x)
{
    t_float f = x->i_count;
    t_int step = x->step;
    x->i_count += step;

    if (x->i_down - x->i_up)
    {
        if ((step > 0) && (x->i_count > x->i_up))
        {
            x->i_count = x->i_down;
            outlet_bang(x->b_out);
        }
        else if (x->i_count < x->i_down)
        {
            x->i_count = x->i_up;
            outlet_bang(x->b_out);
        }
    }

    outlet_float(x->f_out, f);
}

void counter_reset(t_counter *x)
{
    x->i_count = x->i_down;
}

void counter_set(t_counter *x, t_floatarg f)
{
    x->i_count = f;
}

void counter_bound(t_counter *x, t_floatarg f1,
    t_floatarg f2)
{
    x->i_down = (f1 < f2) ? f1 : f2;
    x->i_up = (f1 > f2) ? f1 : f2;
}

//t_symbol: symbolic name for the class instance
    creation
//argc: number of argument passed
//argv: pointer to the list of arguments
void *counter_new(t_symbol *s, int argc, t_atom *
    argv)
{
    t_counter *x = (t_counter *)pd_new(counter_class
    );
    t_float f1 = 0, f2 = 0;

    x->step = 1;
    switch (argc)
    {
    default:
    case 3:
        x->step = atom_getfloat(argv + 2);
    case 2:
        f2 = atom_getfloat(argv + 1);
    case 1:
        f1 = atom_getfloat(argv);
        break;
    case 0:
        break;
    }
    if (argc < 2)
        f2 = f1;

    x->i_down = (f1 < f2) ? f1 : f2;
    x->i_up = (f1 > f2) ? f1 : f2;

    x->i_count = x->i_down;

    inlet_new(&x->x_obj, &x->x_obj.ob_pd,
              gensym("list"), gensym("bound"));
    floatinlet_new(&x->x_obj, &x->step);

    x->f_out = outlet_new(&x->x_obj, &s_float);
    x->b_out = outlet_new(&x->x_obj, &s_bang);

    return (void *)x;
}

void counter_setup(void)
{
    counter_class = class_new(gensym("counter"),
                   (t_newmethod)counter_new, 0,
    sizeof(t_counter), CLASS_DEFAULT, A_GIMME, 0);

    class_addbang(counter_class, counter_bang);
    class_addmethod(counter_class,
    (t_method)counter_reset, gensym("reset"), 0);
    class_addmethod(counter_class,
     (t_method)counter_set, gensym("set"),
                   A_DEFFLOAT, 0);
    class_addmethod(counter_class,
      (t_method)counter_bound, gensym("bound"),
                  A_DEFFLOAT, A_DEFFLOAT, 0);

    class_sethelpsymbol(counter_class, gensym("help-
    counter"));
}
```

In the class setup we see that the class is created using the parameter ‘A_ GIMME’; this means that the class can accept multiple parameter for the creation (multiple inlet).
With ‘class_ addmethod’ we choose which methods are called if the class instance receives a given message (’reset’, ’set’), and we choose which parameter and type they accept.
In the ’counter_ new’ method we define two inlets: one that accept a ’list’ type, and the other that is bound to the class variable ’step’. This constructor defines also two outlets and their output type.
It’s important to remember that we need to define outlets in the class struct, in order to access them later.
The class internal methods operates on our class variables and in the end output the result using a proper ‘outlet_ ’ method.

Finally, let’s see a signal class:

```
#include "m_pd.h"

static t_class *pan_tilde_class;

typedef struct _pan_tilde
{
    t_object x_obj;
    t_sample f_pan;
    t_sample f;
} t_pan_tilde;

t_int *pan_tilde_perform(t_int *w)
{
    t_pan_tilde *x = (t_pan_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]);
    t_sample f_pan = (x->f_pan < 0) ? 0.0 : (x->
    f_pan > 1) ? 1.0 : x->f_pan;

    while (n--)
        *out++ = (*in1++) * (1 - f_pan) + (*in2++) *
     f_pan;
   //we must return the pointer "w" incremented by
    the number of
     // arguments passed in the "dsp_add" methods,
    plus one.
    return (w + 6);
}

void pan_tilde_dsp(t_pan_tilde *x, t_signal **sp)
{
    //sp0 left signal
    //sp1 right signal
    //sp2 output signal
    //every "sp" point to a vector of signals of
    dimension "s_n"
    dsp_add(pan_tilde_perform, 5, x,sp[0]->s_vec, sp
    [1]->s_vec, sp[2]->s_vec, sp[0]->s_n);
}

void *pan_tilde_new(t_floatarg f)
{
    t_pan_tilde *x = (t_pan_tilde *)pd_new(
    pan_tilde_class);

    x->f_pan = f;

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

    return (void *)x;
}

void pan_tilde_setup(void)
{
    pan_tilde_class = class_new(gensym("pan~"),
                               (t_newmethod)
    pan_tilde_new,
                               0, sizeof(
    t_pan_tilde),
                               CLASS_DEFAULT,
                               A_DEFFLOAT, 0);

    class_addmethod(pan_tilde_class,(t_method)
    pan_tilde_dsp, gensym("dsp"), 0);
    CLASS_MAINSIGNALIN(pan_tilde_class, t_pan_tilde,
     f);
}
```

The first difference between a normal class and a signal class is the presence of a tilde in the name of the class: '$gensym("pan \sim ")$'. A signal class must also define the 'CLASS_ MAINSIGNALIN' method, and have a 't_ sample' object in the struct definition.
When a signal is received the class will execure the 'pan_ tilde_ dsp' method:

```
class_addmethod(pan_tilde_class ,(t_method)
    pan_tilde_dsp , gensym("dsp"), 0);
```

In the 'pan_ tilde_ dsp' method we must define a $dsp$ routine using the function 'dsp_ add', in which we choose which will be our routine function ('pan_ tilde_ perform') and which signal parameters we will use.
The 't_ signal' vector 'sp' contains all the audio inlets (first) and outlets; in this example there are two signal inlet and one signal outlet. Every element of the $sp$ vector contains a signal vector 's_ vec' of size 's_ n'. Remember that all elements have the same size 's_ n'. In the 'dsp_ add' function we must also pass our class istance $x$ and the number of arguments passed after the first (5 in our case).
The function 'pan_ tilde_ perform' will use the five arguments passed and must return the arguments pointer incremented by '1 + arguments passed'.

## VI. Tips and Advice

During the creation of an external there is a high chance to get stuck, or to have doubts. Here are some useful links:
*-forum.pdpatchrepo.info*
*-puredata.info/docs*
*-write.flossmanuals.net/pure-data/arrays-graphs-tables*
For other insights watch this Youtube series '*youtu.be/041L3Q5S9qI*', and look at more examples [2].
The best way to learn is always practice!

## References

[1] HOWTO write an External for Pure Data; github.com/pure-data/externals-howto.
[2] Creating Pure Data Externals in C (2016); github.com/cheetomoskeeto/pd-externals-c.
[3] SOFA (Spatially Oriented Format for Acoustics); www.sofaconventions.org/mediawiki/index.php/SOFA_(Spatially_Oriented_Format_for_Acoustics)
[4] libmysofa: Lightweight SOFA API in C (reading); github.com/hoene/libmysofa
[5] HDF5View: Generic SOFA file viewer; support.hdfgroup.org/products/java/hdfview
[6] SOFA API for Matlab/Octave; github.com/sofacoustics/sofa
[7] list-drip.pd patch; github.com/duvtedudug/Pure-Data/blob/master/extra/list-abs/list-drip.pd
[8] Interaural time difference; en.wikipedia.org/wiki/Interaural_time_difference