```c
/*
Program     : Room Impulse Response Generator

Description : Computes the response of an acoustic source to one or more
              microphones in a reverberant room using the image method [1,2].

              [1] J.B. Allen and D.A. Berkley,
              Image method for efficiently simulating small-room acoustics,
              Journal Acoustic Society of America, 65(4), April 1979, p 943.

              [2] P.M. Peterson,
              Simulating the response of multiple microphones to a single
              acoustic source in a reverberant room, Journal Acoustic
              Society of America, 80(5), November 1986.

Author      : dr.ir. E.A.P. Habets (ehabets@dereverberation.org)

Version     : 2.1.20141124

History     : 1.0.20030606 Initial version
              1.1.20040803 + Microphone directivity
                           + Improved phase accuracy [2]
              1.2.20040312 + Reflection order
              1.3.20050930 + Reverberation Time
              1.4.20051114 + Supports multi-channels
              1.5.20051116 + High-pass filter [1]
                           + Microphone directivity control
              1.6.20060327 + Minor improvements
              1.7.20060531 + Minor improvements
              1.8.20080713 + Minor improvements
              1.9.20090822 + 3D microphone directivity control
              2.0.20100920 + Calculation of the source-image position
                             changed in the code and tutorial.
                             This ensures a proper response to reflections
                             in case a directional microphone is used.
              2.1.20120318 + Avoid the use of unallocated memory
              2.1.20140721 + Fixed computation of alpha
              2.1.20141124 + The window and sinc are now both centered
                             around t=0

Copyright (C) 2003-2014 E.A.P. Habets, The Netherlands.

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA  02110-1301  USA
*/

#define _USE_MATH_DEFINES

#include "matrix.h"
#include "mex.h"
#include "math.h"

#define ROUND(x) ((x)>=0?(long)((x)+0.5):(long)((x)-0.5))

#ifndef M_PI
    #define M_PI 3.14159265358979323846
```

```
#endif

double sinc(double x)
{
   if (x == 0)
      return(1.);
   else
      return(sin(x)/x);
}

double sim_microphone(double x, double y, double z, double* angle, char mtype)
{
   if (mtype=='b' || mtype=='c' || mtype=='s' || mtype=='h')
   {
      double gain, vartheta, varphi, rho;
```

Here x, y, z are relative vector from source to receiver.

angle = angle of microphone

```
      // Polar Pattern         rho
      // ---------------------------
      // Bidirectional         0
      // Hypercardioid         0.25
      // Cardioid              0.5
      // Subcardioid           0.75
      // Omnidirectional       1

      switch(mtype)
      {
      case 'b':
         rho = 0;
         break;
      case 'h':
         rho = 0.25;
         break;
      case 'c':
         rho = 0.5;
         break;
      case 's':
         rho = 0.75;
         break;
      };
```

$\theta \rightarrow$ elevation angle

$\phi \rightarrow$ azimuthal angle

```
      vartheta = acos(z/sqrt(pow(x,2)+pow(y,2)+pow(z,2)));
      varphi = atan2(y,x);

      gain = sin(M_PI/2-angle[1]) * sin(vartheta) * cos(angle[0]-varphi) + cos(M_PI/2-angle[1]) * cos(vartheta);
      gain = rho + (1-rho) * gain;

      return gain;
   }
   else
   {
      return 1;
```

$\longrightarrow$ for omnidirectional mic (usual case)

```
   }
}

void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])
{
   if (nrhs == 0)
   {
      mexPrintf("--------------------------------------------------------------------\n"
         "| Room Impulse Response Generator                                  |\n"
         "|                                                                  |\n"
         "| Computes the response of an acoustic source to one or more       |\n"
         "| microphones in a reverberant room using the image method [1,2].  |\n"
         "|                                                                  |\n"
         "| Author    : dr.ir. Emanuel Habets (ehabets@dereverberation.org)  |\n"
```

```
        "|                                                    |\n"
        "| Version   : 2.1.20141124                                       |\n"
        "|                                                    |\n"
        "| Copyright (C) 2003-2014 E.A.P. Habets, The Netherlands.          |\n"
        "|                                                    |\n"
        "| [1] J.B. Allen and D.A. Berkley,                                |\n"
        "|     Image method for efficiently simulating small-room acoustics,|\n"
        "|     Journal Acoustic Society of America,                        |\n"
        "|     65(4), April 1979, p 943.                                  |\n"
        "|                                                    |\n"
        "| [2] P.M. Peterson,                                             |\n"
        "|     Simulating the response of multiple microphones to a single  |\n"
        "|     acoustic source in a reverberant room, Journal Acoustic      |\n"
        "|     Society of America, 80(5), November 1986.                    |\n"
        "----------------------------------------------------------------\n\n"
        "function [h, beta_hat] = rir_generator(c, fs, r, s, L, beta, nsample,\n"
        " mtype, order, dim, orientation, hp_filter);\n\n"
        "Input parameters:\n"
        " c           : sound velocity in m/s.\n"
        " fs          : sampling frequency in Hz.\n"
        " r           : M x 3 array specifying the (x,y,z) coordinates of the\n"
        "               receiver(s) in m.\n"
        " s           : 1 x 3 vector specifying the (x,y,z) coordinates of the\n"
        "               source in m.\n"
        " L           : 1 x 3 vector specifying the room dimensions (x,y,z) in m.\n"
        " beta        : 1 x 6 vector specifying the reflection coefficients\n"
        "               [beta_x1 beta_x2 beta_y1 beta_y2 beta_z1 beta_z2] or\n"
        "               beta = reverberation time (T_60) in seconds.\n"
        " nsample     : number of samples to calculate, default is T_60*fs.\n"
        " mtype       : [omnidirectional, subcardioid, cardioid, hypercardioid,\n"
        "               bidirectional], default is omnidirectional.\n"
        " order       : reflection order, default is -1, i.e. maximum order.\n"
        " dim         : room dimension (2 or 3), default is 3.\n"
        " orientation : direction in which the microphones are pointed, specified using\n"
        "               azimuth and elevation angles (in radians), default is [0 0].\n"
        " hp_filter   : use 'false' to disable high-pass filter, the high-pass filter\n"
        "               is enabled by default.\n\n"
        "Output parameters:\n"
        " h           : M x nsample matrix containing the calculated room impulse\n"
        "               response(s).\n"
        " beta_hat    : In case a reverberation time is specified as an input parameter\n"
        "               the corresponding reflection coefficient is returned.\n\n");
    return;
}
else
{
    mexPrintf("Room Impulse Response Generator (Version 2.1.20141124) by Emanuel Habets\n"
        "Copyright (C) 2003-2014 E.A.P. Habets, The Netherlands.\n");
}

// Check for proper number of arguments
if (nrhs < 6)
    mexErrMsgTxt("Error: There are at least six input parameters required.");
if (nrhs > 12)
    mexErrMsgTxt("Error: Too many input arguments.");
if (nlhs > 2)
    mexErrMsgTxt("Error: Too many output arguments.");

// Check for proper arguments
if (!(mxGetN(prhs[0])==1) || !mxIsDouble(prhs[0]) || mxIsComplex(prhs[0]))
    mexErrMsgTxt("Invalid input arguments!");
if (!(mxGetN(prhs[1])==1) || !mxIsDouble(prhs[1]) || mxIsComplex(prhs[1]))
    mexErrMsgTxt("Invalid input arguments!");
if (!(mxGetN(prhs[2])==3) || !mxIsDouble(prhs[2]) || mxIsComplex(prhs[2]))
```

$\beta \rightarrow$ reflection coefficient or reverb time

```
    mexErrMsgTxt("Invalid input arguments!");
if (!(mxGetN(prhs[3])==3) || !mxIsDouble(prhs[3]) || mxIsComplex(prhs[3]))
    mexErrMsgTxt("Invalid input arguments!");
if (!(mxGetN(prhs[4])==3) || !mxIsDouble(prhs[4]) || mxIsComplex(prhs[4]))
    mexErrMsgTxt("Invalid input arguments!");
if (!(mxGetN(prhs[5])==6 || mxGetN(prhs[5])==1) || !mxIsDouble(prhs[5]) || mxIsComplex(prhs[5]))
    mexErrMsgTxt("Invalid input arguments!");

// Load parameters
double          c = mxGetScalar(prhs[0]);
double          fs = mxGetScalar(prhs[1]);
const double*   rr = mxGetPr(prhs[2]);
int             nMicrophones = (int) mxGetM(prhs[2]);
const double*   ss = mxGetPr(prhs[3]);
const double*   LL = mxGetPr(prhs[4]);
const double*   beta_input = mxGetPr(prhs[5]);
double*         beta = new double[6];
int             nSamples;
char*           microphone_type;
int             nOrder;
int             nDimension;
double          angle[2];
int             isHighPassFilter;
double          reverberation_time = 0;

// Reflection coefficients or reverberation time?
if (mxGetN(prhs[5])==1)
```

reverberation time given

↳ compute reflection coeff. (see eq^n [9])

$$\alpha = 1 - \beta^2 \Rightarrow \beta = \sqrt{1-\alpha}$$

α → absorption coeff.

```
{
    double V = LL[0]*LL[1]*LL[2];
    double S = 2*(LL[0]*LL[2]+LL[1]*LL[2]+LL[0]*LL[1]);
    reverberation_time = beta_input[0];
    if (reverberation_time != 0) {
        double alfa = 24*V*log(10.0)/(c*S*reverberation_time);
        if (alfa > 1)
            mexErrMsgTxt("Error: The reflection coefficients cannot be calculated using the current "
                         "room parameters, i.e. room size and reverberation time.\n           Please "
                         "specify the reflection coefficients or change the room parameters.");
        for (int i=0;i<6;i++)
            beta[i] = sqrt(1-alfa);
    }
    else
    {
        for (int i=0;i<6;i++)
            beta[i] = 0;
    }
}
```

$\boxed{\text{Sabine equation}}$:

$$T_{60} = \frac{24 \ln 10}{c_{20}} \frac{V}{S\alpha}$$

V → volume

α → absorption coeff.

S → surface area

T_{60} → time taken for signal to reduce by 60 dB

c_{20} → Speed of sound at 20°C.

reflection coefficient

```
else
{
    for (int i=0;i<6;i++)
        beta[i] = beta_input[i];
}

// High-pass filter (optional)
if (nrhs > 11 &&  mxIsEmpty(prhs[11]) == false)
{
    isHighPassFilter = (int) mxGetScalar(prhs[11]);
}
else
{
    isHighPassFilter = 1;
}

// 3D Microphone orientation (optional)
if (nrhs > 10 &&  mxIsEmpty(prhs[10]) == false)
{
    const double* orientation = mxGetPr(prhs[10]);
```

mxGetPr → use mxGetDoubles instead !

```
        if (mxGetN(prhs[10]) == 1)
```

returns no. of columns

```
        {
            angle[0] = orientation[0];
            angle[1] = 0;
        }
        else
        {
            angle[0] = orientation[0];
            angle[1] = orientation[1];
        }
    }
    else
    {
        angle[0] = 0;
        angle[1] = 0;
    }

    // Room Dimension (optional)
    if (nrhs > 9 &&  mxIsEmpty(prhs[9]) == false)
    {
        nDimension = (int) mxGetScalar(prhs[9]);
        if (nDimension != 2 && nDimension != 3)
            mexErrMsgTxt("Invalid input arguments!");

        if (nDimension == 2)
        {
            beta[4] = 0;
            beta[5] = 0;
        }
```

for 2-D room, just set reflection coeff. of remaining 2 to be 0.

```
    }
    else
    {
        nDimension = 3;
    }

    // Reflection order (optional)
    if (nrhs > 8 &&  mxIsEmpty(prhs[8]) == false)
    {
        nOrder = (int) mxGetScalar(prhs[8]);
        if (nOrder < -1)
            mexErrMsgTxt("Invalid input arguments!");
    }
    else
    {
        nOrder = -1;
    }

    // Type of microphone (optional)
    if (nrhs > 7 &&  mxIsEmpty(prhs[7]) == false)
    {
```

for null character (like C)

```
        microphone_type = new char[mxGetN(prhs[7])+1];
        mxGetString(prhs[7], microphone_type, mxGetN(prhs[7])+1);
    }
    else
    {
        microphone_type = new char[1];
        microphone_type[0] = 'o';
    }

    // Number of samples (optional)
    if (nrhs > 6 &&  mxIsEmpty(prhs[6]) == false)
    {
        nSamples = (int) mxGetScalar(prhs[6]);
    }
    else
    {
```

```
    if (mxGetN(prhs[5])>1)
    {
       double V = LL[0]*LL[1]*LL[2];
       double alpha = ((1-pow(beta[0],2))+(1-pow(beta[1],2)))*LL[1]*LL[2] +
          ((1-pow(beta[2],2))+(1-pow(beta[3],2)))*LL[0]*LL[2] +
          ((1-pow(beta[4],2))+(1-pow(beta[5],2)))*LL[0]*LL[1];
```

average α

```
       reverberation_time = 24*log(10.0)*V/(c*alpha);
```

→ Sabine equation

```
       if (reverberation_time < 0.128)
          reverberation_time = 0.128;
```

→ why? so that nSamples ≥ 1

```
    }
    nSamples = (int) (reverberation_time * fs);
}

// Create output vector
plhs[0] = mxCreateDoubleMatrix(nMicrophones, nSamples, mxREAL);
```

→ no imaginary elements in the matrix

```
double* imp = mxGetPr(plhs[0]);
```

→ change to mxGetDoubles

```
// Temporary variables and constants (high-pass filter)
```

→ to remove non-physical behaviour at very low frequency. See Peterson et al.

```
const double W = 2*M_PI*100/fs; // The cut-off frequency equals 100 Hz
const double R1 = exp(-W);
const double B1 = 2*R1*cos(W);
const double B2 = -R1 * R1;
const double A1 = -(1+R1);
double      X0;
double*     Y = new double[3];

// Temporary variables and constants (image-method)
const double Fc = 1; // The cut-off frequency equals fs/2 - Fc is the normalized cut-off frequency.
const int   Tw = 2 * ROUND(0.004*fs); // The width of the low-pass FIR equals 8 ms
const double cTs = c/fs;
double*     LPI = new double[Tw];
double*     r = new double[3];
double*     s = new double[3];
double*     L = new double[3];
double      Rm[3];
double      Rp_plus_Rm[3];
double      refl[3];
double      fdist,dist;
double      gain;
int         startPosition;
int         n1, n2, n3;
int         q, j, k;
int         mx, my, mz;
int         n;
```

$$h(t) = \frac{1}{2}\left[1 + \cos(2\pi t/T_w)\right] \mathrm{sinc}(2\pi f_c t), \quad -T_w/2 < t < T_w/2$$

$$= 0, \quad \text{otherwise}$$

ss → source position

cTs → distance covered by sound between 2 samples (since fs is sampling frequency)

LL → room length, width, height

rr → receiver positions

r[0,1,2] → one particular microphone

```
s[0] = ss[0]/cTs; s[1] = ss[1]/cTs; s[2] = ss[2]/cTs;
L[0] = LL[0]/cTs; L[1] = LL[1]/cTs; L[2] = LL[2]/cTs;

for (int idxMicrophone = 0; idxMicrophone < nMicrophones ; idxMicrophone++)
{
    // [x_1 x_2 ... x_N y_1 y_2 ... y_N z_1 z_2 ... z_N]
    r[0] = rr[idxMicrophone + 0*nMicrophones] / cTs;
    r[1] = rr[idxMicrophone + 1*nMicrophones] / cTs;
    r[2] = rr[idxMicrophone + 2*nMicrophones] / cTs;

    n1 = (int) ceil(nSamples/(2*L[0]));
    n2 = (int) ceil(nSamples/(2*L[1]));
    n3 = (int) ceil(nSamples/(2*L[2]));
```

at most these many reflections will be included to get nSamples.

```
    // Generate room impulse response
    for (mx = -n1 ; mx <= n1 ; mx++)
    {
       Rm[0] = 2*mx*L[0];

       for (my = -n2 ; my <= n2 ; my++)
       {
```

n1 * n2 * n3 → no. of image rooms / sources

Rm gives one particular image source

```
        Rm[1] = 2*my*L[1];

        for (mz = -n3 ; mz <= n3 ; mz++)
        {
          Rm[2] = 2*mz*L[2];

          for (q = 0 ; q <= 1 ; q++)
          {
            Rp_plus_Rm[0] = (1-2*q)*s[0] - r[0] + Rm[0];
            refl[0] = pow(beta[0], abs(mx-q)) * pow(beta[1], abs(mx));
```

for ±

now we loop over all sign permutation

$R \pm R_0 \rightarrow$ to source

↳ to receiver

$\pm S_x - r_x + Rm_x$

```
            for (j = 0 ; j <= 1 ; j++)
            {
              Rp_plus_Rm[1] = (1-2*j)*s[1] - r[1] + Rm[1];
              refl[1] = pow(beta[2], abs(my-j)) * pow(beta[3], abs(my));

              for (k = 0 ; k <= 1 ; k++)
              {
                Rp_plus_Rm[2] = (1-2*k)*s[2] - r[2] + Rm[2];
                refl[2] = pow(beta[4],abs(mz-k)) * pow(beta[5], abs(mz));

                dist = sqrt(pow(Rp_plus_Rm[0], 2) + pow(Rp_plus_Rm[1], 2) + pow(Rp_plus_Rm[2], 2));

                if (abs(2*mx-q)+abs(2*my-j)+abs(2*mz-k) <= nOrder || nOrder == -1)
```

↳ current reflection order

```
                {
                  fdist = floor(dist);
                  if (fdist < nSamples)
                  {
                    gain = sim_microphone(Rp_plus_Rm[0], Rp_plus_Rm[1], Rp_plus_Rm[2], angle,
microphone_type[0])
                        * refl[0]*refl[1]*refl[2]/(4*M_PI*dist*cTs);

                    for (n = 0 ; n < Tw ; n++)
```

sum over all in Hanning window

```
                        LPI[n] =  0.5 * (1 - cos(2*M_PI*((n+1-(dist-fdist))/Tw))) * Fc *
sinc(M_PI*Fc*(n+1-(dist-fdist)-(Tw/2)));

                    startPosition = (int) fdist-(Tw/2)+1;
                    for (n = 0 ; n < Tw; n++)
                        if (startPosition+n >= 0 && startPosition+n < nSamples)
                            imp[idxMicrophone + nMicrophones*(startPosition+n)] += gain * LPI[n];
                  }
                }
              }
            }
          }
        }
      }
```

now we have 1 image source and 1 reciver at relative distance Rp_plus_Rm

$$\delta_{LPF}(n-\varepsilon) = \begin{cases} \frac{1}{2}\left(1+\cos\frac{2\pi(n-\varepsilon)}{N_w}\right)\text{sinc}(n-\varepsilon) & -\frac{N_w}{2} < n < \frac{N_w}{2} \\ 0, & \text{otherwise} \end{cases}$$

cut-off = 1

$$\frac{1}{2}\left(1-\cos\left(2\pi\frac{n+1-\varepsilon}{T_w}\right)\right)\cancel{F_c}$$

$$\text{sinc}\left(\pi\cdot\cancel{F_c}\left(n+1-\varepsilon-\frac{T_w}{2}\right)\right)$$

```
    // 'Original' high-pass filter as proposed by Allen and Berkley.
    if (isHighPassFilter == 1)
    {
      for (int idx = 0 ; idx < 3 ; idx++) {Y[idx] = 0;}
      for (int idx = 0 ; idx < nSamples ; idx++)
      {
        X0 = imp[idxMicrophone+nMicrophones*idx];
        Y[2] = Y[1];
        Y[1] = Y[0];
        Y[0] = B1*Y[1] + B2*Y[2] + X0;
        imp[idxMicrophone+nMicrophones*idx] = Y[0] + A1*Y[1] + R1*Y[2];
      }
    }
  }

  if (nlhs > 1) {
    plhs[1] = mxCreateDoubleMatrix(1, 1, mxREAL);
```

```
        double* beta_hat = mxGetPr(plhs[1]);
        if (reverberation_time != 0) {
            beta_hat[0] = beta[0];
        }
        else {
            beta_hat[0] = 0;
        }
    }

    delete[] beta;
    delete[] microphone_type;
    delete[] Y;
    delete[] LPI;
    delete[] r;
    delete[] s;
    delete[] L;
}
```