FDS PLATE

Physical Models for Sound Synthesis Miniproject [*]

Rasmus Kjærbo *Sound and Music Computing*
Aalborg University
Copenhagen, Denmark
rkjarb19@student.aau.dk

***Abstract*—In this miniproject a brief overview of Finite Difference Schemes leads to the implementation of a 2-dimensional plate in the programming language $MATLAB$. Variable pluck size and position along with variable pick-up position is implemented. Frequency (in)dependent damping is implemented. Future work includes complete implementation of a stability check and the conversion of $MATLAB$ code to C++ for a real-time implementation.**

***Index Terms*—Physical Models for Sound Synthesis, Finite Difference Schemes, FDS**

I. Introduction

In exploration of sound or for the sake of experimentation or nostalgia, physical models of acoustic phenomena and musical instruments can be estimated - a field in digital signal processing (DSP) which has gained quite the interested of late since computational power has made real-time models of complex systems (more) approachable. Alternatives to physical models are sample-based synthesis utilizing digital recordings or sinusoidal-based forms like additive synthesis or filter-based forms like subtractive synthesis. Wavetable synthesis, Frequency Modulation (FM), and Amplitude Modulation (AM) synthesis are versatile tools not very costly in DSP power, but they all lack the finesse of higher order models. See figure 1 for an overview and timeline of the various methods. [1] [2] [3]. To approximate the *natural* behaviour of a physical acoustic or musical system, e.g. a set of sympathetic vibrating metal strings fastened on a wooden resonating body with a bridge, nut, neck etc. (i.e. a guitar), different models of synthesis are needed; e.g. digital waveguides, modal synthesis, or finite difference time-domain (FDTD) methods. To model the proposed guitar, one has to discretise each element of the system and derive the corresponding equations which describe how the system as a whole reacts to excitation and dampening. A costly effort resulting in high fidelity output and plenty possibilities of expressivity. As Julius O. Smith states it:

> *Music synthesis based on a physical model promises the highest quality when it comes to imitating natural instruments. Because the artificial instrument can have the same control parameters as the real instrument, expressivity of control is unbounded. [1]*

The costly part of the equations result from discretising the physical objects to a finite set of interconnected points, i.e. stretching the material out on a grid with a finite mask size. Which is exactly the essence of Finite Difference Time-Domain.

[*]This project is an independent work of the author and not supported in any commercial means.

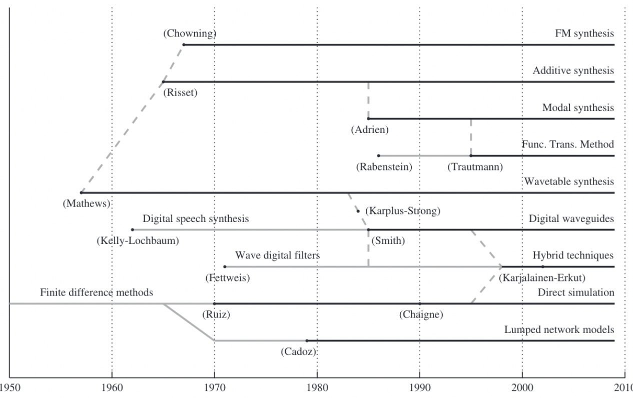


Fig. 1: Historical timeline for digital sound synthesis methods copied from [2] page. 2. "Names of authors/inventors appear in parentheses; dates are approximate, and in some cases have been fixed here by anecdotal information rather than publication dates".

Mass-spring, exciter-resonator, partial differential equations.

II. Finite Difference Time-Domain Methods

Continuous time as difference equations : discretised as a finite difference scheme.

An advantage of a physical model is to toy with the laws of physics, setting certain parameters to extremes for sound design purposes. This could even be done dynamically; imagine an infamous Wendy Carlos plate in any size or shape changing density over time while you hammer away on it. The draw back is the minute attention required to maintain stability, i.e. if your scheme and the physical properties become unstable due to bad energy analysis, the system as a whole might explode (and your program will crash and your ears get tired) [3].

To describe the physical behaviour of a system (i.e. an instrument or an object), differential equations can be used. Differential equations are great tools for describing dynamic motion in both space and time. The brilliant thing about differential equations is that they describe not only the absolute state of a system, but rather the movement of the system, e.g. velocity for the first order time derivative or acceleration for the second order time derivative. The absolute state of the system is the goal, computed by the partial differential

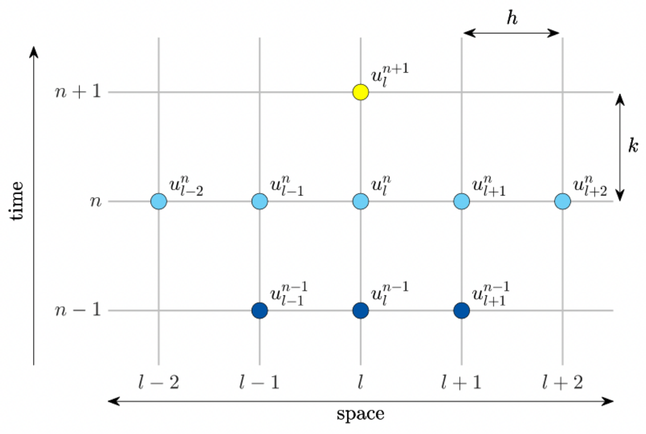
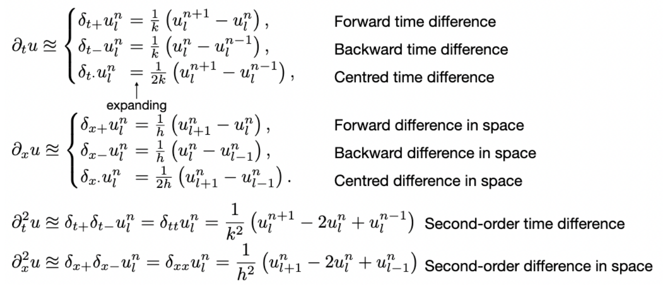


Fig. 2: Stencil of a damped stiff sting copied from [3] page. 80 with time step k and spatial step h



$$\partial_t u \approxeq \begin{cases} \delta_{t+}u_l^n = \frac{1}{k}\left(u_l^{n+1} - u_l^n\right), & \text{Forward time difference} \\ \delta_{t-}u_l^n = \frac{1}{k}\left(u_l^n - u_l^{n-1}\right), & \text{Backward time difference} \\ \delta_{t\cdot}u_l^n \underset{\overset{\uparrow}{\text{expanding}}}{=} \frac{1}{2k}\left(u_l^{n+1} - u_l^{n-1}\right), & \text{Centred time difference} \end{cases}$$

$$\partial_x u \approxeq \begin{cases} \delta_{x+}u_l^n = \frac{1}{h}\left(u_{l+1}^n - u_l^n\right), & \text{Forward difference in space} \\ \delta_{x-}u_l^n = \frac{1}{h}\left(u_l^n - u_{l-1}^n\right), & \text{Backward difference in space} \\ \delta_{x\cdot}u_l^n = \frac{1}{2h}\left(u_{l+1}^n - u_{l-1}^n\right). & \text{Centred difference in space} \end{cases}$$

$$\partial_t^2 u \approxeq \delta_{t+}\delta_{t-}u_l^n = \delta_{tt}u_l^n = \frac{1}{k^2}\left(u_l^{n+1} - 2u_l^n + u_l^{n-1}\right) \quad \text{Second-order time difference}$$

$$\partial_x^2 u \approxeq \delta_{x+}\delta_{x-}u_l^n = \delta_{xx}u_l^n = \frac{1}{h^2}\left(u_{l+1}^n - 2u_l^n + u_{l-1}^n\right) \quad \text{Second-order difference in space}$$

Fig. 3: Stencil of a damped stiff sting copied from lecture slides and [3]

equations (PDEs) which make up the system. Usually, the absolute state is described with the variable u, which can be a function of one-dimensional (1D) space and time $u = u(x,t)$ like a string, or u can be a function of higher order systems like two-dimensional (2D) space and time, $u = u(x,y,t)$ [3]. Describing the dynamic movement of a guitar string with only one spatial dimension is of course a simplified version of the actual physical phenomenon. This reduction of space - or degrees of freedom - is in fact to simplify the computations related to the system. We can reduce the dimensions which is of lesser importance by looking at the difference of the physical dimensions, e.g. the length of a string is by far greater than the circumference. The system state-vector u for a simplified 2D system, like a plate, is a vector $u(x,y,t)$ with coordinates corresponding to the x and y-plane at a given time t Once the physical properties of the differential equations is made, the next step is to discretise the system (i.e. sampling audio) so we can make computations and manipulations of the system. To derive the difference equation for numerical analysis, we make use of stencils (cf. fig. 2) denoting the differential behaviour in space-time of a given system.

Taking fig. 2 as a reference, for a 2D system the space grid is shown as $\Delta h = \frac{x}{l}$ and the time grid as $\Delta k = \frac{t}{n}$ where $(x,y) \in \mathcal{D}$ and $\mathcal{D} = [0, L_x] \times [0, L_y]$. L_x and L_y denotes the length of the given x and y dimensions, and therefore the dimensions of the 2D object. Discretising a 2D object, like a plate, can therefore be written as follows:

$$\begin{aligned} \partial_t^2 u &= c^2\left(\partial_x^2 + \partial_y^2\right)u \\ \partial_t^2 u &= c^2 \Delta u \end{aligned} \tag{1}$$

where Δ = the laplacian operator, describing a second-order spatial derivative in 2D. Expansion of a PDE can be done with a set of finite difference (FD) operators approximating forwards, backwards, and centered differences for space and time, cf. fig. 3.

Using the FD operators from fig. 3, the finite difference scheme eq. 1 with added damping $2\sigma_1\delta_t\Delta u$ is discretised to the following FDTD approximation:

$$\begin{aligned} \partial_t^2 u &= c^2 \Delta u + 2\sigma_1 \delta_t \Delta u \\ \delta_{tt}^2 u_{l,m}^n &= c^2\left(\delta_{xx} + \delta_{yy}\right) + 2\sigma_1\delta_t \\ &\quad - \left(\delta_{xx} + \delta_{yy}\right)u_{l,m}^n \\ \delta_{tt}^2 u_{l,m}^n &= c^2\left(\delta_{xx} + \delta_{yy}\right) + 2\sigma_1\delta_t \\ &\quad - \left(\frac{1}{h^2}\left(u_{l+1,m}^2 + u_{l-1,m}^2 + u_{l,m+1}^2 + u_{l,m-1}^2 - 4u_{l,m}^n\right)\right) \end{aligned} \tag{2}$$

The full expansion of eq. 2 with the finite difference operators of is show below:

$$\begin{aligned} u_{l,m}^{n+1} = \frac{2\sigma_1}{h^2 k}\Big(& u_{l+1,m}^n + u_{l-1,m}^n + u_{l,m+1}^n + u_{l,m-1}^n - 4u_{l,m}^n - \\ & u_{l+1,m}^{n-1} + u_{l-1,m}^{n-1} - u_{l,m+1}^{n-1} + u_{l,m-1}^{n-1} + 4u_{l,m}^{n-1}\Big) \end{aligned} \tag{3}$$

Notice that the above full expansion in eq. 3 is a discrete time *approximation* of the given continuous system, and therefore note 100% accurate.

A. *Boundary Conditions*

Discretising dynamic systems of a given size and shape naturally has boundaries. These boundaries can have different conditions, yielding various physical and sonic / acoustic results. According to the literature [1] [2] [3] the numerical boundary conditions of physical systems must be set. Usually, this can be to either fixed boundaries (Dirichlet) or free (Neumann) and is directly related to what system you are modelling. A 1D string is fixed at both ends, i.e. the nut and the bridge. A clarinet on the other hand (a tube with travelling waves of air) have one fixed end (mouthpiece / barrel) and one free end (bell). The fixed boundary condition, Dirichlet, states that the end points of the system are *fixed* at 0 and remains unchanged, inverting travelling waves reaching the boundary. Free boundary conditions (Neumann) fixes the slope (not the value), so that incoming travelling waves *bounce back* or *mirror* itself orthogonal to the direction of the medium (string). The wave remains un-inverted [2] [3].

$$u(0,t) = u(L,t) = 0 \quad \text{(Fixed, Dirichlet)} \tag{4}$$

$$\partial_x u(0,t) = \partial_x u(L,t) = 0 \quad \text{(Free, Neumann)} \tag{5}$$

Recall the range of your medium, $l \in \{0,\ldots,N\}$. Implementing the fixed (Dirichlet) boundary condition excludes the end-points from the update equation, so that $l \in \{1,\ldots,N-1\}$. For the free (Neumann) boundary condition, $l \in \{0,\ldots,N\}$. One important aspect of the free (Neumann) boundary condition is the implementation of virtual grid points in order to fix the slope.

Discretising the boundary conditions of eq. 4 and 5 yields:

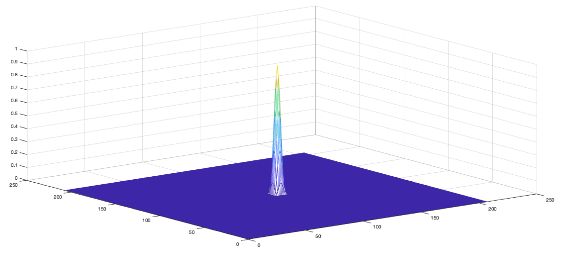


Fig. 4: Plucking of 2D Plate

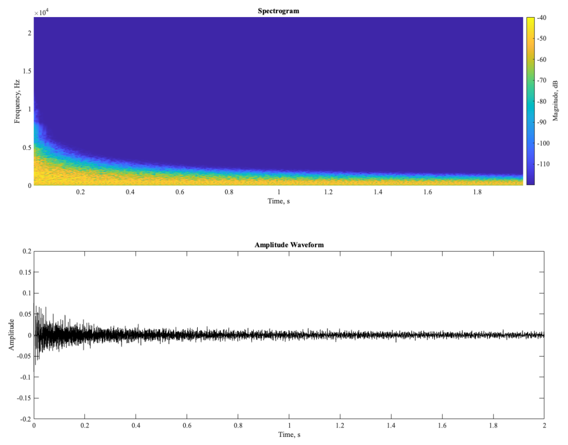


Fig. 5: Spectrogram of output vector

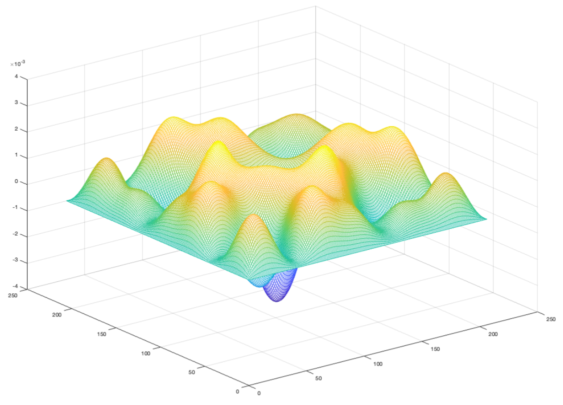


Fig. 6: Snapshot of active modes in 2D plate

$$u_0^n = u_N^n = 0 \quad \text{(Fixed, Dirichlet)} \tag{6}$$

$$\delta_x \cdot u_0^n = \delta_x \cdot u_N^n = 0 \quad \text{(Free, Neumann)} \tag{7}$$

III. Implementation

Scripts and code can be found on https://github.com/rallevondalle/PMSMiniProject. See Section IV Appendix A: Code for code.

References

[1] J. O. Smith, “Physical Modeling Using Digital Waveguides,” *Computer Music Journal*, vol. 16, no. 4, p. 74, 1992.

[2] S. Bilbao, *Numerical Sound Synthesis*. Chichester, UK: John Wiley & Sons, Ltd, Oct. 2009.

[3] S. Willemsen, *The Emulated Ensemble*. PhD thesis, Aalborg University, Copenhagen, July 2021.

IV. APPENDIX A: CODE

Scripts and code can be found on https://github.com/rallevondalle/PMSMiniProject

A. 2D Plate Matlab Implementation

```
\small
%2D Wave Guide
%% Initialise variables
fs = 44100;               % sample rate [Hz]
k = 1 / fs;               % time step [s]
M = 1;                    % mass
K = 1;
kappa = 0.01;             % set to smaller for spring, like kappa = 0,006
c = 300;                  % speed of sound in air, 342 m/s
Lx = 2;                   % length of x axis
Ly = Lx;                  % length of y axis
sigma0 = 0.8;             %
sigma1 = 0.005;           %

%Initialize grid space, calculated from the Courant-number (Stability condition)
h = sqrt(2*(c.^2*k.^2 + 4*sigma1*k));
Nx = floor(Lx/h);         %
Ny = floor(Ly/h);         %
h = Lx / Nx;              % assuming grid space is equal for both dimensions

lengthSound = 5*fs;       % length of the simulation in seconds

f0x = c / (2*Lx);         % fundamental frequency [Hz]
lambda = (c*k)/h;         % CFL condition, Courant-Friedrichs-Lewy
muSq = ((kappa*k)/h^2)^2; %stability constant, 0:1
pickupPosition = floor(0.2*(fs/Nx)); % Samplerate dependent pickup position,
%close to left (N = 0) position
pickSize = 1;             % size of pick from 1-10
%smaller number yields higher frequencies and vice versa

%% Stability check
stability = 1;
% if h >= c*k
%     stability = 1;
% else
%     stability = 0;
% end

%omega0 = 2 * pi * f0; % angular (fundamental) frequency [Hz]
%M = 100; % mass [kg]
%K = omega0^2 * M; % spring constant [N/m]

%Set boundary conditions (clamped, simply supported, free [wind instruments])
%Clamped l = {2,...,N-2}
%Unexpanded FD scheme is deltatt*uln = -k^2deltaxxxx*unl
%Expanded dxxxxunl= 1/h^4(ul+2 - 4ul+1 + 6ul - 4ul-1 + ul-2)
%Boundary condition uln = dxxuln = 0 at l=0,N --> un-1 = -un1 and uN+1 =
%-uN-1


% initialise matrices, not vectors in this case since we are in two
% dimension
uNext    = zeros(Nx+1,Ny+1);
u        = zeros(Nx+1,Ny+1);
uPrev    = zeros(Nx+1,Ny+1);
range    = zeros(Nx+1,Ny+1); %total grid space with boundary conditions
out      = zeros(lengthSound,1);


%Initial conditions, vector lenght of x, add raised cosine to middle,
%transpose and vector multiply to get a sqaure matrix size of your plate
pluckVector = zeros(Nx+1,1);
pluckPosition = floor((Nx/2)+5);
pluckSize = 5;
```

```
pluckVector(pluckPosition-pluckSize:pluckPosition+pluckSize) = rkHannWin(2*pluckSize);
pluckMatrix = pluckVector * transpose(pluckVector);
u = pluckMatrix;
uPrev = u;
%figure;
%mesh(u);                % visualize plucking

X = linspace(-1,1,Nx+1)'; Y = X;

visualizeOn = 0;         %visualize plate behaviour

% Simulation loop
if stability == 1
    %disp("System stable");
    %disp(pickupPosition);
    for n = 3:lengthSound %TIME ITERATION, LAPLACIAN DISCRETISATION

        uNext(2:Nx,2:Ny) = ...
                -uPrev(2:Nx,2:Ny) + 2*u(2:Nx,2:Ny) + lambda*lambda*(u(3:Nx+1,2:Ny) + ...
                u(2:Nx,3:Ny+1) - 4*u(2:Nx,2:Ny) + u(1:Nx-1,2:Ny) + u(2:Nx,1:Ny-1)) + ...
                ((2*sigma1*k)/(h.^2)) * (u(3:Nx+1,2:Ny) + u(1:Nx-1,2:Ny) + u(2:Nx,3:Ny+1) + ...
                u(2:Nx,1:Ny-1) - 4*u(2:Nx,2:Ny) - uPrev(3:Nx+1,2:Ny) - uPrev(1:Nx-1,2:Ny) - ...
                uPrev(2:Nx,3:Ny+1) - uPrev(2:Nx,1:Ny-1) + 4*uPrev(2:Nx,2:Ny)) ;
                %with frequency dependent damping

        if visualizeOn == 1
            Z = uNext;
            mesh(X,Y,Z);
            zlim([-0.5 0.5]);
            drawnow;
        end

        % Update output at specific position in space, simulating a pick-up (single point)
        out(n,:) = u(pluckPosition-10,pluckPosition-10);

        % Energy check (not implemented yet)
        % Kinetic energy
        %kinEnergy(n) = M / 2 * (1/k * (u - uPrev))^2;

        % Potential energy
        %potEnergy(n) = K / 2 * u * uPrev;

        % Total energy (Hamiltonian)
        %totEnergy(n) = kinEnergy(n) + potEnergy(n);

        % Update system states
        uPrev = u;
        u = uNext;
    end
else
    disp("System unstable, check variables");
end

%% OUTPUT TO WAVEFILE
mkdir(sprintf('audioExports'));

% TIME STAMP FOR FILE OUTPUT
filename = sprintf('audioExports/plate_%ihz_%s.wav',fs,datestr(now,'yyyymmdd-HHMMSS'));
disp(['Exported to ./' filename ' at ' num2str(fs) 'Hz']);

%%audiowrite(filename,out,fs);

%% PLOT

figure("Name","Time domain plot");
plot(out);
soundsc(out,fs);

%% PLOT SPECTROGRAM
```

```
wlen  = 1024;                     % window length
hop   = wlen/4;                   % hop size
nfft  = 4096;                     % number of fft points
N     = length(out);              % N data points
tWave = (0:N-1)/(fs);             % time vector for waveform plot

% perform STFT
win = blackman(wlen);
[S, f, t] = stft(out, win, hop, nfft, fs);

% calculate the coherent amplification of the window
C = sum(win)/wlen;

% take the amplitude of fft(x) and scale it, so not to be a
% function of the length of the window and its coherent amplification
S = abs(S)/wlen/C;

% correction of the DC & Nyquist component
if rem(nfft, 2)                       % odd nfft excludes Nyquist point
    S(2:end, :) = S(2:end, :).*2;
else                                  % even nfft includes Nyquist point
    S(2:end-1, :) = S(2:end-1, :).*2;
end

% convert amplitude spectrum to dB (min = -120 dB)
S = 20*log10(S + 1e-6);

% plot the spectrogram
figure(3)
subplot(2,1,1);
surf(t, f, S)
shading interp
axis tight
view(0, 90)
set(gca, 'FontName', 'Times New Roman', 'FontSize', 14)
xlabel('Time, s')
ylabel('Frequency, Hz')
title('Spectrogram')

hcol = colorbar;
set(hcol, 'FontName', 'Times New Roman', 'FontSize', 14)
ylabel(hcol, 'Magnitude, dB')

subplot(2,1,2);
plot(tWave,out,'black');
set(gca, 'FontName', 'Times New Roman', 'FontSize', 14)
xlabel('Time, s')
ylabel('Amplitude')
title('Amplitude Waveform')
ylim([-0.2 0.2]);


% Window functions
function [window] = rkHammingWin(size);
if size > 0
    M = size;
    w = .54 - .46*cos(2*pi*(0:M)'/(M-1));       %for 1 based systems 0:M, for 0 based systems 0:M-1
else
end
end

function [window] = rkHannWin(size)
if size > 0
    M = size;
    window = 0.5 * (1 - cos(2*pi*(0:M)'/M));    %for 1 based systems 0:M, for 0 based systems 0:M-1
else
end
end
```

```
% STFT FUNCTION
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%              Short-Time Fourier Transform            %
%               with MATLAB Implementation             %
%                                                      %
% Author: Ph.D. Eng. Hristo Zhivomirov        12/21/13 %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [STFT, f, t] = stft(x, win, hop, nfft, fs)

% function: [STFT, f, t] = stft(x, win, hop, nfft, fs)
%
% Input:
% x          - signal in the time domain
% win        - analysis window function
% hop        - hop size
% nfft       - number of FFT points
% fs         - sampling frequency, Hz
%
% Output:
% STFT       - STFT-matrix (only unique points, time
%              across columns, frequency across rows)
% f          - frequency vector, Hz
% t          - time vector, s

% representation of the signal as column-vector
x = x(:);

% determination of the signal length
xlen = length(x);

% determination of the window length
wlen = length(win);

% stft matrix size estimation and preallocation
NUP = ceil((1+nfft)/2);     % calculate the number of unique fft points
L = 1+fix((xlen-wlen)/hop); % calculate the number of signal frames
STFT = zeros(NUP, L);       % preallocate the stft matrix

% STFT (via time-localized FFT)
for l = 0:L-1
    % windowing
    xw = x(1+l*hop : wlen+l*hop).*win;

    % FFT
    X = fft(xw, nfft);

    % update of the stft matrix
    STFT(:, 1+l) = X(1:NUP);
end

% calculation of the time and frequency vectors
t = (wlen/2:hop:wlen/2+(L-1)*hop)/fs;
f = (0:NUP-1)*fs/nfft;

end
```