How to build a MATLAB demonstrator solving dynamical systems in real-time, with audio output and MIDI control

Tom Colinot[a,b], Christophe Vergez[b]
[a] Buffet Crampon, Mantes-la-Ville, France
[b] Aix Marseille Univ, CNRS, Centrale Marseille, LMA, Marseille, France

Summary

This paper explains and provides code to synthesize and control, in real-time, the audio signals produced by a dynamical system. The code uses only the Matlab programming language. It can be controlled with an external MIDI (Musical Instrument Data Interface) device, such as a MIDI keyboard or wind controller, or with mouse-operated sliders. In addition to the audio output, the demonstrator computes and displays the amplitude and fundamental frequency of the signal, which is useful to quantify the dynamics of the model. For the sake of this example, it is a type of Van der Pol oscillator, but more complex systems can be handled. The demonstrator holds potential for pedagogical and preliminary research applications, for various topics related to dynamical systems: direct and inverse bifurcations, transient effects such as dynamical bifurcations, artifacts introduced by integration schemes, and above all, the dynamics of self-sustained musical instruments.

1 Introduction

Autonomous dynamical systems are complicated objects to study and teach. Even some of the simplest ones to formulate are extremely unpredictable. The richness of this behavior is not encapsulated in the usual description of the permanent equilibrium points or periodic regimes [1, 2]. Some of their solutions are non-periodic, or coexist with other stable solutions [3]. This makes it difficult to predict which type of solution is obtained in any given situation. When the system parameters vary, complicated transient effects emerge, such as hysteresis cycles [4] or dynamical bifurcations [5].

Self-oscillating musical instruments such as wind instruments or bowed strings are modeled using autonomous dynamical systems [6]. Their example illustrates how transient effects are essential to a complete description of the system's real-life behavior, since they are experienced (at least) at the beginning and end of each note. In this spirit, it seems that a reasonable and compelling approach to experience and explore how a dynamical system reacts is implementing a virtual "musical instrument" demonstrator. By manipulating the control parameters, the user can see and hear phenomena typical of nonlinear systems in real-time, in a very controlled and repeatable environment. This is particularly relevant in all fields related to music, such as musical acoustics and instrument making. In these fields, one can go even further by linking the system's behavior to musical terms, such as intonation (flat, sharp), nuance (piano, forte) or transient dynamics (staccato, legato). The presented demonstrator aims to be as general as possible, meaning that the example model can be replaced by any simple dynamical system with only minor adjustments. While other possible environments for real-time sound synthesis exist, such as C++ (notably the JUCE library), Max/MSP, or Faust, this demonstrator is presented in a pure Matlab implementation, using only the Audio Toolbox (audio_system_toolbox, matlab, signal_blocks). This is advantageous for the many researchers who may

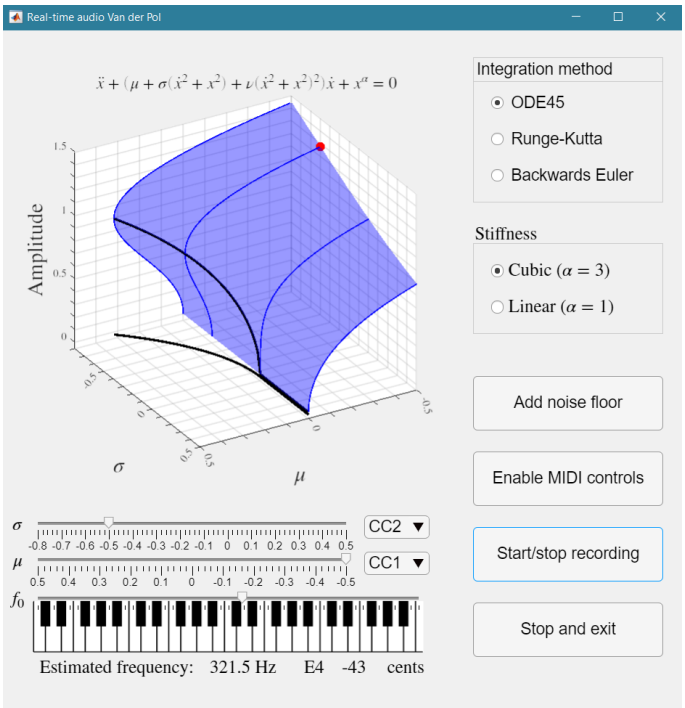

Figure 1: Interface of the real time dynamical system demonstrator.

want to reuse their preexisting codes, solvers, systems, analysis or display tools. Note that the built-in Matlab tools necessary were not available before R2016a (for the Audio toolbox and most of the elements of the user interface) and R2018a (for the MIDI message handling).

After a quick presentation of the dynamical system used in the demonstrator in Section 2, the implementation choices are detailed is Section 3. Ideally, the authors wanted this paper to follow the actual lines of codes implementing the demonstrator. Sadly, the complete code is too lengthy for a journal paper (notably due to display functions, UI building and general housekeeping). With the intent of being as explicit as possible, Section 3 is built around several code snippets addressing the main challenges of a real-time audio demonstrator. The first of these code excerpts is self-contained and functional, and outputs sound generated by a dynamical system in real-time. The structure of the rest of the section follows that of the code, which is:

- Initialization of the audio output object (the current example uses the `audiodevicewriter` object).
- Construction of the user interface: buttons, sliders and MIDI device handling.
- Execution of the synthesis loop, typically a while loop with generation of new sound samples, and output of the audio through the audio output object.
- Extraction of audio descriptors and display.

A link to the complete Matlab code is given in Appendix, as well as a compiled version with slightly faster reaction to user controls obtained with the Matlab compiler.

2 A simple system

For this demonstration, we use a modified Van der Pol oscillator with a quadratic and fourth power nonlinearity in the damping term. It can be seen as a simplified model of self-sustained musical instrument. Notably, some fingerings of the saxophone display a similar dynamic [7].

The system is studied in its linear stiffness form in [8]. It is very close to the normal form of the Bautin bifurcation [9]. As such, its behavior is well-known, but rich enough so that it illustrates a good number of phenomena typical of autonomous nonlinear systems. The governing equation is

$$\ddot{x} + \left[\mu + \sigma(\dot{x}^2 + x^2) + \nu(\dot{x}^2 + x^2)^2\right]\dot{x} + x^\alpha = 0. \quad (1)$$

Hereafter, we set $\nu = 0.5$ and $\alpha = 1$ (linear stiffness) or $\alpha = 3$ (cubic stiffness). The parameters μ and σ are controlled by the user. The amplitude of the oscillations can be approximated analytically as shown in [8], setting $x = X\cos(t)$, as

$$X = \sqrt{\frac{-\sigma \pm \sqrt{\sigma^2 - 4\mu\nu}}{2\nu}}. \quad (2)$$

In the range of parameters explored here, this approximation is extremely precise. This formula gives the blue surface in Figure 1. The system exhibits a Hopf bifurcation at $\mu = 0$, meaning that a periodic oscillation emerges from the equilibrium [10]. This corresponds to the point at which the linear damping coefficient changes sign. Hence, the linearized oscillator becomes active (energy is created) when $\mu < 0$. The Hopf bifurcation is supercritical for $\sigma > 0$ and subcritical for $\sigma < 0$. There is a saddle-node bifurcation at $\sigma^2/(4\nu)$ when $\sigma < 0$, marking the turning point where the unstable and stable periodic solutions collapse together [11]. The saddle-node bifurcation merges with the Hopf bifurcation at $\sigma = 0$, forming the codimension 2 Bautin bifurcation [12]. This structure implies a bistability zone, where both the equilibrium and an oscillating (periodic) regime are stable. In this zone, the user can experience the practical implications of bistability around an inverse Hopf bifurcation, such as hysteresis cycles or the impossibility to obtain an oscillation of arbitrarily low amplitude.

A modified version of Eq. (1) is used, in order to produce several notes without changing the dynamics of the continuous-time system. With a new parameter ω_0, the state-space representation of the system becomes

$$\dot{X} = \begin{pmatrix} \dot{x} \\ \dot{y} \end{pmatrix} = F(X) \quad (3)$$

$$= \omega_0 \begin{pmatrix} y \\ -x^\alpha - (\mu + (\sigma(y^2 + x^2) + \nu(y^2 + x^2)^2)y \end{pmatrix}. \quad (4)$$

The time domain integration of Eq. (1) is realized using three different discretization schemes. This demonstrate the versatility of the approach, and highlights the influence of the discretization scheme. The user can switch between schemes at any time. The simplest discretization scheme is the explicit Euler method [13], giving the $n+1$-th values of x and y as a function of the n-th:

$$x[n+1] = x[n] + \frac{\omega_0}{F_s} y[n] \quad (5)$$

$$y[n+1] = y[n] + \frac{\omega_0}{F_s}(-x[n]^\alpha - (\mu \quad (6)$$

$$+ (\sigma(y[n]^2 + x[n]^2) + \nu(y[n]^2 + x[n]^2)^2)y[n]). \quad (7)$$

The second method is a fourth-order Runge Kutta (RK4) integration scheme [14]. Using the F notation from Eq. 3, one computes the next sample by

$$X[n+1] = X[n] + \frac{1}{6F_s}\left(K_1 + 2K_2 + 2K_3 + K_4\right) \quad (8)$$

where

$$K_1 = F(X[n]) \qquad K_2 = F(X[n] + \frac{K_1}{2F_s}) \quad (9)$$

$$K_3 = F(X[n] + \frac{K_2}{2F_s}) \quad K_4 = F(X[n] + \frac{K_3}{F_s}). \quad (10)$$

The third and last method is Matlab's built-in `ode45` solver [15]. It is also a type of Runge-Kutta integration scheme but with a auto-adaptive time step.

3 Implementation of the demonstrator

This section describes the implementation of the main functionalities of the demonstrator. The complete source code can be downloaded from the repository in Zenodo https://doi.org/10.5281/zenodo.8170856 [16].

3.1 Real-time audio on Matlab

The current implementation uses the Audio toolbox object `audiodevicewriter`, which communicates with the audio driver of the computer. The Matlab version used during the writing of this article is 2021a. Fig. 2 is a self-contained, minimal working example using the system from section 2 solved with `ode45`. When running this code, please lower the volume as the sound can be quite loud.

Note that, especially on Windows, best results are obtained using ASIO drivers instead of the default audio driver. In that case, the sample rate and buffer size of the ASIO driver should be equal to those of the Matlab `audioDeviceWriter` object. However, numerous different ASIO drivers exist depending on each user's setup. Therefore, it is hard to provide a flexible and compact ASIO-based solution. Users of the code are encouraged to adjust the `audioDeviceWriter` parameters to their particular hardware and driver.

3.2 User interface

The user interface displayed in Figure 1 is comprised of a main display graph, three control sliders (μ, σ and $f_0 = \omega_0/(2\pi)$), two button groups setting the integration scheme and the stiffness exponent α, and four buttons for other user actions.

3.3 Musical instrument control through MIDI

A natural way to control the demonstrator is through the MIDI protocol. This is especially relevant for any kind of music-related interpretation, as external MIDI controllers allow to control the demonstrator like a keyboard synthesizer or a wind instrument. In a more general context, MIDI controllers allow for a more fluid control than a mouse and slider. For instance, MIDI controllers facilitate the simultaneous variation of several parameters.

Matlab's audio toolbox support MIDI through the `mididevice` object. First, the `mididevice` object is created based on a user input given through the `MIDIlistbox` object.

```
midicontroller = ...
    mididevice('Input',MIDIlistbox.Value);
```

Then, at every iteration of the synthesis loop (i.e. before each audio buffer is filled with samples), the pending MIDI message are gathered using

```
msgs = midireceive(midicontroller);
```

This gives an array of `midimsg` objects. They signal control parameter changes or note changes, depending on their type. This information can be retrieved by accessing the `Type` property of the `midimsg` and comparing it to another string, for instance 'NoteOn' or 'ControlChange' (CC). This is slower (sometimes by a factor of ten) than directly reading and comparing the bytes of the messages, which hold the same information. The gain in speed is especially interesting in the case of a wind controller, which sends ControlChange messages very often to translate the blowing pressure of the musician. An array of the message bytes is created by

```
msgsbytes = vertcat(msgs.MsgBytes);
```

and then parsed using the first byte as the message type identifier (176 for ControlChange, 144 for NoteOn). In the case of ControlChange, the second byte in the array indicates the CC number. Different CC numbers can be linked to different parameters. For example, parameter μ is linked to CC number `muCC` (2 by default), which is parsed from the MIDI message bytes by

```
imsgCCmu = find((msgsbytes(:,1)==176)...
       &(msgsbytes(:,2)==muCC),1,'last');
```

In order to apply only the most recent user-provided command, only the last message is read. The new value of the control is contained in the third byte of that message (at index `imsgCCmu` in the array):

```
newCCmu = double(msgsbytes(imsgCCmu,3));
```

This value, scaled between the control parameter limits, gives the new control parameter value.

4 Synthesis loop

Each iteration of the synthesis loop produces enough audio samples to fill one audio buffer. Its structure in pseudocode is

```
while (stopbutton is not pushed)
    Read user controls
    Solve equation during one audio buffer
    Format solution as audio output
    Extract signal descriptors from solution
    Update display
```

```
nu = 0.5; sigma = -0.5; mu = -0.5;  Nbuf = 512; w0 = 2*pi*440; Fs = 44100;
ADW = audioDeviceWriter(Fs);
X = [1;1]; t = (0:Nbuf)/Fs;
while 1,
    [t,Xs]=ode45(@(t,X) w0*[X(2);-X(1)-(mu+sigma*(X(2)^2+X(1)^2)+nu*(X(2)^2+X(1)^2)^2)*X(2)],t,X.');
    X = Xs(end,:);
    ADW(Xs(2:end,:));
end
```

Figure 2: Minimal working example solving the oscillator of section 2 in real-time, and outputting the audio stream. This code can be copy-pasted directly to the Matlab command window (depending on the pdf font used the " ^ " power character needs to be manually rewritten). It is also provided as an M-file in the code archive that can be downloaded in the Appendix.

```
    Record solution
    Output sound
    Check pending displays or callbacks
end
```

The following subsections detail each line of this pseudocode block.

4.1 Read user controls

User controls are read using either the MIDI messages or the sliders. There are three methods to assign the value of a slider to a variable. First, it is possible to check the Value property of the slider on each loop iteration. This is done for the f_0 parameter. This solution is very close to using the `ValueChangedFcn` callback. In both cases, the changes are applied when the user releases the slider thumb. The third solution is the slider callback function `ValueChangingFcn`, which is called while the user moves the slider. This solution is necessary to render progressive variations of the parameters. The control parameters μ and σ are updated with this method. This is useful to follow a quasi-static path along the bifurcation diagram, or execute a slow attack through the Hopf bifurcation.

The button values are read and stored in separate variables, to be used in the loop.

4.2 Solve equation during one audio buffer

Depending on the structure of the solver function, this step can take two forms. If the solver sets its own time-step, and a fortiori if it is auto-adaptive (like `ode45`), the solver function is called once to generate the total number `Nbuf` of samples in one audio buffer. This is implemented as

```
[~,Xts] = ode45(@(t,Xt) VanDerPol5_odefun(...
 Xt,t,mu,nu0,sigma,2*pi*f0),(0:Nbuf)/Fs,X.');
X = Xts(end,:);
positions = Xts(2:end,1);
speeds = Xts(2:end,2);
```

Note that the initial condition `X` is returned as the first line of the solution `Xts`. However, it is (by definition) the last line of previous solution. Repeating it twice in the audio stream causes clicks and artifacts. Therefore, `ode45` is called for `Nbuf+1` time steps, and only the last `Nbuf` constitute the audio output.

If, on the contrary, the solver simply gives $X[n+1]$ as a function of $X[n]$ (like the RK4 and explicit Euler schemes), it is called `Nbuf` times. Then, the solving step is

```
for ibuf = 1:Nbuf
    X_np1 = VanDerPol5_backwardsEuler(...
            X,mu,nu0,sigma,2*pi*f0,Fs);
    X = X_np1;
   positions(ibuf) = X(1); speeds(ibuf) = X(2);
end
```

We use a test to estimate the maximum number of oscillators that can be integrated simultaneously. The test is done with no user interface or separate display. It uses 44.1 kHz sample rate and a 512 sample buffer. On the laptop this was implemented on, between 3 and 9 `ode45`-solved oscillators can run in parallel while keeping a fluid audio flux (depending on power consumption options). Between 4 and 12 parallel oscillators run with the RK4 solver. Using a simpler explicit Euler scheme allows between 28 and 82 oscillators to run in parallel. This gives an idea of the headroom of this architecture to accommodate bigger systems. As this result heavily depends on the user's hardware, the code to reproduce this test is given in the code archive linked in the Appendix, so each user can know their potential for more complex systems.

4.3 Format solution as audio output

The solution of the equation must fit inside a `Nbuf`-by-two matrix, which is passed as argument to the `audiodevicewriter` object. Here, for this simple system, minimal processing is applied. The solution is scaled by `Xmax`, an analytical estimate of the maximum amplitude for the considered control parameter range. The left and right channel are passed x and y respectively. This way, a direct xy plot of the audio output represents the phase space of the oscillator.

```
audioout = [positions(:) speeds(:)]/Xmax;
```

In a more general case, it can be useful to listen to certain physical variables rather than others, or pro-

cess them in a specific manner (filtering or nonlinear processing) for illustrative or aesthetic purposes.

4.4 Extract signal descriptors from solution

Only basic signal descriptors are extracted in this demonstrator: RMS amplitude (or rather, mean distance to the origin in the phase space) and fundamental frequency. The RMS amplitude is used as the main display indicator. For this system, it is sufficient to differentiate solution branches and locate bifurcations. The fundamental frequency estimate is computed by a simplistic algorithm based on zero-crossings. It helps to quantify the detuning effect of the different integration schemes, and of the cubic stiffness.

4.5 Update display

Any systematic real-time display concurrent with an audio process on Matlab needs to be kept as light as possible to not perturb the audio flux. This demonstrator updates a `plot` with a single point, and an `animatedline` object inside the loop. These graphical objects represent the current and recent RMS amplitude of the solution.

4.6 Record solution if necessary

A button on the interface records the descriptors, control parameters and button values in a structure once per loop iteration. In the present demonstrator, no variable at audio rate is recorded. This prevents excessive memory usage and disk access in the event of a long recording. The data is saved in a mat-file. It is also used as soon as the recording is stopped to provide a multipurpose plot designed to support a quick analysis of the results. An example of this plot is displayed in Figure 3, where the effect of the integration scheme on fundamental frequency is illustrated. Starting with the RK4 solver, the following parameter variation is applied: beginning at $\mu = 0.5$ and $\sigma = 0.5$, the μ parameter is decreased slowly to its minimum $\mu = -0.5$, followed by parameter σ which also decreases until around $\sigma = -0.6$. At this point, the RMS value of the oscillation is at its highest point. The parameters σ and then μ are then slowly brought back to their initial values. This scenario is repeated with ODE45, starting from approximately 22s, and with the explicit Euler scheme, starting from approximately 39s. One can see on the fundamental frequency subplot that the RK4 scheme provides the most consistent fundamental frequency, followed closely by the ODE45 solver where variations do not exceed 1 Hz around the 440 Hz expected frequency (i.e. the eigenfrequency of the linearized oscillator). However, the explicit Euler scheme entails considerable pitch flattening when the signal amplitude increases, down to about -6 Hz (about -20 cents) from the expected frequency.

4.7 Output sound

The `audiodevicewriter` object is used to output audio. It is also responsible for the scheduling of the loop.

```
ADW(audioout);
```

4.8 Check pending displays or callbacks

The execution of user interface object callbacks and refreshing of the display is ensured by the command `drawnow limitrate` placed at the end of the loop. A simple `drawnow` would degrade the audio output by introducing too many pauses, but `drawnow limitrate` limits the number of pauses to 20 per second. This is essential to keep a robust audio stream.

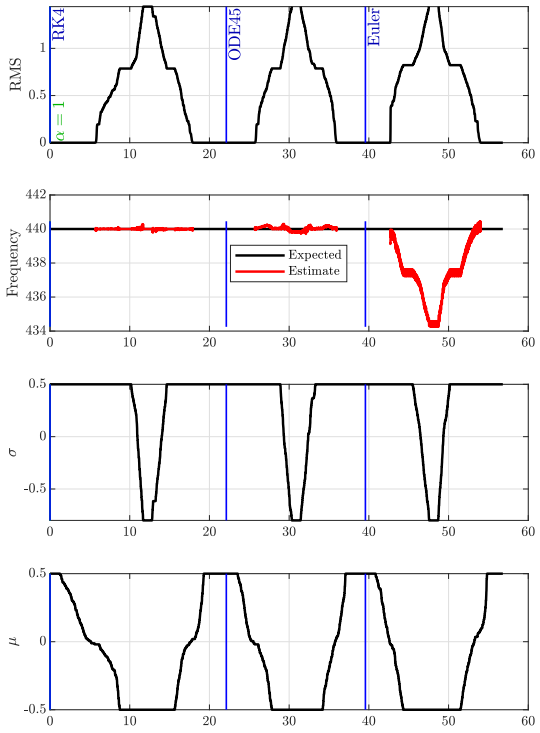

Figure 3: Example of the multipurpose plot created at the end of each recording, illustrating the fundamental frequency variation due to different integration schemes. See section 4.6 for details.

5 Video demonstration

The video, also linked in the caption of Fig 4, illustrates possible uses of the demonstrator, and showcases the fluidity of the controls.

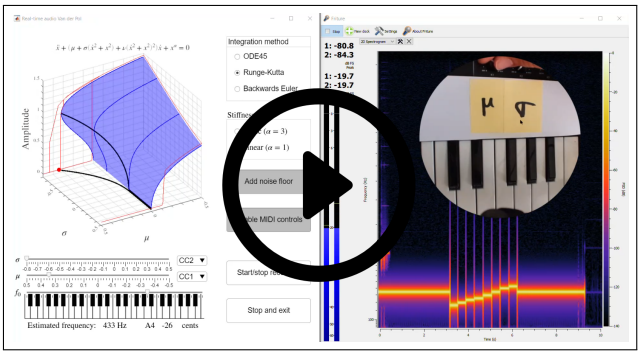

Figure 4: A snapshot of the illustrative video linked at https://youtu.be/_ExgRsgB7wc (until a more stable host is determined).

6 Conclusion

The presented demonstrator holds a lot of potential for teachers and researchers combining dynamical systems with audio engineering or music. In terms of teaching, it illustrates transient effects in direct and entertaining ways. It can also be useful for proof of concepts, to quickly assess the behavior of a dynamical system, or to compare two slightly different versions (parameter values, physical hypotheses or integration scheme).

Because it relies solely on Matlab, any researcher that has mainly been coding in Matlab can reuse their usual tools. In particular, we show that a continuous-time formulation of a system can be sufficient to produce sound, simply leaving the integration to a built-in method such as `ode45`. There is also reasonable headroom to adapt the demonstrator to a more complex system, especially if one is willing to simplify the integration scheme.

In music-related fields, this kind of demonstrator is all the more interesting as it bridges the gap between the physical model of an instrument and the actual instrument, by allowing to control a model with any musical controller supporting the MIDI protocol. The authors strongly believe in the potential of the real-time physical model control as a way to contribute to an objective definition of the 'playability' or 'ease of playing' of a musical instrument.

Acknowledgments

This study has been supported by the French ANR LabCom LIAMFI (ANR-16-LCV2-007-01). The authors are grateful to Teodor Wolter for proofreading the English.

Data Availability Statement

The source code and .exe installer file for the demonstrator, as well as M-files reproducing results for sections 3.1 and 4.2, can be downloaded from the repository in Zenodo https://doi.org/10.5281/zenodo.8170856.

References

[1] N. M. Krylov and N. N. Bogoliubov, Introduction to non-linear mechanics. No. 11, Princeton university press, 1950.

[2] R. Seydel, Practical bifurcation and stability analysis, vol. 5. Springer Science & Business Media, 2009.

[3] C. Grebogi, E. Ott, and J. A. Yorke, "Chaos, strange attractors, and fractal basin boundaries in nonlinear dynamics," Science, vol. 238, no. 4827, pp. 632–638, 1987.

[4] T. Tachibana and K. Takahashi, "Sounding mechanism of a cylindrical pipe fitted with a clarinet mouthpiece," Progress of Theoretical Physics, vol. 104, no. 2, pp. 265–288, 2000.

[5] B. Bergeot and C. Vergez, "Analytical expressions of the dynamic hopf bifurcation points of a simplified stochastic model of a reed musical instrument," Nonlinear Dynamics, vol. 107, pp. 3291–3312, 2022.

[6] A. Chaigne and J. Kergomard, Acoustique des instruments de musique (Acoustics of musical instruments). Belin, 2008.

[7] T. Colinot, Numerical simulation of woodwind dynamics: investigating nonlinear sound production behavior in saxophone-like instruments. PhD thesis, Aix-Marseille Université, 2020.

[8] D. Dessi, F. Mastroddi, and L. Morino, "A fifth-order multiple-scale solution for hopf bifurcations," Computers & structures, vol. 82, no. 31-32, pp. 2723–2731, 2004.

[9] J. Guckenheimer and Y. A. Kuznetsov, "Bautin bifurcation," Scholarpedia, vol. 2, no. 5, p. 1853, 2007.

[10] Y. A. Kuznetsov, "Andronov-hopf bifurcation," Scholarpedia, vol. 1, no. 10, p. 1858, 2006.

[11] Y. A. Kuznetsov, "Saddle-node bifurcation," Scholarpedia, vol. 1, no. 10, p. 1859, 2006.

[12] W. Beyn, A. Champneys, E. Doedel, W. Govarets, U. Kuznetsov, A. Yu, and B. Sandstede, Handbook of Dynamical Systems (Vol 2), chapter Numerical Continuation, and Computation of Normal Forms. Elsevier, 2002.

[13] J. C. Butcher, Numerical methods for ordinary differential equations. John Wiley & Sons, 2016.

[14] J. C. Butcher, "A history of runge-kutta methods," Applied numerical mathematics, vol. 20, no. 3, pp. 247–260, 1996.

[15] L. F. Shampine and M. W. Reichelt, "The matlab ode suite," SIAM journal on scientific computing, vol. 18, no. 1, pp. 1–22, 1997.

[16] T. Colinot and C. Vergez, "Dynamical system audio demonstrator," 2023. [Code]. GitHub. https://github.com/Tom-Colinot/Dynamical-System-Audio-Demonstrator/.

List of Figures