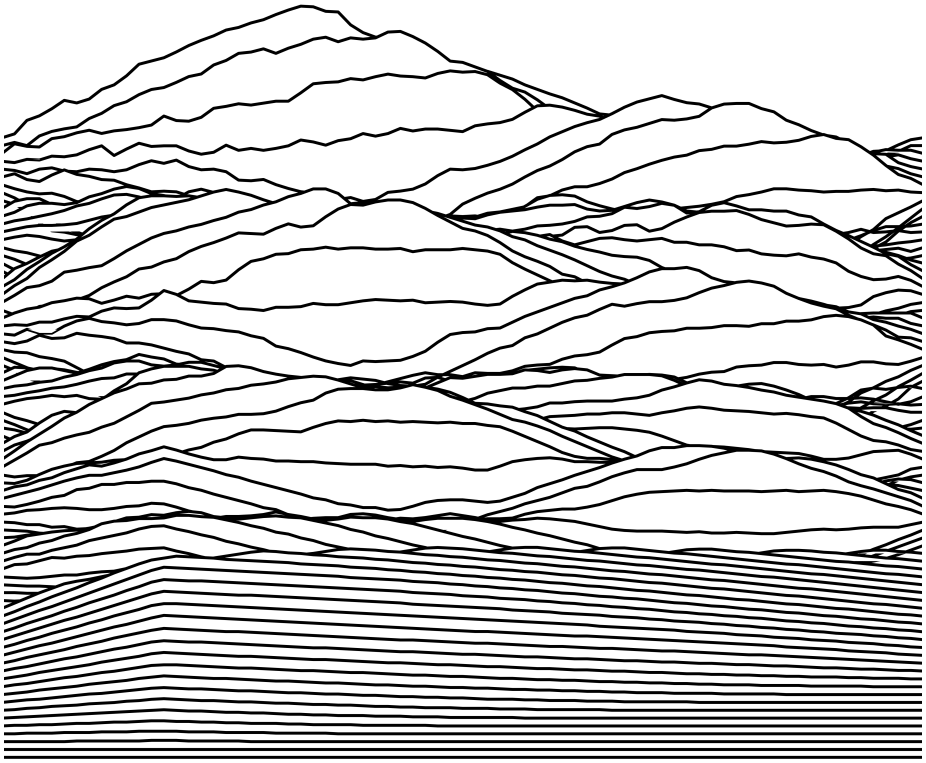
Master's Thesis:
An Interactive Musical Installation Using Physical Modeling and Algorithmic Composition

P. J. Christensen

May 2020



AALBORG UNIVERSITET
KØBENHAVN

IT and Design
Aalborg University
https://aau.dk

Title
Master's Thesis: An Interactive Musical Installation Using Physical Modeling and Algorithmic Composition

Theme
Sound and Music Computing, Physical Modeling for Sound Synthesis, Algorithmic Composition

Project Period
Spring semester 2020

Participants
Pelle Juul Christensen

Supervisors
Stefania Serafin
Silvin Willemsen

Page Count
61

Abstract
The Danish Music Museum strives to offer its visitors an engaging musical experience, which they achieve, in part, by exhibiting interactive installations. This report covers the design of such an installation. A user role framework is used in a review of the state of the art of musical installations, and subsequently to arrive at a design for the installation. Finite-difference time-domain techniques are used to create physical models of a bowed string and struck pitched percussion, which serves as the auditory part of the installation. Then, the history of algorithmic composition is covered and leads to the explanation of the current composition, whose main elements are split-note rhythm generation and tone row techniques. User input is provided by a Leap Motion hand tracker that allows the user to control the rhythmic and melodic content and dynamics of the piece in real-time. Prospects of future work are discussed and finally, the project is wrapped with a conclusion.

Typeset with Computer Modern using LaTeX.
Front page figure: the vibrations of a bowed stiff string over time.

"With the aid of electronic computers the composer becomes a sort of pilot: he presses the buttons, introduces coordinates, and supervises the controls of a cosmic vessel sailing in the space of sound, across sonic constellations and galaxies that he could formerly glimpse only as a distant dream."

— Iannis Xenakis in Formalized Music, 1971

Contents

Chapter 1

Introduction

This report describes the development of an interactive musical installation commissioned by the Danish Music Museum (henceforth The Museum). The Museum did not give any requirements for the installation (from now on the Installation) except that it should fit in the provided exhibition space and preferably be interactive. Thus, the design and purpose of the Installation were left for me to decide. A demonstration video of the installation software can be found online[1].

The Installation should offer an engaging auditory, visual, and haptic experience. The museum wanted an interactive installation because a lot of their exhibition is currently not, and interactivity might increase engagement.

To analyse the interactivity of musical installations I will talk about them in terms of the following three user roles: the composer, the performer, and the conductor.

As the composer, the user would choose *what* music would be played. Assuming that most of the museum visitors are not musically trained, free composition would be inaccessible, and therefore the role can only work if the compositional choices made available to the user are reduced to high-level structural ones. As the creator of the Installation, I would, as a composer, like to retain some aesthetic authority over the piece, which is done by letting the user take the high level, structural decisions, while the details are predetermined.

The possibility of a performer role is limited because of the visitors' assumed lack of musical training, and one cannot expect an average visitor to pick up a novel instrument and play it right away. The difficulty of performing on an instrument-like setup might decrease the user's engagement, so the details of the performance should be left to the Installation, i.e., the computer.

As the conductor, the user would make high-level decisions about *how* the music should be played. This role is ideal because it offers an engaging real-time task that could be simplified enough that the users could approach it without prior training.

[1] `https://youtu.be/WifqiApaw14`

1.1 State of the Art

The Museum already contains a few installations or installation-like instruments. One of these, the mash machine[2], allows its user(s) to create music by selecting loops to be played from a pool of pre-composed loops. Which loops to play are chosen by placing plastic blocks with a visual identifier on a glass surface. Upon recognizing the identifier, the Mash Machine will start playing the corresponding loop. Audio effects are applied to the loop based on the location of the block. The Mash Machine corresponds mostly to the conductor role — most of the musical material is predetermined, and the user only makes structural decisions.

Similar to the Mash Machine instrument is the Reactable[3], not present at the Museum, which also uses plastic blocks on a glass surface, but with visuals projected on the surface for real-time visual feedback. The blocks of the Reactable can represent various DSP or musical objects, such as an oscillator, filter, sequencer, and loop player. Blocks placed in close proximity are patched together, similar to a modular synthesizer. The Reactable is versatile, and the user's role can be both composer, conductor, and performer, and additionally also sound designer.

Also present at the Museum is the Omni[4], a large, circular, dome-shaped instrument covered in colored ceramic tiles. By hitting a tile with their hand, the user will trigger a loop. Tiles closer to the center of the instrument starts longer loops, while those at the perimeter start shorter loops or one-shot sounds. The user role of the Omni is somewhere between conductor and performer.

Both the Mash Machine, Reactable, and Omni seem to have been created with collaboration in mind. The Reactable and Omni are both round, so multiple people can stand around them and participate in the music-making. The same goes for the Mash Machine, except in a square form factor, catering to four participants.

Another music museum, the Haus der Musik in Vienna, seem to possess several custom made installations. While the Danish Music Museum mostly exhibits ancient and rare instruments, the Vienna music museum focuses on communicating the Viennese music's history and providing an interactive experience. A part of their permanent exhibition, Sonotopia, seems to contain multiple installations illustrating various acoustic phenomena and some more kid-oriented ones[5]. The Vienna Museum also has *The Virtual Conductor* installation[6], in which the visitor takes on the role of the conductor of the Vienna Philharmonic, and can control the tempo and dynamics of famous pieces recorded by using a virtual baton. The Virtual Conductor is reminiscent of Max Mathew's Radio Baton and conductor program [18][7], which lets the user control

[2] `https://www.mashmachines.com/`

[3] `http://reactable.com/`

[4] I was not able to find a reference for the Omni, but a picture is available on the museum's website `https://en.natmus.dk/museums-and-palaces/the-danish-music-museum/`

[5] `https://www.youtube.com/watch?v=oRTe4BRpOGc`

[6] `https://www.hausdermusik.com/en/museum/4-etage-der-virtuelle-dirigent/`

[7] `https://www.youtube.com/watch?v=3ZOzUVD4oLg`

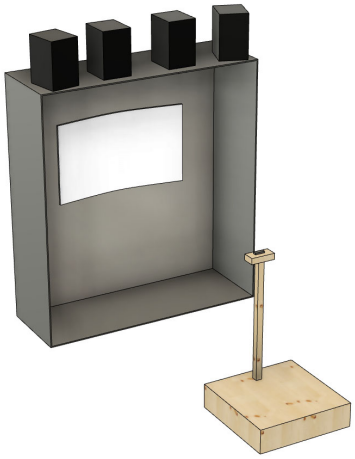


Figure 1.1: An early 3D concept drawing of the envisioned Installation featuring the podium, a large screen inside a display case, and four speakers now replaced by headphones.

the tempo and dynamics of the MIDI playback of a piece.

1.2 Installation Design

The final user role is a combination of the composer and conductor roles. How the user precisely controls the music is covered in section 3.1. Because of time limitations, I wanted to build as little hardware as possible, which meant going with consumer products whenever practical. Inspired by the movements of a real conductor, the main interface would be the position user's hands, tracked by some sensor — a task done expertly by Ultraleaps Leap Motion sensor[8]. An advantage of the Leap Motion sensor is that the user does not have to touch anything, making the Installation less prone to breaking. To mount the Leap Motion, and as a visual gimmick, a podium, like the one a real conductor uses, was envisioned. An early concept drawing of the Installation can be seen in Figure 1.1.

To guide the user, and to add visual interest, the Installation should visualize the user interaction and modeled instruments. Such visualization could be done using one or more big screens showing software-generated visuals, coupled with the sound synthesis application. Due to time limitations, no work was done on visualization except for a developer interface.

Because other instruments and installations occupy it, the room in which the Installation will be located is altogether saturated with sound. Therefore, the best option for sound reproduction is headphones, which might also offer

[8] `https://www.ultraleap.com/product/leap-motion-controller/`

a more intimate experience to the user. By providing multiple headphones, multiple users would be able to listen to the Installation at the same time.

To produce sound, I knew I wanted to use physical modeling because, having worked with physical modeling before, I was intrigued to see how musical I could make the models sound. I ended up using three voices: a high violin-like sound, a deep cello-like sound, and a bright glockenspiel-like sound.

1.3 Document Summary

Chapter 1. introduces the project hand, which is an installation made in collaboration with the Danish Music Museum. Using a framework of composer, performer, and conductor user roles, several state-of-the-art musical installations are analyzed. Afterward, the design of the Installation is presented.

In Chapter 2 the two instrument models are presented, and a review of the theory concerning the finite-difference time-domain method for physical modeling provides a base for the chapter. Theory in hand, two instrument models, a bowed string and struck bar, are introduced and covered in detail.

Chapter 3 explains what music the Installation is going to be playing. A review of ancient and modern mechanical composition methods culminates in the presentation of the musical material used in the Installation. A section on mapping and interactivity describes how the user may interact with the instrument models and composition, and after that, the measures taken to achieve expressive playback are outlined. To aid the composition task, a music notation language, Munola, is developed and its syntax and semantics are elaborated on.

Lastly, Chapter 4 contains perspectives on future work related to the Installation and concludes with a summary of the project.

Chapter 2

Instrument Modeling

The primary sound for the Installation is a bowed string sound, reminiscent of a violin or cello, which provides a melodic and harmonic base. A pitched struck percussion instrument, like a marimba or vibraphone, was added to add timbral contrast and rhythmic interest. This chapter will cover the theory and implementation of these two kinds of virtual instrument using physical modeling for sound synthesis.

Many physical modeling methods exist, such as modal synthesis, digital waveguides, banded waveguides, and mass-spring systems. This project employs finite difference time domain (FDTD) techniques. See [24] for a review of the various methods. Unlike most other methods, FDTD simulates the physics of vibrating systems directly by implementing the differential equations describing their dynamics. By doing so, FDTD methods offer control of the systems in terms of physical parameters and make possible rich, nonlinear interactions.

Note that the goal of the presented models was not to achieve supreme accuracy, but rather to create algorithms that would sound great, be easy to tune and interact with, and suit the aesthetic of the Installation.

2.1 Introduction to Finite Difference Schemes

FDTD methods are a way of numerically solving differential equations by discretizing time and space (and any other dimensions) and approximating their derivatives. For example, a first-order derivative can be approximated by

$$\frac{d}{dt}u(t) \approx \frac{u(t+k) - u(t)}{k}, \tag{2.1}$$

where k is some small value. As k approaches zero, the approximation becomes more accurate. In fact, in the limit as $k \to 0$, the above is exactly the definition of the first-order derivative.

When solving a system of space and time, e.g. $u(x,t)$, it is first discretized by sampling both dimensions at regular intervals: k for time and h for space.

This is expressed by the notation

$$u_l^n = u(lh, nk), \tag{2.2}$$

where u is now a grid function indexed in time by n and in space by l.

As a notational device, one may define several *finite difference operators* [5]; approximations to first order temporal derivatives, the backwards, forwards, and centered difference operator, are defined as

$$\delta_{t-} u_l^n \triangleq \frac{1}{k}(u_l^n - u_l^{n-1}) \tag{2.3}$$

$$\delta_{t+} u_l^n \triangleq \frac{1}{k}(u_l^{n+1} - u_l^n) \tag{2.4}$$

$$\delta_{t\cdot} u_l^n \triangleq \frac{1}{2k}(u_l^{n+1} - u_l^{n-1}). \tag{2.5}$$

The definitions for spatial derivatives are analogous, except they use l and a spatial sample period h, e.g.,

$$\delta_{x-} u_l^n \triangleq \frac{1}{h}(u_l^n - u_{l-1}^n). \tag{2.6}$$

Lower order operators can be combined to form higher order approximations, e.g for the second order derivative:

$$\delta_{tt} u_l^n \triangleq \delta_{t-}\delta_{t+} u_l^n = \frac{1}{k^2}(u_l^{n+1} - 2u_l^n + u_l^{n-1}). \tag{2.7}$$

In some cases one will need *averaging operators* [5] to ensure stability the solution, these, the forwards, backwards, and centered averaging operator, are defined as

$$\mu_{t-} u_l^n \triangleq \frac{1}{2}(u_l^n + u_l^{n-1}) \tag{2.8}$$

$$\mu_{t+} u_l^n \triangleq \frac{1}{2}(u_l^{n+1} + u_l^n) \tag{2.9}$$

$$\mu_{t\cdot} u_l^n \triangleq \frac{1}{2}(u_l^{n+1} + u_l^{n-1}), \tag{2.10}$$

and can likewise be combined to form higher order averaging operators.

As an example, take the differential equation for a mass spring system

$$M\frac{d^2}{dt^2}u = -Ku, \tag{2.11}$$

where M is the mass, u is the displacement of the mass, and K is the spring constant. Subscript notation is often used to shorten expression, e.g. the above becomes

$$Mu_{tt} = -Ku, \tag{2.12}$$

where u_{tt} is the second order temporal derivative of u. To discretize this, u is sampled in time and the derivatives are replaced with suitable difference operators, to get a *finite difference scheme* (FDS) for the equation

$$M\delta_{tt}u^n = -Ku^n. \tag{2.13}$$

Since the system is point-like, there is no need to discretize space, hence why l does not appear above. After expanding the difference operators, the above becomes

$$M\frac{1}{k^2}(u^{n+1} - 2u^n + u^{n-1}) = -Ku^n, \tag{2.14}$$

which may then be solved for u_l^{n+1}

$$u^{n+1} = -k^2\frac{K}{M}u^n + 2u^n - u^{n-1}. \tag{2.15}$$

This expression is the *update rule* for the solution which can be used to compute the next state u^{n+1} from the current and previous state. In this example, u^{n+1} could be solved *explicitly*. In some cases, often due to nonlinearities, the scheme can only be solved *implicitly*, which requires the use of iterative numerical methods.

2.1.1 Stability

A topic of great importance when dealing with FDSs is ensuring the stability of the scheme. A scheme is said to be stable if we can choose a set of initial conditions for which we can prove that the state of the system stays within some bound. Several ways of proving stability exist, and a common method is to perform *frequency domain analysis* in which one assumes a test solution of the form of a complex phasor and proceeds to show bounds for that solution [5, chapter 2, 3 and 5]. When applied to distributed systems, such as the 1D wave equation, this is known as *Von Neumann Analysis*. Such methods work great for linear systems, but cannot be applied to non-linear systems in general.

A more general method is the use of energetic techniques that use an expression for the energy of a system to prove bounds for the solution [5, chapter 2]. The mass-spring system of the previous section will be used as an example.

A good place to start is to show that energy is conserved for the continuous system. Using the mass-spring system of Equation (2.12), move all terms to the left hand side and multiply by u_t to get

$$Mu_tu_{tt} + Ku_tu = 0. \tag{2.16}$$

Now apply the identities

$$u_tu_{tt} = \left(\frac{1}{2}(u_t)^2\right)_t \qquad u_tu = \left(\frac{1}{2}(u)^2\right)_t \tag{2.17}$$

to get

$$\left(\frac{M}{2}(u_t)^2 + \frac{K}{2}(u)^2\right)_t = 0. \tag{2.18}$$

The inner parts of the parenthesis may be identified, respectively, as the kinetic and potential energy of the system, that is

$$\mathfrak{K} = \frac{M}{2}(u_t)^2 \qquad\qquad \mathfrak{U} = \frac{K}{2}(u)^2. \tag{2.19}$$

The total energy, also called the Hamiltonian energy, is thus

$$\mathfrak{H} = \mathfrak{K} + \mathfrak{U}. \tag{2.20}$$

Equation (2.18) then shows that $\mathfrak{H}_t = 0$, which means that the total energy of the system is unchanging and thus conserved.

A similar procedure can be done on the FDS to get a discrete energy expression [5, chapter 3]. Multiply the scheme of Equtaion (2.13) by $\delta_{t\cdot}u$ and use the identities

$$(\delta_{t\cdot}u^n)\delta_{tt}u^n = \delta_{t+}\left(\frac{1}{2}(\delta_{t-}u^n)^2\right) \qquad (\delta_{t\cdot}u^n)u^n = \delta_{t+}\left(\frac{1}{2}u^n u^{n-1}\right) \tag{2.21}$$

to get

$$\delta_{t+}\left(\frac{M}{2}(\delta_{t-}u^n)^2 + \frac{K}{2}u^n u^{n-1}\right) = 0, \tag{2.22}$$

where

$$\mathfrak{k} = \frac{M}{2}(\delta_{t-}u^n)^2 \qquad\qquad \mathfrak{u} = \frac{K}{2}u^n u^{n-1} \tag{2.23}$$

are discrete approximations of the kinetic and potential energy. This means the we have the discrete Hamiltonian $\mathfrak{h} = \mathfrak{k} + \mathfrak{u}$ and $\delta_{t+}\mathfrak{h} = 0$. Energy is thus, in some discrete sense, conserved, however, to prove that u stays bounded we must show that the energy expression is non-negative [5, 2].

An equation of the quadratic form

$$f(x, y) = x^2 + y^2 + 2axy \tag{2.24}$$

will be a positive definite paraboloid if $|a| < 1$ and non-negative for $a = 1$. For $|a| > 1$ the equation is hyperbolic. This is illustrated in Figure 2.1. Observe that by tracing a level curve at some constant value $f(x, y) = c$ we produce, in the case $|a| < 1$, an ellipse. As long as $f(x, y)$ is constant or decreasing, x and y will be bounded by this ellipse. In the case $|a| > 1$ the level curve is not closed and we cannot make a statement about bound for x and y.

Going back to the energy definition from Equation (2.22), the operators can be expanded to get

$$\frac{M}{2}(\frac{1}{k}u^n - u^{n-1})^2 + \frac{k}{2}u^n u^{n-1}, \tag{2.25}$$

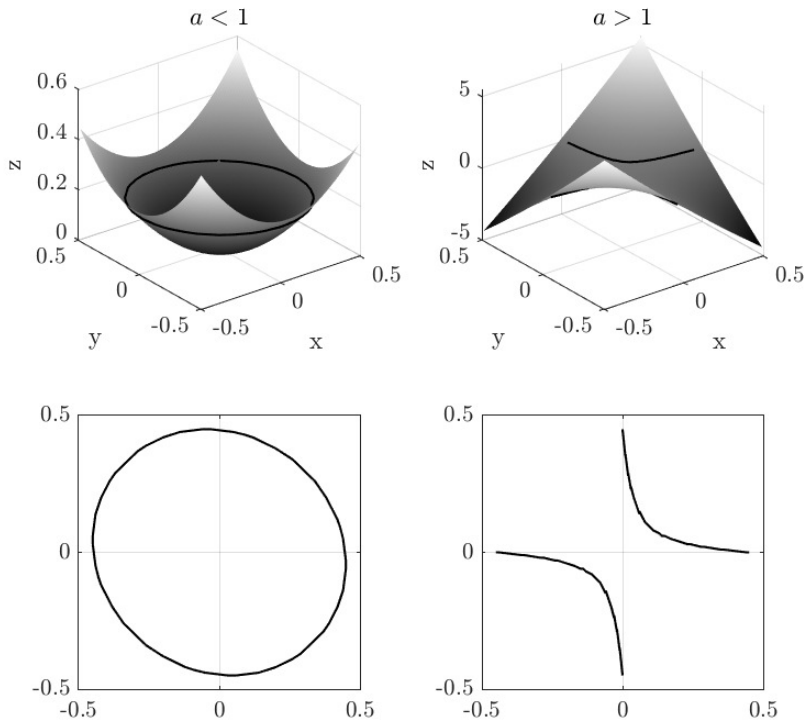


Figure 2.1: (Top) Plots of Equation (2.24) for two values of a. For $|a| < 1$ (left) the function is positive definite, for $|a| > 1$ (right) the result is hyperbolic. (Bottom) level curves of the plots taken at some $f(x, y) = c$.

which is then rearranged into the quadratic form

$$(u^n)^2 + (u^{n-1})^2 + \left(2k^2\frac{K}{M} - 2\right) u^n u^{n-1}. \tag{2.26}$$

Now let

$$a = k^2\frac{K}{M} - 1 \tag{2.27}$$

and insert into the inequality $|a| < 1$ to get

$$\left|k^2\frac{K}{M} - 1\right| < 1, \tag{2.28}$$

which implies that when

$$k < \frac{2}{K/M} \tag{2.29}$$

the system will be positive definite, the values of u^n and u^{n-1} will be bounded, and therefore stable. This is called the *stability condition* for the scheme and it allows one to choose parameters that ensure stability. For example, we usually have $k = 1/44100$, which mean that we have to choose K/M so

$$\frac{K}{M} < \frac{2}{k} = 2 \cdot 44100. \tag{2.30}$$

Since the frequency of oscillation of the mass-spring system is $f_0 = 2\pi(K/M)$, the above limits the maximum frequency to 44100 Hz$/\pi \approx 14037$ Hz.

Stability conditions for all the models presented in this document can be derived using energy techniques. However, showing stability conditions for distributed models, such as those presented later in this chapter, is slightly more involved since they need to be applied over the entire domain and account for boundary conditions; nonetheless, the technique is fundamentally the same.

2.2 Solving FDSs Using SymPy

As demonstrated in Equations (2.13) through (2.15) the FDS is solved for the next time step to derive and update rule. Solving the FDS of the simple system of the example was quite simple, but as one tackle more complex systems, the equations can become unwieldy to do by hand — a high number of terms will make the process error-prone, and implementation problems are hard to debug.

Therefore, I chose to experiment with solving FDSs using a *computer algebra system* (CAS), a software that allows one to do symbolic math with the aid of the computer.

The CAS used is SymPy [1] [19], a mature, open-source Python library. SymPy was extended with a custom library named SymFds, developed by me, that implements the handling of FDS state variables and finite difference operators. Basic use of SymFds, deriving the solution the example from section 2.1., looks like

```
from sympy import *
from symfds import *

omega0 = symbols('omega0')
u = StateSymbol('u')
u.isDistributed = False

fds = Eq(dtt(u), omega0 * u)
updateRule = solution(fds, nextt(u))

print(getCppExpression(updateRule))
```

which outputs

[1] https://www.sympy.org

```
(-1.0 * up) + (2.0 * u) + (omega0 * powf(k, 2.0) * u)
```

Note that this output is the same as Equation (2.15), but in a slightly less readable form. This is a general problem with CASs — they do not produce human-readable output; however, the expressions are correct.

SymFds can handle both distributed and point-like systems. It contains routines for solving FDSs (leveraging SymPy's built-in solvers), for optimizing expressions for real-time implementation, and for generating C++ code.

The optimizer eliminates constant terms from the given expression so they can be pre-computed prior to running the update loop for the given model. The optimizer also eliminates common terms by using SymPy's `cse` function, so the same computation is not performed multiple times.

SymPy integrates nicely with Jupyter notebooks, which offers a development environment capable of printing actual mathematical notation. All the systems of the following sections were solved using SymPy/SymFds and the notebooks used to do so can be viewed in Appendix A.

2.2.1 The pal-fds library

At the initial stages of this project, I was working on an FDS C++ library called pal-fds, in an attempt to create a completely modular physical modeling library that could implement multiple resonators and exciters from a common code base. The library got quite far along and worked great for simple models; however, for more advanced models, such as those of the following sections, a generalized implementation is hard to accomplish, so I opted to write specific implementations for each model, which motivated me to write SymFds.

2.3 Bowed and Fingered Stiff String

This section covers the model that is used to achieve the violin and cello-like sounds. The basis is a damped stiff string — a string whose restoring forces are the tension and the stiffness of the material. To excite the string, a (violin) bow is modeled, and a "finger" is modeled to give the model ability to play different pitches. First, the system is presented is continuous time and is then discretized.

2.3.1 Definition in Continuous Time

The model for a damped stiff string with external forces is defined by the differential equation [5]

$$u_{tt} = \frac{T_0}{\rho A} u_{xx} - \frac{EI}{\rho A} u_{xxxx} - 2\sigma_0 u_t + 2\sigma_1 u_{txx} + F \tag{2.31}$$

where $u = u(x,t)$ is the displacement of the string at point x, T_0 is the tension of the string, ρ is the material density, A is the cross-sectional area, E is the

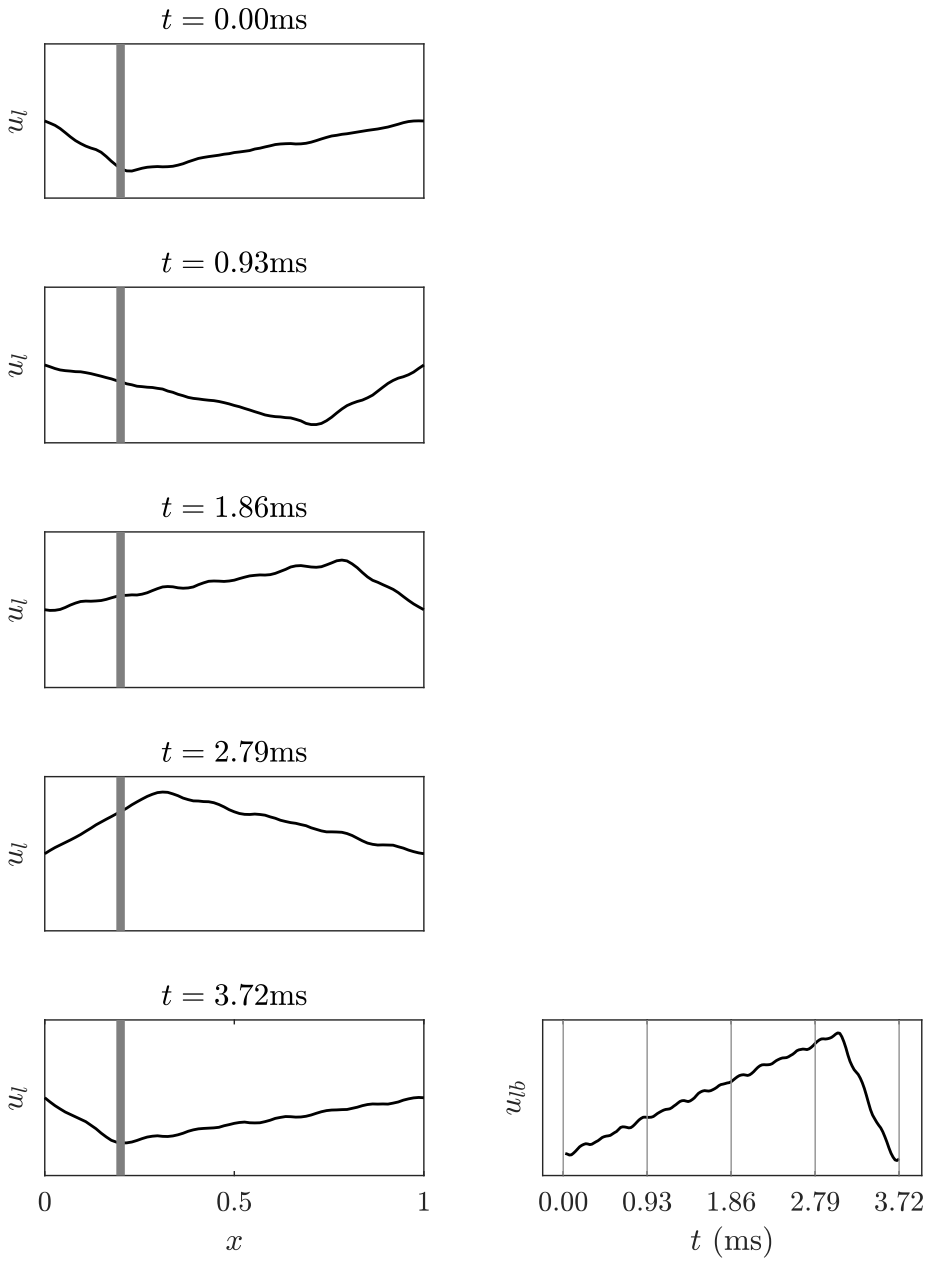


Figure 2.2: (Left) Plots of the bowed string's movement over time showed at five points of one cycle of Helmholtz-type motion. The first frame is where the stick part of the stick-slip process begins. The slip motion happens between frames four and five. (Right) The displacement of the string at the bowing point during the stick-slip cycle.

Young's modulus of the material, I is the moment of inertia, σ_0 and σ_1 are damping parameters, and F is a collection of external forces to be defined later.

It is often convenient to work with scaled versions of models [4], which is done by defining a new normalized coordinate $x' = x/L$, where L is the length of the domain, henceforth referred to as just x. To reduce the number of parameters one may collect redundant parameters into a few high-level parameters. When scaled, Equation (2.31) becomes

$$u_{tt} = \gamma^2 u_{xx} - \kappa^2 u_{xxxx} - 2\sigma_0 u_t + 2\sigma_1 u_{txx} + F \tag{2.32}$$

where

$$\gamma = \sqrt{\frac{T_0}{\rho A L^2}} \qquad\qquad \kappa = \sqrt{\frac{EI}{\rho A L^4}}. \tag{2.33}$$

The damped stiff string equation is arrived at by applying Newton's second law of motion to an infinitesimal section of the string [16]

$$\rho A u_{tt} = F. \tag{2.34}$$

Assuming the string is under tension, one may analyze the tension forces acting on the tiny section of the string and show that they are equivalent to $\gamma^2 u_{xx}$. Similarly, by analyzing the elastic forces within the string [10], occurring due to the material properties, one may arrive at the term $\kappa^2 u_{xxxx}$. The term $2\sigma_0 u_t$ models loss due to radiation, internal friction, and viscous drag of the air. The term $2\sigma_1 u_{txx}$ causes higher frequencies to die out faster than lower ones, a phenomenon observed in e.g., piano strings [3].

If the string is of finite length, one must define what happens at the boundaries — the left and right ends of the string. To mimic the bridge and nut of a real instrument, simply supported boundaries is a good choice, which amounts to letting [5, 6]

$$u = u_{xx} = 0, \tag{2.35}$$

at either end, when $x = 0$ or $x = 1$.

One may show, employing energetic techniques that, using simply supported boundaries[2], the energy of the stiff string model is monotonically decreasing [5].

The Bow

The bow works by a friction force between the string and the bow's hairs, defined by [5, chapter 4 and 7]

$$F_\beta = -\frac{1}{\rho A} F_\mathrm{b} \beta(v^n_\mathrm{rel}), \tag{2.36}$$

The function $\beta(v^n_\mathrm{rel})$ scales the friction force F_b depending on the relative velocity

$$v^n_\mathrm{rel} = u_t(x_\mathrm{b}) - v_\mathrm{b} \tag{2.37}$$

[2]Simply supported boundary conditions is not the only options. Free or clamped, and a multitude of other boundary conditions will also work

where v_b is the bow velocity and x_b is the bowing position. There are multiple ways of defining β, but

$$\beta(v_\mathrm{rel}^n) = \sqrt{2a} v_\mathrm{rel} e^{-a v_\mathrm{rel}^2 + 1/2}, \tag{2.38}$$

is useful because it is differentiable [5] — a requirement of the implementation, covered in section 2.3.2. A plot of β is shown in Figure 2.3

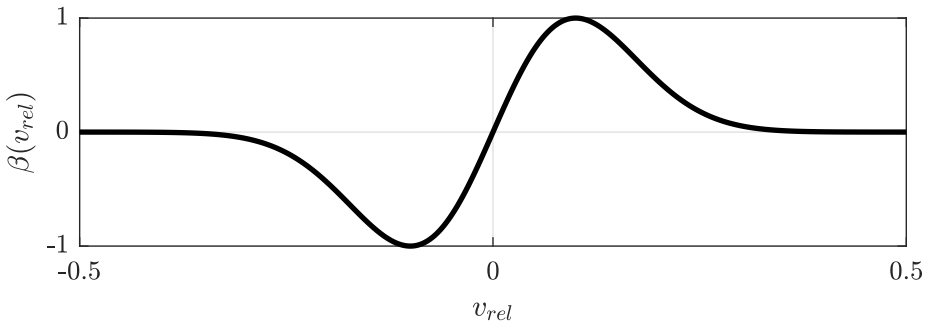


Figure 2.3: A plot of $\beta(v_\mathrm{rel}^n)$ with $a = 50$. In my experience, suitable values for a lie between 50 and 100.

The bow will cause the string to start oscillating by undergoing a stick-slip interaction [10], which is illustrated in Figure 2.2. At the beginning of the cycle, the friction force makes the string stick to the bow and causes it to move. The string moves with the bow until the restoring forces become too great, and the string slips back until it sticks to the bow again, and the cycle repeats. As a result, the string will undergo Helmholtz-type motion and produce a periodic output.

Using energetic techniques one may show that the energy of the bowed string system will be bounded by the energy introduced by the bow.

The Fingers

A crude way of providing pitch control for the string would be to simply change γ^2, however, the resulting pitch change sounds artificial and can cause problems with accuracy and stability when discretized. Instead, "fingers" can be modeled to effectively reduce the length of string allowed to vibrate, producing a more pleasing pitch change while also preserving the model's accuracy and stability.

The fingers are modeled by a damped spring-like connection to a rigid point, adapted from the prepared string systems found in [5, chapter 7] , defined by

$$F_\mathrm{f} = -E\left[\omega_\mathrm{f}^2 u(x_\mathrm{f}) + 2\sigma_\mathrm{f} u_t(x_\mathrm{f})\right] \tag{2.39}$$

where $E = E(t)$ is a time-dependent envelope function that allows for control of the finger. The parameter ω_f^2 is the stiffness of the connection, σ_f is the damping, and x_f is the point at which the finger is applied.

Note that, by the use of E, Equation (2.39) is not strictly physically correct. One may show that by holding E constant, the energy of the scheme is monotonically decreasing [5], however, if E increases quickly while $u(x_\mathrm{f}) \neq 0$, then additional potential energy has been introduced into the system at point x_f, and thus energy is not conserved. However, one can argue that fingering a string does introduce energy into the system, evidenced by e.g., the tapping technique on the guitar.

Alternatively, and more correctly, one could model the finger as a mass colliding with the string while also modeling a fingerboard. Bilbao and Torin did this in [7], where they also modeled the frets of a guitar. However, they solved their schemes using iterative methods, which is not ideal for a real-time solution. Another way of modeling the collision could be to use the state-of-the-art theory of [8]. I attempted at an early point of the project, but I was not able to get a satisfactory result.

Combining the Forces

The bow force F_β and finger force F_f are combined to form F like so

$$F = \delta(x - x_\mathrm{b})\frac{1}{\rho A}F_\beta + \delta(x - x_\mathrm{f})F_\mathrm{f} \tag{2.40}$$

where $\delta(x)$ is the Dirac delta function that has unity value when $x = 0$. Note that F_β is unscaled (multiplied $1/\rho A$), which allows us to specify the bowing force F_b in Newtons, so it can be matched to the force used when playing a real violin. For the finger, F_f is left scaled because the finger parameters ω_f^2 and σ_f do not need to be chosen based on any real-world forces.

2.3.2 Finite Difference Scheme and Implementation

To implement Equation (2.32) the following FDS can be used [5]

$$\delta_{tt}u_l^n = \gamma^2\delta_{xx}u_l^n + \kappa^2\delta_{xx}\delta_{xx}u_l^n - 2\sigma_0\delta_{t\cdot}u_l^n + 2\sigma_1\delta_{t-}\delta_{xx}u_l^n + f_l^n. \tag{2.41}$$

The discretized external forces f_l^n are now defined as

$$f_l^n = J(x_\mathrm{b})\frac{1}{\rho A}f_\beta^n + J(x_\mathrm{f})f_\mathrm{f}^n. \tag{2.42}$$

where $J(x)$ is some spreading operator, somewhat a discrete version of the Dirac delta function, that scales whatever it modifies by $1/h$ and distributes it to grid points close to x. See appendix B.1 for a more in depth explanation of spreading and interpolation operators.

After the external forces have been computed, Equation (2.41) can be solved for u_l^{n+1} which gives us our update rule. Using discrete energetic methods (or von Neumann analysis) one may arrive at the stability condition [25]

$$h \geq h_\mathrm{min} = \sqrt{\frac{\gamma^2k^2 + 4\sigma_1k + \sqrt{(\gamma^2k^2 + 4\sigma_1k)^2 + 16\kappa^2k^2}}{2}}. \tag{2.43}$$

After choosing the sample rate, wave speed, stiffness, and damping coefficients, the stability condition is used find $h_{\min}$, then, by choosing $N = \text{floor}(1/h_{\min})$ one has $h = 1/N$.

The simply supported boundary conditions are implemented by solving the discrete approximation

$$u_l^n = \delta_{xx} u_l^n = 0 \tag{2.44}$$

at either end of the string, $l = 0$ and $l = N - 1$. At the left side, using δ_{xx}, the scheme depends on u_{-1}^n which lies outside the string. This is known as a *virtual* point, whose value can by solving Equation (2.44), which gives $u_{-1}^n = -u_1^n$. The same can be done for the right side.

The Bow

For the bow, Equation (2.37) can be discretized like [5, chapter 4 and 7]

$$v_{\text{rel}}^n = \delta_{t\cdot} I(x_{\text{b}}) u^n - v_{\text{b}}, \tag{2.45}$$

where $I(x_{\text{b}})$ is some interpolation operator of the same order as the spreading operator used to apply f_β^n.

Using $\delta_{t\cdot} u_l^n$ makes the scheme *implicit*, which means iterative methods, such as Newton-Raphson, must be used to solve it. If $\delta_{t-} u_l^n$ had been used instead, the scheme would have been explicit. However, it would be hard to make a statement about the stability of the system [5, chapter 3]. Also, in my experience, the scheme using the centered difference sounds much better, especially if the system close to the stability condition.

Given a possibly non-linear function $g(x)$, the Newton-Raphson method, allows one to numerically find an x so that $g(x) = 0$ [23]. The method involves iteratively performing

$$x \leftarrow x - \frac{g(x)}{g_x(x)}, \tag{2.46}$$

where each iteration will (ideally) yield a better approximation of the x for which $g(x) = 0$, though there is no guarantee that the algorithm will converge.

To compute v_{rel}^n, solve (2.45) and (2.41) for u_l^{n+1}, set them equal to one another, and put them into a form like $g(v_{\text{rel}}^n) = 0$. After taking the derivative of g with respect to v_{rel}^n, corresponding to $g_x(x)$, run the Newton-Raphson step until the change in v_{rel}^n is sufficiently small or a set maximum of iterations have been reached. The implementation is simplified by assuming that $x_{\text{b}} \neq x_{\text{f}}$. After v_{rel}^n is known the discrete bow force f_β^n may be computed directly.

Using the centered difference operator and matching the order of the interpolation and spreading operator, one may show that the bow does not influence the stability of the string and that the energy will be bounded by that supplied by the bow [4, chapter 7].

The Finger

The finger, Equation (2.39), is discretized as [5, chapter 7]

$$f_{\text{f}} = -E \left[\omega_{\text{f}}^2 I(x_{\text{f}}) \mu_{t\cdot} u_l^n + 2\sigma_{\text{f}} I(x_{\text{f}}) \delta_{t\cdot} u_l^n \right]. \tag{2.47}$$

A simpler scheme would omit the $\mu_{t\cdot}$, but one can show that such a scheme would result in a tighter stability condition [5, chapter 7]. The scheme in (2.47) has no impact on the stability of the string overall and can, despite its dependence on $I(x_\mathrm{f})u_l^{n+1}$, be evaluated without the use of iterative methods (a semi-implicit solution).

To do so, apply an interpolation operator to (2.41), including f_f^n, and solve for $I(x_\mathrm{f})u_l^{n+1}$. Once $I(x_\mathrm{f})u_l^{n+1}$ is known f_f^n can be calculated directly using Equation (2.47).

Implementation Notes

The parameters of the string,γ, κ, σ_0, and σ_1 are kept constant. γ is chosen based on the lowest desired pitch, and κ is computed using the material properties E, I, ρ, and A.

The phrasing of the music, the way in which notes are played and later released, controls the parameters of the bow. When a phrase begins, an ADSR envelope increases F_b and v_b, and at the end of the phrase, the ADSR's are released, setting them back to zero. The bow force is further modulated by several factors described in Chapter 3.

The desired pitch controls the positions and envelopes of the fingers. Two fingers are used in turn: when we wish to play a new pitch, a finger is moved to the corresponding position, and the envelope is triggered while the envelope of the other finger is released. The position of a finger can be computed by

$$x_\mathrm{f} = \frac{f_0}{\tau f} \tag{2.48}$$

where f is the desired frequency and

$$\tau = \tau_0^{n/12} \tag{2.49}$$

is a tracking parameter that corrects for inaccuracies in the implementation where n is the number of semi-tones between f_0 and f. Suitable values for τ_0 were found by ear.

2.4 Hammered Bar of Variable Cross Section

This section covers the model used to achieve the struck pitched percussion sound. The sounding objects of such instruments, such as marimbas, glockenspiels, and vibraphones, are called bars — blocks of some, usually very stiff, material suspended by foam pads or wires. Both the bars and the suspension will be modeled. The bars are traditionally struck with mallets — wooden sticks with heads made of materials of varying hardness. Mallets are modeled in a way that allows controls of strike speed and mallet hardness and weight.

2.4.1 Definition in Continuous Time

The vibrations of a thin bar with damping are described by [5, chapter 7] [4, 6]

$$u_{tt} = -\kappa^2 u_{xxxx} - 2\sigma_0 u_t + 2\sigma_1 u_{txx} + F, \tag{2.50}$$

which is identical to Equation (2.32) but without the tension term $\gamma^2 u_{xx}$ — the only restoring force of the bar is the stiffness of the material.

Like real bars, Equation (2.50) exhibits inharmonic overtones. However, inharmonicity is undesirable for musical purposes, so real bars have material removed at specific places to tune their overtones to harmonic ratios. The same can be done virtually by modeling a bar of variable cross section, which is defined by [5, chapter 7]

$$\theta u_{tt} = -\kappa_0^2(\theta^3 u_{xx})_{xx} - 2\sigma_0 u_t + 2\sigma_1 u_{txx} + F, \tag{2.51}$$

where $\theta = \theta(x)$ is a function representing the variation in material thickness and κ_0 is a reference stiffness (the stiffness of the bar where $\theta = 1$).

One may show that the bar is energy conserving under the conditions that θ is constant near the boundaries, and the boundary condition introduced in the next section is used[3][4, chapter 7].

The Suspensions

The bars of marimbas and vibraphones are suspended by strings going through the bar, allowing their ends to move freely. Using the free boundary condition

$$u_{xx} = u_{xxx} = 0, \tag{2.52}$$

the same is allowed for the bar model. The suspensions keeping the bar in place are modeled by two damped spring-like connections, defined by [5, chapter 7]

$$F_\mathrm{s} = -\omega_\mathrm{s}^2 u(x_\mathrm{s}) - 2\sigma_0 u_t(x_\mathrm{s}). \tag{2.53}$$

The Mallet

A model of a collision with a mallet is used to excite the string. A common choice for modeling the collision force is [5, chapter 4 and 7][8]

$$F_\mathrm{m} = K_\mathrm{m}([\eta]^+)^q \tag{2.54}$$

where

$$\eta = u_\mathrm{m} - u(x_\mathrm{m}), \tag{2.55}$$

where u_m is the position of the hammer, K_m is the hammer stiffness, q is a nonlinear compression constant, x_m is the impact position, and $[\eta]^+ = (\eta + |\eta|)/2$ is the positive part of η. Note that using this definition of η, the mallet should hit the bar from below, which is a convention adopted from [5] and [8]; by

[3] Again, other boundary condition can also be energy conserving.

inverting η and the consequent forces the mallet could made to hit from above instead.

The movement of the mallet is determined by

$$(u_{\mathrm{m}})_{tt} = -\frac{F_{\mathrm{m}}}{M_{\mathrm{m}}}, \tag{2.56}$$

where M_{m} is the mass of the mallet. One may show that the mallet/bar system is energy preserving [5, chapter 7].

A plot of a mallet striking the bar is seen in Figure 2.4. While the collision model is quite good, the overall mallet does not accurately model what is happening when a percussion player strikes their instrument. The model counts on the collision to make the mallet retract, a technique used when playing drums, however, when playing pitched percussion instruments, the player usually actively retracts the mallet.

Combining The Forces

The external forces F in Equation (2.51) are determined by

$$F = \delta(x - x_{\mathrm{m}})\frac{F_{\mathrm{m}}}{\rho A_0 \theta(x_{\mathrm{m}})} + \delta(x - x_{\mathrm{s1}})F_{\mathrm{s1}} + \delta(x - x_{\mathrm{s2}})F_{\mathrm{s2}} \tag{2.57}$$

where x_{m} is the hammer impact position, ρA_0 is the reference linear density of the bar, and x_{s1} and x_{s2} are the positions of the supports. Like the bow model, the hammer is left unscaled because it allows us to tweak the parameters using physical references.

2.4.2 Finite Difference Scheme and Implementation

The obvious way to discretize Equation (2.51) would be [5, chapter 7]

$$[\theta]\, u_l^n = -\kappa_0^2 \delta_{xx}(\theta^3 \delta_{xx} u_l^n) - 2\sigma_0 \delta_{t\cdot} u_l^n + 2\sigma_1 \delta_{t-} \delta_{xx} u_l^n + f_l^n, \tag{2.58}$$

where $[\theta]$ is some second order approximation of θ, e.g. $\mu_{xx}\theta$. The definition of the discrete forces f_l^n is postponed until the Implementation notes subsection later in this section.

Alas, the scheme in Equation (2.58) does not perform well in general. The variations in stiffness and constant grid spacing make the scheme less accurate at points of low stiffness, causing a loss of high-frequency overtones and increased numerical dispersion[4].

To avoid such problems, the bar can instead be discretized with variable grid spacing by doing a change of variable for x [5, chapter 5]. This is done by introducing the new coordinate $\alpha = \alpha(x)$ under which the spatial derivatives become

$$\frac{\partial}{\partial x} \quad \rightarrow \quad \alpha_x \frac{\partial}{\partial \alpha} \tag{2.59}$$

$$\frac{\partial^2}{\partial x^2} \quad \rightarrow \quad \alpha_x \frac{\partial}{\partial \alpha}\left(\alpha_x \frac{\partial}{\partial \alpha}\right). \tag{2.60}$$

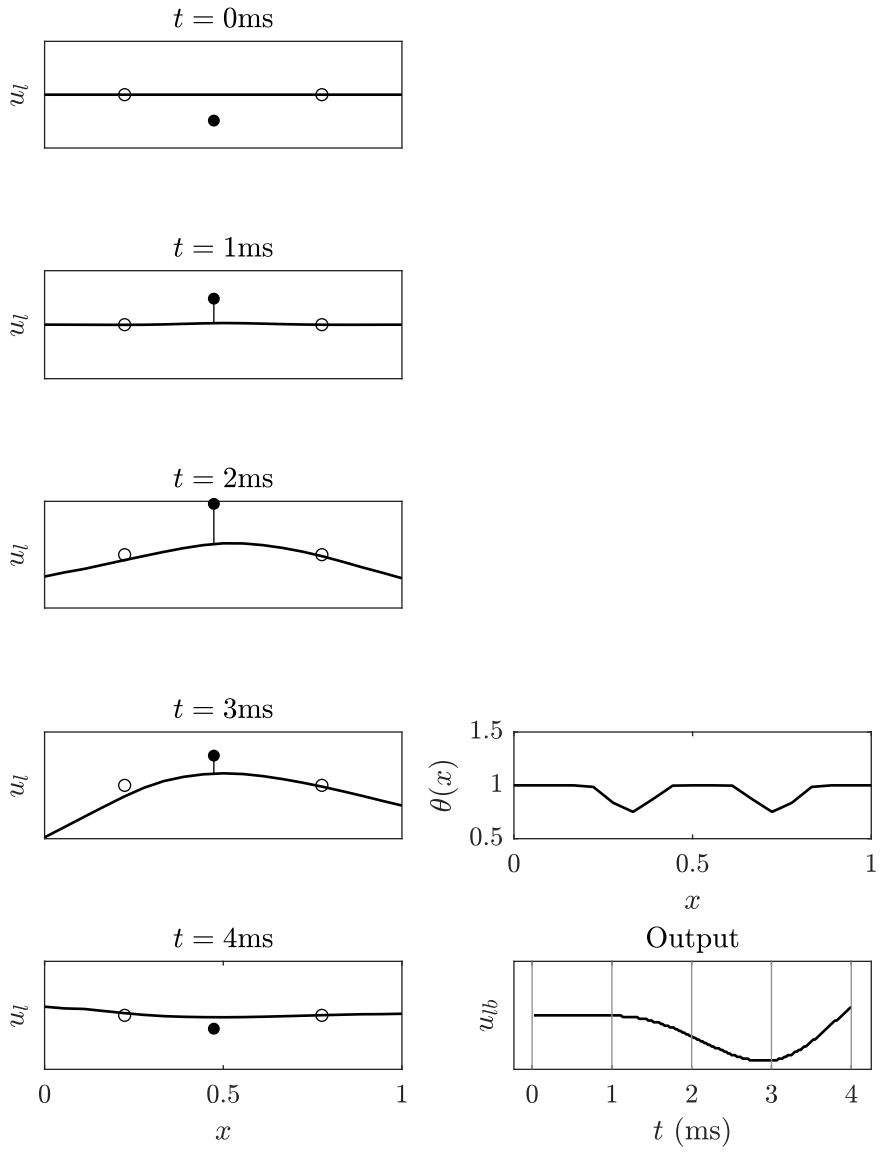


Figure 2.4: (Left) A plot of a mallet hitting a bar shown at five instances just before and just after impact. The mallet is represented by the solid black dot and the suspensions by the hollow dots. (Right, top) The stiffness variation $\theta(x)$ of the bar. (Right, bottom) the displacement of the bar at point $x = 0.17$ during the impact.

Finite difference operators transform similarly, e.g. the second order spatial difference operator becomes

$$\delta_{xx} \quad \rightarrow \quad \alpha_x \delta_{\alpha+}((\mu_{\alpha-}\alpha_x)\delta_{\alpha-}). \tag{2.61}$$

In case of the bar model a suitable choice for α is [5, chapter 7]

$$\alpha(x) = \frac{1}{\alpha_{\text{av}}} \int_0^x \frac{1}{\sqrt{\theta(\eta)}} d\eta, \tag{2.62}$$

where α_{av} is a constant defined by

$$\alpha_{\text{av}} = \int_0^1 \frac{1}{\sqrt{\theta(\eta)}} d\eta. \tag{2.63}$$

A plot of an example θ and the corresponding α can be seen in Figure 2.5.

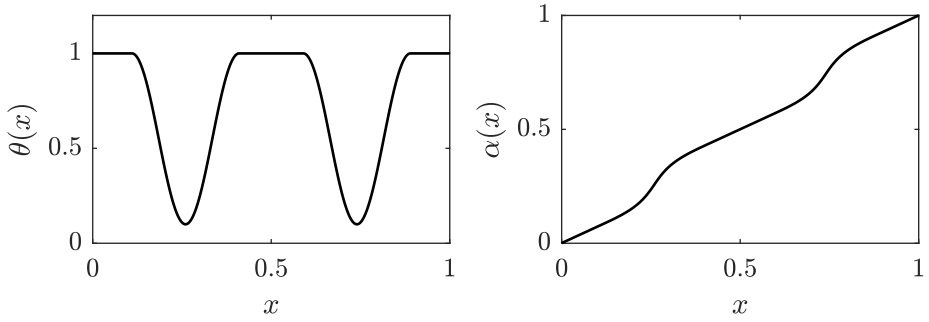


Figure 2.5: A plot of a θ (bar cross section) function similar to the one actually implemented and the corresponding α.

Under this transformation Equation (2.51) becomes

$$\begin{aligned} \theta^{3/2} u_{tt} = &- \frac{\kappa_0^2}{\alpha_{\text{av}}^4} \left(\theta^{-1/2} \left(\theta^{5/2} \left(\theta^{-1/2} u_\alpha \right)_\alpha \right)_\alpha \right)_\alpha \\ &- 2\sigma_0 u_t + 2\sigma_1 \frac{1}{\alpha_{\text{av}}^2} \left(\theta^{-1/2} \left(\theta^{-1/2} u_\alpha \right)_\alpha \right)_t \\ &+ F. \end{aligned} \tag{2.64}$$

How the transformation is applied is covered in detail in appendix B.2. A suitable scheme for this is

$$\begin{aligned} \left[\theta^{3/2}\right] \delta_{tt} u_l^n = &- \frac{\kappa_0^2}{\alpha_{\text{av}}^4} \delta_{\alpha+} \left(\mu_{\alpha-} \theta^{-1/2} \delta_{\alpha-} \left(\theta^{5/2} \delta_{\alpha+} \left(\mu_{\alpha-} \theta^{-1/2} \delta_{\alpha-} u_l^n \right) \right) \right) \\ &- 2\sigma_0 \delta_{t\cdot} u_l^n + \frac{2\sigma_1}{\alpha_{\text{av}}^2} \delta_{t-} \delta_{\alpha+} \left(\theta^{-1/2} \left(\mu_{\alpha-} \theta^{-1/2} \delta_{\alpha-} u_l^n \right) \right) \\ &+ f_l^n, \end{aligned} \tag{2.65}$$

which can be solved explicitly for u_l^{n+1}.

This scheme performs much better than the one in Equation (2.58) with regards to bandwidth. Energy analysis of the scheme without the damping terms will reveal the stability condition

$$h \geq h_{\min} = \frac{2\kappa k}{\alpha_{\text{av}}^2}. \tag{2.66}$$

when $\left[\theta^{3/2}\right]$ is chosen as

$$\left[\theta^{3/2}\right] = \mu_{x+}\left((\mu_{x-}\theta^{5/2})\mu_{x-}(\theta^{-1/2})^2\right). \tag{2.67}$$

This stability condition does not account for the frequency dependent damping, but by being conservative in the selection of h, e.g. $h = 1.05h_{\min}$, and avoiding too high values of σ_1, stability is not a problem.

The Suspensions

Free boundary conditions can be achieved by solving

$$\delta_{xx}u_0^n = \delta_{x-}\delta_{xx}u_0^n = 0 \tag{2.68}$$

for u_{-1}^n and u_{-2}^n, and

$$\delta_{xx}u_{N-1}^n = \delta_{x+}\delta_{xx}u_{N-1}^n = 0 \tag{2.69}$$

for u_N^n and u_{N+1}^n, similarly to how it was done for the stiff string.

The supports, defined in Equation (2.53) are implemented using the simple scheme

$$f_{\text{s}} = -\omega_{\text{s}}^2 I(x_{\text{s}})u_l^n - 2\sigma_0 I(x_{\text{s}})\delta_{t-}u_l^n. \tag{2.70}$$

Because the energy contribution of this scheme is not necessarily non-negative, this scheme interferes with the stability of the bar, which means that a stronger stability condition must be enforced, see [5, section 7.7.1]. However, by choosing h conservatively, and if ω_{s} is not too large, stability is not a problem. Note that δ_{t-} is used instead of the more accurate $\delta_{t\cdot}$ to simplify the implementation of the scheme.

The Mallet

The obvious way to implement the mallet is to compute the discrete mallet impact force f_{m}^n directly with $\eta^n = u_{\text{h}}^n - u_{l_{\text{m}}}^n$ where $l_{\text{m}} = \text{floor}(x_{\text{m}}/h)$, and then update the mallet position using the scheme

$$\delta_{tt}u_{\text{m}}^n = -\frac{f_{\text{m}}^n}{M_{\text{m}}}. \tag{2.71}$$

However, such a solution has a tendency to numerical oscillation, which is undesirable [5, chapter 5 and 7]. A better way would be to compute f_{m}^n by

$$f_{\mathrm{m}}^n = K_{\mathrm{m}}([\eta^n]^+)^{q-1}\mu_{t\cdot}\eta^n \tag{2.72}$$

which can be solved semi-implicitly. Unlike the schemes presented until now, this scheme cannot be shown to be energy conserving. However it can be shown to be almost conservative when $q = 1$ or $q = 3$ [5, chapter 4 and 7].

To get a generally energy conserving solution one might now try iterative methods, however, new developments in non-linear modeling provides a way to implement our hammer in an energy-conserving manner, without the use of iterative solver [8, 13]. The system does, however, need to be stated differently, as will be shown now.

First, rewrite F_{m} as

$$F_{\mathrm{m}} = \phi_\eta \tag{2.73}$$

where $\eta = u_{\mathrm{m}} - u(x_{\mathrm{h}})$, and

$$\phi_\eta = \frac{d}{d\eta}\phi = K_{\mathrm{m}}([\eta]^+)^q. \tag{2.74}$$

which means that

$$\phi = \frac{K}{q+1}([\eta]^+)^{q+1} \tag{2.75}$$

Equation (2.73) can be rewritten as

$$F_{\mathrm{m}} = \psi\frac{\psi_t}{\eta_t} \tag{2.76}$$

where $\psi = \sqrt{2\phi}$, see Appendix B.3 for more details. This can be discretized by

$$f_{\mathrm{m}}^n = (\mu_{t+}\psi^{n-1/2})g^n \tag{2.77}$$

where

$$\psi^{n-1/2} = \mu_{t-}\psi_l^n \tag{2.78}$$

and

$$g^n = \frac{\delta_{t+}\psi^{n-1/2}}{\delta_{t\cdot}\eta^n}, \tag{2.79}$$

which may be computed directly using

$$g^n = \psi_\eta|_{\eta=\eta^n} = \left.\frac{\phi_\eta}{\sqrt{2\phi}}\right|_{\eta=\eta^n}. \tag{2.80}$$

Using the identity

$$\mu_{t+}\psi^{n-1/2} = \frac{k}{2}\delta_{t+}\psi^{n-1/2} - \psi^{n-1/2} \tag{2.81}$$

and (2.79) one may rewrite (2.77) as

$$f_{\mathrm{m}}^n = \frac{k(g^n)^2}{2}\delta_{t\cdot}\eta^n - g^n\psi^{n-1/2}. \tag{2.82}$$

Assuming that the hammer impact point does not coincide with any of the supports, $f_{l_{\mathrm{m}}}^n$, the force at the mallet impact point, is

$$f_{l_{\mathrm{m}}}^n = \frac{f_{\mathrm{m}}}{\rho_0\theta_{l_{\mathrm{m}}}}. \tag{2.83}$$

Pairing Equation (2.65) at point l_{h} with (2.71) one may solve for $u_{l_{\mathrm{h}}}^{n+1}$ and u_{m}^{n+1} and thus, because the zeroth order interpolation/spreading operators are used, update both the hammer and bar position.

One may show using energy analysis that if ϕ is non-negative then the energy of the system will also be non-negative and thus stable under the stability condition introduced earlier [8].

Implementation Notes

The mallet scheme takes care of updating the bar at point l_{m}, scheme (2.65) takes care of everything else. Therefore, f_{m} needs not be included in the discrete definition of f_l^n, which is

$$f_l^n = J(x_{\mathrm{s}1})f_{\mathrm{s}1} + J(x_{\mathrm{s}2})f_{\mathrm{s}2}. \tag{2.84}$$

To tune the bar I followed the process suggested in [14] for tuning real bar instruments. This entailed making the cross section thinner at points corresponding to the anti-nodes of the second mode using movable Hann functions defined by

$$\nu(x) = \begin{cases} d\sin(\pi(x - w/2 - p)/w)^2 & p - w/2 \le x \le p + w/2 \\ 0 & \text{otherwise} \end{cases} \tag{2.85}$$

where d is the depth of the notch, w is the width, and p is the center position. Using two such functions, so that $\theta = \nu_1 + \nu_2$, overtones were adjusted by ear until they sounded pleasing. The profile used can be seen in Figure 2.4. right, top. Afterwards, κ was set to control the overall pitch.

Whereas bowed string instruments usually play multiple pitches on one string, bar instruments have one bar for each note. This is implemented by creating a new bar with the desired pitch (stiffness) for each played note. Finding the correct value for κ, though roughly linear with respect to frequency, proved difficult, and each note of the Ionian scale from C4 to C6, the desired range, had to be tuned by hand. Linear interpolation is used to determine the stiffness of notes outside the scale.

Figure 2.6: A screenshot of the developer GUI for the implementation, with a window for each voice and one for general parameters.

2.5 Software Design

Two Jupyter notebooks, one for each instrument, were created to solve the models' FDSs and generate C++ code for computing the solution. The C++ was then pasted by hand into custom-written C++ classes.

PAL (the pretentious audio library) was used for audio IO and a developer GUI, a screenshot of which can be seen in Figure 2.6. For UI, PAL uses Dear ImGui[4], an immediate-mode UI library that makes it fast and easy to modify the GUI for debugging purposes. An invaluable feature to have while troubleshooting.

The application is structured in a hierarchy to separate concerns. First are the instrument model implementations, which implement the FDSs and utility functions related to those. Second are voice classes that take care of the higher level, musical control of the instrument models, including note sequencing, the KTH performance rules (see Chapter 3), tuning, and most other mappings. The third and final layer is the main application loop, which presents the UI, calls the audio rendering methods of the lower layers, and gets input from the user.

[4] `https://github.com/ocornut/imgui`

Chapter 3

Composition and Interactivity

The act of composition involves using one's æsthetic sensibilities to choose from an infinite pool of possibilities what musical elements should be part of the piece. During the composition process, the composer might delegate some amount of choice to an external process. For example, in the medieval times, Guido of Aresso devised a methodical way of assigning notes to syllables [17, chapter 9]. In the Renaissance and Baroque eras, strict rules were developed for harmony and voice leading [1, chapter 5 and 6]. In the classical period, so-called würfelspiel pieces were developed, some attributed to Mozart and Haydn [17, chapter 9][21], which allowed one to compose music by rolling dice. The second Viennese school, Schoenberg, Webern, and others, developed the method of 12-tone serialism, later extended Messiaen, Boulez, and Babbitt[22, chapter 11]. John Cage used the randomness of the Chinese divination method I Ching as a basis for some of his compositions [22, chapter 11], and Iannis Xenakis used the stochastic rules governing the movement of gasses [26]. In his piece *Klaverstück XI*, Stockhausen distributed 19 musical fragments on a page. During performance, any fragment can follow any other as chosen by the performer; choice is delegated to the performer. In Terry Riley's piece, *In C*, 53 short phrases are played at a pace chosen by the performers, multiple performers will progress at different speeds resulting in a complex musical structure.

An excellent tool to which the composer may delegate choice is the computer, and much work has been done in the realm of computer driven composition. An early example is the *Illac Suite* by Hiller and Isaacson, a string quartet composed using Monte Carlo methods [20]. Since then, a staggering amount of techniques has been used, including grammars, constraint satisfaction techniques, Markov chains, cellular automata, evolutionary methods, and machine learning, see e.g., [20] or [9] for a comprehensive review. In recent years it seems that machine learning methods have been getting much attention e.g., the Coconet by Google

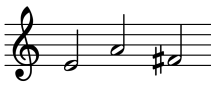
Magenta[1], which they trained to harmonize melodies in the style of Bach [15].

In the music composed for this project choice is delegated to both the computer and the user while some is retained for the composer (myself): rhythm and some amount of the pitch material is predetermined.

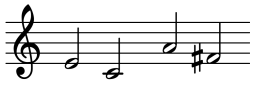
The rhythmic material consists of a collection of *split-note rhythms*. These are created by starting with a rhythmic sequence and splitting a note in two, e.g., one 1/4 note becomes two 1/8 notes. For each note we split we get a new rhythmical pattern, which we can then use to generate more sequences. I call this split-note rhythms. Starting with a whole-note and successively running the algorithm on the previous output, we get 1, 2, 3, 6, 24, and 120 rhythms at each level. This is visualized, from levels one to six, in Figure 3.1.

The pitches to be played over the rhythms are derived from the C octatonic, half-whole scale. A tone row technique is used to create melodies, which involves arranging the notes of a scale in a sequence where no note may be repeated before all other notes have been played. A row may be retrograded (reversed), inverted (intervals inverted), or both. This is the basis of the twelve-tone technique often associated with Schoenberg and the second Viennese school, who used tone-row techniques on the chromatic scale. Tone rows can, however, be constructed from other scales, such as the octatonic scale.

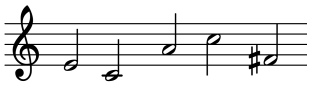
The tone row to be played will depend on the rhythmic level. At levels two and below the following tone row is used



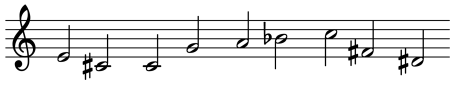
The tone rows for level three and four are elaborations on this and are respectively



and



At level five and above the tone row encompasses all the notes of the scale



All rows may be retrograded, but not inverted.

Additional rhythmical interest is obtained by converting some notes to rests. Let p_r be the chance that a note will be converted to a rest. Modulating this value will yield varying degrees of density in the music.

[1] `https://magenta.tensorflow.org/`

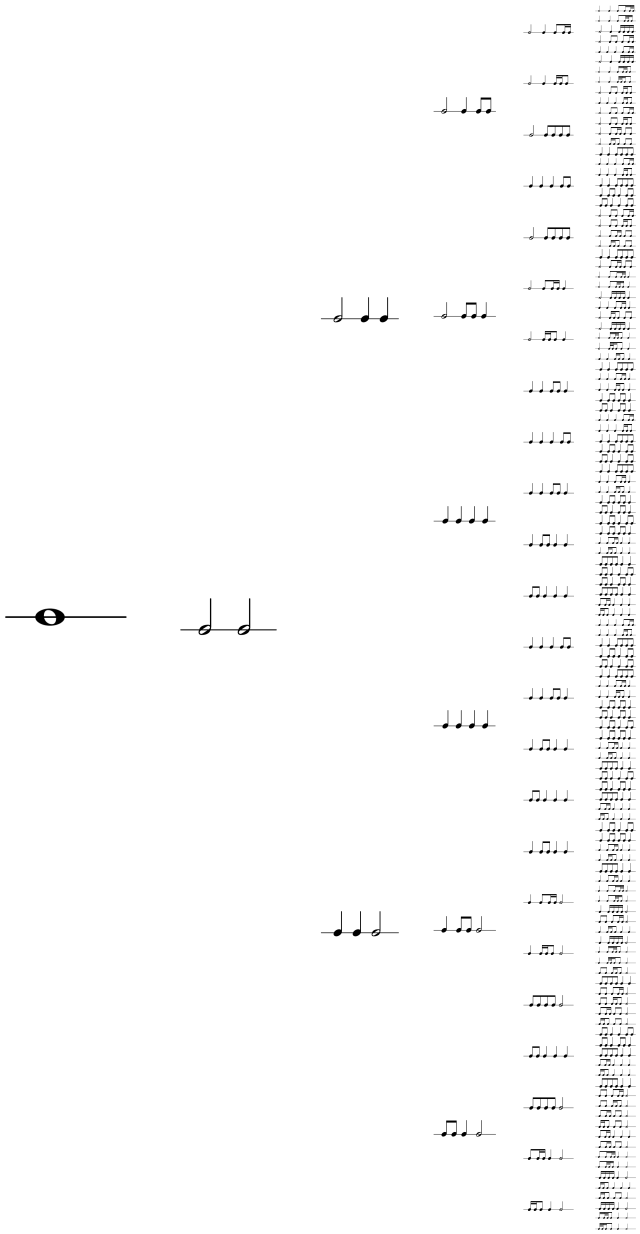


Figure 3.1: Figure showing how split-note rhythms are generated. Starting on the left with the whole note, all notes are split to generate a new rhythm for each note and the process is applied again.

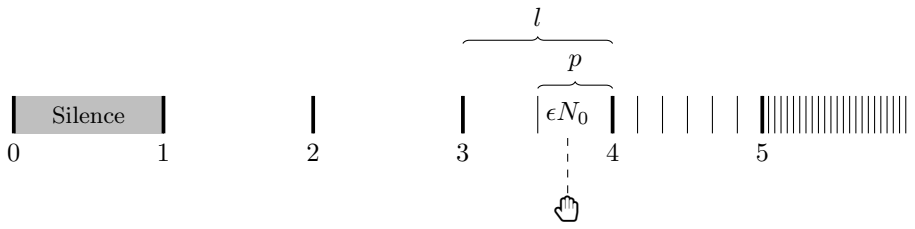


Figure 3.2: Illustration of how the user's hand position is mapped to a rhythmical pattern. In this case $N_\mathrm{l} = 6$. The player hand elevation ϵN_l is around 3.7, so the second pattern within level three is selected.

3.1 Mapping and Interactivity

The user controls the composition in real-time using an Ultraleap Leap Motion, a device that tracks the position of the user's hands, arms, and fingers using near-infrared cameras and computer vision.

The elevation of the user's hands selects which rhythm to play. The range of the sensor is divided into six regions (zero to five). While in region zero, no music is played. In regions one to five, rhythms from the corresponding rhythm level is played. Patterns within a level are sorted in ascending order according to their intensity, which I have defined as

$$\iota = \sum_{n \in \mathcal{N}} -\log_2(d_n) \tag{3.1}$$

where $\mathcal{N}$ is the set of notes in the given pattern and d_n is the duration of note n. The elevation of the user's hand ϵ, which normalized from the range (100 mm, 500 mm) to $(0, 1)$, is used to select the rhythm level and the pattern within that level. The level is determined by

$$l = \mathrm{floor}(\epsilon N_\mathrm{l}) \tag{3.2}$$

where N_l is the total number of levels. The pattern within the level is selected by

$$p = \mathrm{round}(N_\mathrm{p}(\epsilon N_\mathrm{l} - l)) \tag{3.3}$$

where N_p is the number of rhythms within the level and p is the index of the rhythm to play. How the user selects rhythms is illustrated in Figure 3.2.

The user's left hand controls the violin voice, and the right hand controls the cello voice. The percussion voice is controlled by taking the average of the two hands and mapping it to the range $(0, 7)$, causing the percussion to play rhythms of higher complexity than the bowed instruments.

When the user's hand is open, the p_r of the corresponding voice is zero. When the user closes their hand to a fist, p_r is set to 0.5. Thus, the number of rests, and therefore the density, can be controlled for each voice. p_r for the percussion voice is the average of the two others with 0.1 added.

The elevation of the user's hand is also mapped to the bow force F_b, output volume, and mallet velocity. Most mappings use the function[2]

$$f(\epsilon, M) = M\left(\frac{e^{s\epsilon} - 1}{e^s - 1}\right) \tag{3.4}$$

which will map ϵ to the range $(0, M)$, where s is a free parameter controlling the shape of the mapping adjusted for each case. For $s > 0$, the above say $s = 2$ or $s = 3$ will yield an exponential curve. For $s < 0$ it will be logarithmic. When s is close to zero, the curve is almost linear. $s = 0$ is not permitted as the result is undefined.

3.2 Musicality

In traditional composition, the composer counts on the performers to add expression to the music — a feat performers provide by varying the rhythm, dynamics, and intonation of the piece. The same cannot be expected of computers which, if not instructed otherwise, will play back the material regularly and monotonically.

To obtain better results from computer performers, one may implement a set of performance rules such as those of the KTH rule system, which instructs how notes are to be played back based on a collection of heuristic functions [11][12].

Most of the KTH rules have to do with rhythm: shortening or elongating notes to add expression. However, due to the difficulty of implementing such rules for ensemble music, though Friberg does present a way of doing so [11], only rules concerning amplitude were implemented. The following rules from [11] are included:

DPC 1B. High Loud Higher pitched notes are played louder.

DDC 1. Durational Contrast Notes with duration 30ms to 600ms are decreased in loudness according to a piecewise function.

DPC 2A. Melodic Charge Notes of the chromatic scale are given weights based on their distance to the root in the circle of fifths. Further away notes are played louder.

To impart higher-level musical structure, and to enhance the sense of departure and arrival, an overall phrase amplitude function was also added, which somewhat relates to the KTH rule concept of a phrase arc. This makes it so notes towards the middle of a phrase are played louder. Phrase lengths are chosen at random and can be either 4, 8, or 12 measures.

[2] I came up with this function myself, but I would not be surprised to find it somewhere in the literature.

When playing at rhythmical level five, a bowed voice may switch to playing notes staccato — notes longer than 1/16 will be converted to a 1/16 note plus a rest.

All notes played by the bowed instruments are subject to a slight vibrato controlled by an ADSR envelope, that is, the finger positions are modulated by a slow sine wave, whose amplitude is governed by the ADSR.

3.3 Munola

During development, there was a need for a way of inputting music that the virtual instruments could play in order to experiment with different musical material. The composition I wanted to do required symbolic manipulation of the music, which the library implementing the notation would have to support.

One option was MIDI, but MIDI uses a binary file format that is not editable without the use of dedicated software. Also, the way MIDI represents music is not suited for the style of symbolic manipulation I wanted to do.

A textual format would be more suitable, as it would be editable using any text editor. One such system is LilyPond[3], and open source music notation language and software package. However, LilyPond is a heavy-weight package intended for full score notation and rendering and does not seem designed for integration into other projects nor symbolic manipulation. A similar project is ABC notation[4], which would be a great choice except for a few kinks in the language and the lack of a mature C++ implementation (that I know of).

To overcome the problems with existing solutions, and to enable design specific to the needs of this project, a custom notation language *Munola* (Music notation language) was created. The goal was for the language to be easy to read and write for both humans and computers, and to provide a C++ library for easy symbolic manipulation. Munola served as an excellent tool during both the technical and creative development of this project. In the end, the music is not written out in entirely in Munla, but the Munola backend is still used for symbolic representation.

White-key pitches of the equal-tempered scale are notated with upper case letters

```
C D E F G A B
```

The default octave is 4. A pitch can be raised or lowered one semitone by prepending `#` or `b`, multiple accidentals can be applied to the same note, e.g.

```
#C bC ##C bbC #bC
```

which evaluates to $C^\sharp$, $C^\flat(B)$, $C^{\flat\flat}(B^\flat)$, and just C. Accidentals are not persistent as in traditional western notation. The octave of a pitch can be raised and lowered by `^` and `_` e.g.

[3] `https://lilypond.org/`
[4] `https://abcnotation.com/`

```
^C _C _#C
```

which evaluates to $C5$, $C3$, and $C^\sharp 3$. The general octave can be selected by writing the octave number

```
3 C D E  5 F G A
```

which evaluates to $C3$, $D3$, $E3$, $F5$, $G5$, and $A5$.

The default duration of a note is one beat. The duration can be doubled using `+`, halved using `-`, and multiplied by 3/2 (dotted) using `.`, e.g.

```
+C ++C -C --C .+C
```

which evaluates to a two (2/4) beat, four (4/4) beat, 1/2 (1/8) beat, 1/4 (1/16) beat, and a three beat note.

Rests are notated using `R`. All modifiers apply to rests, but pitch modifiers have no effect.

A note can be marked as the end of a phrase using `~`, e.g.

```
C D ~E  R F G ~A
```

which are two phrases, the first ending on E and the second on A. The first note of the second phrase is a rest.

Notes can be accented by prepending `!`.

Munola supports functions which are notated by

```
function(arg1, arg2, arg3)
```

the function name cane be any string of lower-case characters. A function may have any number of arguments, including zero. Nested function calls are not supported. An example of a function employed during this project is `either`, e.g.

```
C E either(G, A, ^C)
```

which will evaluate to $C4, E4$, and $G4$, $A4$, or $C5$.

Chapter 4

Future Work and Conclusion

While the composition, synthesis, and interaction part is pretty much ready for exhibition, no time has been spent on visualization or building the Installation. The plan was to have the completed Installation finished at the end of May 2020. However, due to the global COVID-19 crisis, the Museum and university were closed for most of the project. Therefore, I geared the focus of the project more towards sound synthesis. The primary Future work is, of course, to finish the physical and visual part of the Installation, so that it may be exhibited at the Museum.

Another topic for future work could be to more directly use the results of the literature on the design of the Installation. For example, the NIME community has years of published results on the topic of musical interaction and novel interface. Similarly, research on multi-modal perception and cognition could provide a more robust empirical basis for the chosen design.

As seen in the discussion of the state-of-the-art musical installations, most installations at the Museum are designed with collaboration in mind. The Installation could also be made collaborative, perhaps by using two Leap Motion controllers — one for each voice.

More work could be done on making the playback more expressive. In fact, the combination of physical modeling and performance modeling might offer great results, but I haven't encountered it in the literature. So far, two KTH performance rules have been implemented, but more rules exist and have been refined since their introduction in [11]. Especially the timing-related rules might improve expressiveness.

While the Installation software runs decently in real-time, more effort could be spent on optimizing the code. This will probably become a necessity if visualizations have to be produced by the same computer running the synthesis algorithm. Parallelization of the voices is an obvious strategy to try. Also, the code produced by SymFds does not seem very efficient, and more optimization

routines could probably be implemented.

After some cleanup, most of the code modules created for the Installation could become useful stand-alone libraries or applications. The instrument models could be extracted and used to create two VST instruments. The pal-fds library, created in the early stages of the project, while not used in the end, could still be useful for other purposes. SymFds could be streamlined and made easier to use. Maybe in the future one could be able to enter differential equation and SymFds would be able to derive a suitable scheme, boundary conditions, perform stability analysis, and generate immediately runnable C++ code. Munola could be used in other projects as an easily extendable embedded music notation language.

4.1 Conclusion

This report has covered the development of an installation for the Danish Music Museum. In chapter 1 the premises for the project were introduced, and an overview of the report was given. Afterward, the state of the art of musical instruments was reviewed in the context of a composer/performer/conductor user role frame work. The conductor role, in which the user is allowed to make high-level structural decisions about the music to be played, was identified as the best option for the Installation. The review of the state of the art showed that most of the museum's installations tend towards the composer role. Afterward, the initial design idea for the Installation was described. The user would be able to act as a conductor of a virtual ensemble by using a Leap Motion to track their hand movements.

Chapter 2 covered the implementation of two virtual instruments, a bowed string and a struck bar, using the FDTD method. A short introduction to the subject was given, after which the instrument models were discussed in continuous and discrete time. The bowed string was modeled as a damped stiff string excited by a non-linear function mimicking the friction characteristic of a violin bow. A finger was modeled to add damping at a specific point along the string, shortening the section of string allowed to vibrate, which makes it possible to play different pitches. The struck percussion was modeled as a stiff thin damped bar of variable cross-section with free boundary conditions. Two spring-like connections model the suspension of the bar, and were used to keep the bar in place. To excite the bar, a collision with a mallet was modeled. The discretization of the bar called for the use of stretched coordinates. The implementation of the mallet model was done using a state-of-the-art non-iterative solution. The models were implemented using C++, and the software architecture was described.

The music to be played by the Installation was described in chapter 3 along with mapping and interactivity, rules to make the computer performance more expressive, and a description of the custom textual music notation language Munola. After a review of the history of algorithmic composition, the composition of the Installation was introduced. An algorithm of splitting notes was

used to generate rhythmic sequences. Tone row techniques were used to provide melodic content. A mapping was implemented that uses the elevation of the user's hands to control the melodic, rhythmic, and timbral intensity of the piece. To acheive a more expressive musical performance, three KTH performance rules were implemented to modulate the amplitude of the notes being played. Three options for musical notation were discussed, MIDI, LilyPond, and ABC, but none were found suitable. A custom language, Munola, was developed, and its syntax and semantics were described in detail.

In the current and final chapter, the possibilities for future work were discussed. The primary task is, of course, to finish the physical and visual part of the Installation. Other future goals include a more thorough study of interaction, making the Installation collaborative, making the playback more expressive, optimizing the application, and preparing the developed code modules for publishing and use in other projects.

Bibliography

[1] Edward Aldwell, Carl Schachter, and Allen Clayton Cadwallader. *Harmony & voice leading*. Schirmer/Cengage Learning, Boston, MA, 4th ed edition, 2011. OCLC: ocn144518118.

[2] Zdenêk P. Bažant and Luigi Cedolin. *Stability of structures: elastic, inelastic, fracture and damage theories*. World Scientific Pub, Hackensack, NJ ; London, world scientific ed edition, 2010. OCLC: ocn645707081.

[3] Julien Bensa, Stefan Bilbao, Richard Kronland-Martinet, and Julius O. Smith. The simulation of piano string vibration: From physical models to finite difference schemes and digital waveguides. *The Journal of the Acoustical Society of America*, 114(2):1095–1107, August 2003.

[4] Stefan Bilbao. A Modular Percussion Synthesis Environment. In *Proc. of the 12th Int. Conference on Digital Audio Effects*, volume 12, Como, Italy, 2009. 12th Int. Conference on Digital Audio Effects.

[5] Stefan Bilbao. *Numerical Sound Synthesis*. John Wiley & Sons, Ltd, Chichester, UK, October 2009.

[6] Stefan Bilbao. Finite-difference schemes in musical acoustics: A tutorial. In Rolf Bader, editor, *Springer Handbook of Systematic Musicology*, chapter 19, pages 349–384. Springer Berlin Heidelberg, New York, NY, 2018.

[7] Stefan Bilbao and Alberto Torin. Numerical Modeling and Sound Synthesis for Articulated String/Fretboard Interactions. *Journal of the Audio Engineering Society*, 63(5):336–347, June 2015.

[8] Michele Ducceschi and Stefan Bilbao. Non-iterative Solvers for Nonlinear Problems: The Case Of Collisions. page 8, 2019.

[9] Jose D. Fernandez and Francisco Vico. AI Methods in Algorithmic Composition: A Comprehensive Survey. *Journal of Artificial Intelligence Research*, 48:513–582, November 2013.

[10] Neville H. Fletcher and Thomas D. Rossing. *The Physics of Musical Instrument*. Springer, New York, 2nd ed edition, 1998.

[11] Anders Friberg. Generative Rules for Music Performance: A Formal Description of a Rule System. *Computer Music Journal*, 15(2):56, 1991.

[12] Anders Friberg, Roberto Bresin, and Johan Sundberg. Overview of the KTH rule system for musical performance. *Advances in Cognitive Psychology*, 2(2):145–161, January 2006.

[13] Donald Greenspan. Conservative Numerical Methods for x = f(x). *Journal of Computational Physics*, 56(1):28–41, October 1984.

[14] Bart Hopkin and John Scoville. *Musical instrument design: practical information of musical instrument making*. See Sharp Press, Tuscon, Arizona, seventh printing edition, 2010. OCLC: 838343377.

[15] Cheng-Zhi Anna Huang, Tim Cooijmans, Adam Roberts, Aaron Courville, and Douglas Eck. Counterpoint by Convolution. *arXiv:1903.07227 [cs, eess, stat]*, March 2019. arXiv: 1903.07227.

[16] Randall D Knight. *Physics for Scientists and Engineers: A Strategic Approach 4/E*. Pearson, 2017.

[17] Gareth Loy. *Musimathics: the mathematical foundations of music*. MIT Press, Cambridge, Mass. ; London, 2006. OCLC: ocm60856240.

[18] Max V. Mathews. The Radio Baton and Conductor Program, or: Pitch, the Most Important and Least Expressive Part of Music. *Computer Music Journal*, 15(4):37, 1991.

[19] Aaron Meurer, Christopher P. Smith, Mateusz Paprocki, Ondřej Čertík, Sergey B. Kirpichev, Matthew Rocklin, AMiT Kumar, Sergiu Ivanov, Jason K. Moore, Sartaj Singh, Thilina Rathnayake, Sean Vig, Brian E. Granger, Richard P. Muller, Francesco Bonazzi, Harsh Gupta, Shivam Vats, Fredrik Johansson, Fabian Pedregosa, Matthew J. Curry, Andy R. Terrel, Štěpán Roučka, Ashutosh Saboo, Isuru Fernando, Sumith Kulal, Robert Cimrman, and Anthony Scopatz. SymPy: symbolic computing in Python. *PeerJ Computer Science*, 3:e103, January 2017.

[20] George Papadopoulos and Geraint Wiggins. AI Methods for Algorithmic Composition: A Survey, a Critical View and Future Prospects. In *Proc. Intl. Computer Music Conf.*, page 8, Waseda University, Japan, 1993. International Computer Music Conference.

[21] Gary Potter. *The Role of Chance In Contemporary Music*. Ph.d. thesis, Indiana University, 1971.

[22] Alex Ross. *The Rest is Noise: Listening to the Twentieth Century*. Harper Perennial, London, harper perennial ed edition, 2009. OCLC: 837140014.

[23] Endre Süli and David Mayers. *An introduction to numerical analysis*. Cambridge University Press, Cambridge ; New York, 2003. OCLC: ocm50525488.

[24] Vesa Välimäki, Jyri Pakarinen, Cumhur Erkut, and Matti Karjalainen. Discrete-time modelling of musical instruments. *Reports on Progress in Physics*, 69(1):1–78, January 2006.

[25] Silvin Willemsen, Nikolaj Andersson, Stefania Serafin, and Stefan Bilbao. REAL-TIME CONTROL OF LARGE-SCALE MODULAR PHYSICALMODELS USING THE SENSEL MORPH. In *Proceedings of the 16th Sound and Music Computing Conference*, pages 151 – 158, Malaga, May 2019.

[26] Iannis Xenakis. *Formalized music: thought and mathematics in composition.* Number no. 6 in Harmonologia series. Pendragon Press, Stuyvesant, NY, rev. ed edition, 1992.

Appendix A

Jupyter Notebooks

This appendix contains the Jupyter Notebooks used to solve the FDSs of the stiff string and struck bar. Please excuse that the output do not fit on the page. The notebooks, as well as the SymFds source code, is provided with the hand-in.

Bowed and Fingered Stiff String

May 28, 2020

1 Solving the Bowed and Fingered Stiff String

Import SymPy and SymFds for symbolic math support and initialize printing.

```
In [1]: from sympy import *
        from symfds import *
        init_printing();
```

1.1 The String

The symbols needed for the string equation are

```
In [2]: k, h, c, kappa, sigma0, sigma1 = symbols('k h c kappa sigma_0 sigma_1')
        u = StateSymbol('u')
        F = StateSymbol('F')
```

The finite difference scheme for the string is

```
In [3]: stringFdsEq = Eq(dtt(u), c**2 * dxx(u) - kappa**2 * dxx(dxx(u))\
                    - 2 * sigma0 * dtc(u) + 2 * sigma1 * dtb(dxx(u)) + F)
        display(stringFdsEq)
```

$$\frac{1}{k^2}\left(-2u + un + up\right) = F + c^2\delta_{x+}(\delta_{x-}(u)) - \kappa^2\delta_{x+}(\delta_{x-}(\delta_{x+}(\delta_{x-}(u)))) - \frac{\sigma_0}{k}\left(un - up\right) + \frac{2\sigma_1}{k}\left(\delta_{x+}(\delta_{x-}(u)) - \delta_{x+}(\delta_{x}\right.$$

Solve for u^{n+1} to get the update rule

```
In [5]: stringUpdate = expandx(solution(stringFdsEq, nextt(u)))
        display(stringUpdate)
```

$$\frac{1}{k\sigma_0 + 1}\left(Fk^2 + 2u + upk\sigma_0 - up + \frac{c^2k^2}{h}\left(\frac{1}{h}\left(-u + uf\right) - \frac{1}{h}\left(u - ub\right)\right) - \frac{k^2\kappa^2}{h}\left(\frac{1}{h}\left(\frac{1}{h}\left(-\frac{1}{h}\left(-u + uf\right) + \frac{1}{h}\left(-uf\right.\right.\right.\right.$$

Use `optimize` and `cse` to remove constant terms and common sub expressions and print the C++ expressions for the resulting expressions

```
In [6]: (stringConstantTerms, stringOptimized) = optimize(stringUpdate)

        (stringCommonTerms, stringOptimized2) = cse(stringOptimized)

        stringOptimized2 = stringOptimized2[0]

        for (name, e) in stringConstantTerms:
            expStr = getCppExpression2(e, indexName='l')
            print(f'const float {name} = {expStr};')

        print('')

        for (name, e) in stringCommonTerms:
            expStr = getCppExpression2(e, indexName='l')
            print(f'const float {name} = {expStr};')

        print('')

        expStr = getCppExpression2(stringOptimized2, indexName='l');
        print(f'un[l] = {expStr};')
```

```
const float c0 = (1.0 / (1.0 + (k * sigma0)));
const float c1 = powf(k, 2.0);
const float c2 = powf(c, 2.0);
const float c3 = (1.0 / (h));
const float c4 = powf(kappa, 2.0);

const float x0 = 2.0 * u[l];
const float x1 = powf(c3, 2.0);
const float x2 = x1 * (x0 + (-1.0 * u[l-1]) + (-1.0 * u[l+1]));
const float x3 = 2.0 * k * sigma1;

un[l] = c0 * (x0 + (-1.0 * up[l]) + (c1 * F[l]) + (-1.0 * x2 * x3) + (k * sigma0 * up[l]) + (x
```

1.1.1 Boundary Conditions

For simply supported bondary conditions we first have

```
In [9]: stringUpdateLeft0Exp = N(0)
        display(Eq(u, stringUpdateLeft0Exp))
```

$$u = 0$$

and also

```
In [10]: simplySupportedDef = Eq(dxx(u), 0)
         display(simplySupportedDef)
```

$$\delta_{x+}(\delta_{x-}(u)) = 0$$

which can be solved to get

```
In [11]: simplySupportedEq =\
             solution(Eq(expandx(dxx(u)), 0).subs(u, 0), prevs(u))
         simplySupportedEq
```

Out[11]:

$$-uf$$

The C++ expression for this is

```
In [12]: stringUpdateLeft1 = stringOptimized.subs(prevs(prevs(u)), -u)
         stringUpdateLeft1Exp = getCppExpression2(stringUpdateLeft1).replace('l', '1')
         print(stringUpdateLeft1Exp)
```

```
c0 * ((-1.0 * up[1]) + (2.0 * u[1]) + (c1 * F[1]) + (k * sigma0 * up[1]) + (-1.0 * c1 * c2 * p
```

Now doing the same for the right side

```
In [14]: stringOptimizedRight = N(0)
         display((Eq(u, stringOptimizedRight)))
```

$$u = 0$$

```
In [15]: stringUpdateRight1 = stringOptimized.subs(nexts(nexts(u)), -u)
         stringUpdateRight1Exp = \
             getCppExpression2(stringUpdateRight1).replace('l', 'L-2')
         print(stringUpdateRight1Exp)
```

```
c0 * ((-1.0 * up[L-2]) + (2.0 * u[L-2]) + (c1 * F[L-2]) + (k * sigma0 * up[L-2]) + (-1.0 * c1 *
```

```
In [16]: stringUpdateRight0Exp = getCppExpression2(stringOptimizedRight, distributed=True)
         print(stringUpdateRight0Exp)
```

```
0.0
```

1.1.2 Stability conditions

To compute the minimal value of h use

```
In [17]: hmin = sqrt((c**2 * k**2 + 4 * sigma1 * k\
                      + sqrt((c**2 * k**2 + 4 * sigma1 * k)**2 + 16 * kappa**2 * k**2)) / 2)
         hmin
```

Out[17]:

$$\sqrt{\frac{c^2k^2}{2} + 2k\sigma_1 + \frac{1}{2}\sqrt{16k^2\kappa^2 + (c^2k^2 + 4k\sigma_1)^2}}$$

Which is implemented in C++ by

```
In [18]: hminExp = getCppExpression2(hmin).replace('_', '')
         print(f'float hmin = {hminExp};\nint L = 0.9 * floor(1 / hmin);')
```

```
float hmin = sqrt((0.5 * sqrt(powf((powf(c, 2.0) * powf(k, 2.0)) + (4.0 * k * sigma1), 2.0) +
int L = 0.9 * floor(1 / hmin);
```

1.2 The Bow

The bow model is defined by the equation

```
In [19]: Fb, Jb, vb = symbols('F_b J_b v_b')
         eta = StateSymbol('e')
         eta.isDistributed = False
         phi = Function('phi')(eta)
         Fbow = -Jb * Fb * phi
         Fbow
```

Out[19]:

$$-F_b J_b \phi(e)$$

Inserted into the string update equation we get

```
In [21]: stringFdsWithBow = stringFdsEq.subs(F, Fbow)
         stringFdsWithBow
```

Out[21]:

$$\frac{1}{k^2}(-2u + un + up) = -F_b J_b \phi(e) + c^2\delta_{x+}(\delta_{x-}(u)) - \kappa^2\delta_{x+}(\delta_{x-}(\delta_{x+}(\delta_{x-}(u)))) - \frac{\sigma_0}{k}(un - up) + \frac{2\sigma_1}{k}(\delta_{x+}(\delta_{x-}(u))$$

eta is the relative velocity between the bow and the string defined by

```
In [22]: etaEq = Eq(eta, dtc(u) - vb)
         etaEq
```

Out[22]:

$$e = -v_b + \frac{un - up}{2k}$$

We see that *eta* is dependent on u^{n+1}, so we must use iterative methods to solve it. First, put the equation into normal form

In [23]:
```
etaEqNormal = etaEq.rhs - etaEq.lhs
etaEqNormal
```

Out[23]:

$$-e - v_b + \frac{un - up}{2k}$$

An then substitue the solution of u^{n+1} of the string scheme into this

In [24]:
```
stringFdsWithBowUpdate = expandx(solution(stringFdsWithBow, nextt(u)))
etaEqNormalExpanded = etaEqNormal.subs(nextt(u), stringFdsWithBowUpdate)
etaEqNormalExpanded
```

Out[24]:

$$-e - v_b + \frac{1}{2k}\left(-up + \frac{1}{k\sigma_0 + 1}\left(2u + upk\sigma_0 - up - F_b J_b k^2 \phi(e) + \frac{c^2k^2}{h}\left(\frac{1}{h}\left(-u + uf\right) - \frac{1}{h}\left(u - ub\right)\right) - \frac{k^2\kappa^2}{h}\left(\frac{1}{h}\left(\frac{1}{h}\right.\right.\right.\right.$$

To compute the Newton-Raphson update we must take the derivate of this expression with respect to *eta*

In [78]:
```
etaEqNormalExpandedDiff = diff(etaEqNormalExpanded, eta)
etaEqNormalExpandedDiff
```

Out[78]:

$$-\frac{F_b J_b k \frac{d}{de}\phi(e)}{2\left(k\sigma_0 + 1\right)} - 1$$

Which is used in the definition of delta that is used to update *eta* (v_{rel})

In [28]:
```
delta = simplify(etaEqNormalExpanded / etaEqNormalExpandedDiff)
delta
```

Out[28]:

$$\frac{1}{h^4 k\left(F_b J_b k \frac{d}{de}\phi(e) + 2k\sigma_0 + 2\right)}\left(uph^4\left(k\sigma_0 + 1\right) + 2h^4 k\left(e + v_b\right)\left(k\sigma_0 + 1\right) + h^4\left(-2u - upk\sigma_0 + up + F_b J_b k^2 \phi(e)\right) + h$$

We can optimiza the delta expression

In [29]:
```
(deltaConstantTerms, deltaOptimized) = optimize(delta)
```

In [30]:
```
deltaConstantExp = '';

for (name, e) in deltaConstantTerms:
    deltaConstantExp += 'const float ' + str(name) + ' = '\
    + str(getCppExpression2(e)) + ";\n"

print(deltaConstantExp)
```

```
const float c0 = powf(h, -4.0);
const float c1 = (1.0 / (k));
const float c2 = 2.0 * k * sigma0;
const float c3 = powf(h, 4.0);
const float c4 = powf(k, 2.0);
const float c5 = powf(h, 2.0);
const float c6 = powf(c, 2.0);
const float c7 = 1.0 + (k * sigma0);
const float c8 = powf(kappa, 2.0);
```

And then compute delta using

```
In [32]: deltaExpReady = deltaOptimized.subs(diff(phi, eta),\
                                             Symbol('phid')).subs(phi, Symbol('phi'))
         deltaExp = getCppExpression2(deltaExpReady, distributed=False)
         print(deltaExp)
```

```
(1.0 / (c1)) * (1.0 / (2.0 + c2 + (Fb * Jb * k * phid))) * ((powf(c1, 2.0) * ((-2.0 * u) + (-1
```

To compute phi use

```
In [33]: alpha = symbols('alpha')
         phiEta = sqrt(2 * alpha) * eta * exp(-alpha * eta**2 + 0.5)
         phiEta
```

Out[33]:

$$1.64872127070013\sqrt{2}e\sqrt{\alpha}e^{-e^2\alpha}$$

```
In [34]: phiEtaExp = getCppExpression2(phiEta, distributed=True)
         phiEtaExp
```

```
Out[34]: '1.6487212707001282 * 1.4142135623730951 * sqrt(alpha) * e * (exp((-1.0 * alpha * pow
```

And to compute $\phi'\eta$

```
In [39]: diff(phiEta, eta)
```

Out[39]:

$$-3.29744254140026\sqrt{2}e^2\alpha^{\frac{3}{2}}e^{-e^2\alpha} + 1.64872127070013\sqrt{2}\sqrt{\alpha}e^{-e^2\alpha}$$

```
In [40]: phidEtaExp = getCppExpression2(diff(phiEta, eta), distributed=True)
         print(phidEtaExp)
```

```
(1.6487212707001282 * 1.4142135623730951 * sqrt(alpha) * (exp((-1.0 * alpha * powf(e, 2.0)))))
```

Finally, the code for computing the bow force is

```
In [41]: with open('template_nr.txt', 'r') as f:
             source = f.read()
             source = source.replace('<constantExp>', deltaConstantExp)
             source = source.replace('<etaInitExp>',\
                                     getCppExpression2(dtb(u) - vb, distributed=False))
             source = source.replace('<intermediateExp>',\
                                     f'float phi = {phiEtaExp};\n' + f'float phid = {phidEtaExp
             source = source.replace('<deltaExp>', deltaExp)
             source = source.replace('<forceExp>',\
                                     getCppExpression2(Fbow.subs(phi, Symbol('phi'))))
             source = source.replace('.at(l', '.at(lb')
             source = source.replace('_', '')
             print(source)
```

```
float delta = 1;
float e = (-1.0 * vb) + ((1.0 / (k)) * ((-1.0 * up) + u));
float Jb = 1.0 / h;
int maxIterations = 50;

const float c0 = powf(h, -4.0);
const float c1 = (1.0 / (k));
const float c2 = 2.0 * k * sigma0;
const float c3 = powf(h, 4.0);
const float c4 = powf(k, 2.0);
const float c5 = powf(h, 2.0);
const float c6 = powf(c, 2.0);
const float c7 = 1.0 + (k * sigma0);
const float c8 = powf(kappa, 2.0);


for (int i = 0; i < maxIterations && fabs(delta) > 1e-4; i++)
{
    float phi = 1.6487212707001282 * 1.4142135623730951 * sqrt(alpha) * e * (exp((-1.0 * alpha
float phid = (1.6487212707001282 * 1.4142135623730951 * sqrt(alpha) * (exp((-1.0 * alpha * pow


    delta = (1.0 / (c1)) * (1.0 / (2.0 + c2 + (Fb * Jb * k * phid))) * ((powf(c1, 2.0) * ((-2.
    e = e - delta;
}

F.at(lb) = -1.0 * Fb * Jb * phi;
```

1.3 Fingers

Fingers are modeled as a rigid connection between the stirng and an achor point. The force acting on the string is defined by

```
In [42]: omegaF, sigmaF, env, Jf= symbols('omega_f sigma_f env J_f')
         zeta  = StateSymbol('z')
         e = StateSymbol('e')
         FFin = env * Jf * (-omegaF**2 * utc(u) - 2 * sigmaF * dtc(u))
         FFin
```

Out[42]:

$$J_f env\left(-\omega_f^2\left(0.5un + 0.5up\right) - \frac{\sigma_f}{k}\left(un - up\right)\right)$$

This is plugged into the string equation to get

```
In [44]: stringFdsWithFinger = stringFdsEq.subs(F, FFin)
         display(stringFdsWithFinger)
```

$$\frac{1}{k^2}\left(-2u + un + up\right) = J_f env\left(-\omega_f^2\left(0.5un + 0.5up\right) - \frac{\sigma_f}{k}\left(un - up\right)\right) + c^2\delta_{x+}(\delta_{x-}(u)) - \kappa^2\delta_{x+}(\delta_{x-}(\delta_{x+}(\delta_{x-}(u))))$$

which is then solved for u^{n+1}

```
In [46]: stringFdsWithFingerUpdate = expandx(solution(stringFdsWithFinger, nextt(u)))
         stringFdsWithFingerUpdate
```

Out[46]:

$$\frac{1}{0.5J_f envk^2\omega_f^2 + J_f envk\sigma_f + k\sigma_0 + 1}\left(2u - 0.5upJ_f envk^2\omega_f^2 + upJ_f envk\sigma_f + upk\sigma_0 - up + \frac{c^2k^2}{h}\left(\frac{1}{h}\left(-u + uf\right) - \frac{1}{h}\right.\right.$$

The optimized C++ code to compute this is

```
In [47]: (commonTerms, stringFdsWithFingerUpdateOptimized) =\
             cse(stringFdsWithFingerUpdate)

         stringFdsWithFingerUpdateOptimized = stringFdsWithFingerUpdateOptimized[0]

         print(f'float Jf = (1 / h);')

         for (name, e) in commonTerms:
             expStr = getCppExpression2(e, indexName='lf', distributed = False)
             print(f'const float {name} = {expStr};')


         etanExp = getCppExpression2(stringFdsWithFingerUpdateOptimized,\
                                     distributed = False).replace('_', '')

         print(f'float en = {etanExp};\n')
```

```
        print(f'float ep = up;')
        print(f'float e = u;')

        FFinEta = stateSubs(FFin, StateSymbol('e'))

        print(f'float f = {getCppExpression2(FFinEta, False).replace("_", "")};')
```

```
float Jf = (1 / h);
const float x0 = k * sigma0;
const float x1 = Jf * env * k * sigmaf;
const float x2 = powf(k, 2.0);
const float x3 = 0.5 * Jf * env * x2 * powf(omegaf, 2.0);
const float x4 = -1.0 * up;
const float x5 = 2.0 * k * sigma1;
const float x6 = (1.0 / (h));
const float x7 = x6 * ((-1.0 * u) + uf);
const float x8 = x6 * ((-1.0 * ub) + u);
const float x9 = x6 * (x7 + (-1.0 * x8));
float en = (1.0 / (1.0 + x0 + x1 + x3)) * (x4 + (2.0 * u) + (x0 * up) + (x1 * up) + (x5 * x9) +

float ep = up;
float e = u;
float f = Jf * env * ((-1.0 * powf(omegaf, 2.0) * ((0.5 * en) + (0.5 * ep))) + (-1.0 * sigmaf +
```

Hammered Bar of Variable Cross Section

May 28, 2020

1 Solving the Hammered Bar of Variable Cross Section

Import SymPy and SymFds for symbolic math support and initialize printing.

```
In [1]: from sympy import *
        from symfds import *
        init_printing()
```

1.1 The Bar

The FDS for the bar of variable cross section is defined by

```
In [2]: alphaav, kappa, sigma0, sigma1 = symbols('alpha_av kappa sigma_0 sigma_1')
        p = StateSymbol('p') # phi
        u = StateSymbol('u')
        F = StateSymbol('F')

        # phi^(3 / 2)
        p32 = uxf(uxb(p**(5.0 / 2.0)) * uxb(p**(-0.5))**2.0)
        stiffness = -(kappa**2.0 / alphaav**4.0) *\
                    dxf(uxb(p**-0.5) * dxb(p**(5.0 / 2.0) * dxf(uxb(p**-0.5) * dxb(u))))
        damp = -2 * sigma0 * dtc(u)
        freqDamp = 2 * sigma1 * dtb((1 / alphaav**2) * p**-0.5 * dxf(uxb(p**-0.5) * dxb(u)));
        varBarFds = Eq(p32 * dtt(u), stiffness + damp + freqDamp + F)
        # varBarFds = Eq(dtt(u), -kappa**2 * dxx(dxx(u)))

        display(varBarFds)
```

$$\frac{1}{k^2}\left(-2u+un+up\right)\mu_{x+}\left(\mu_{x-}^{2.0}\left(p^{-0.5}\right)\mu_{x-}\left(p^{2.5}\right)\right)=F-\frac{\kappa^{2.0}}{\alpha_{av}^{4.0}}\delta_{x+}\left(\delta_{x-}\left(p^{2.5}\delta_{x+}\left(\delta_{x-}(u)\mu_{x-}\left(p^{-0.5}\right)\right)\right)\mu_{x-}\left(p^{-0.5}\right)\right)-\frac{\sigma_0}{k}($$

Which can be solved for u^{n+1}

```
In [4]: varBarUpdate = expandx(solution(varBarFds, nextt(u)))
        display(varBarUpdate)
```

$$\frac{1}{p^{0.5}pp^{0.5}\alpha_{av}^{8.0}\left(k\sigma_0+0.5\left(\frac{0.5}{p^{0.5}}+\frac{0.5}{pb^{0.5}}\right)^{2.0}(0.5p^{2.5}+0.5pb^{2.5})+0.5\left(\frac{0.5}{p^{0.5}}+\frac{0.5}{pf^{0.5}}\right)^{2.0}(0.5p^{2.5}+0.5pf^{2.5})\right)}\left(Fp^{0.5}pp^{0.5}\alpha_{av}^{8}\right.$$

The optimized C++ expression for this is

```
In [13]: (constExp, varBarUpdateOptimized) = optimize(varBarUpdate)
         (commonExp, varBarUpdateOptimized2) = cse(varBarUpdateOptimized)
         varBarUpdateOptimized2 = varBarUpdateOptimized2[0]

         for (name, exp) in constExp:
             expStr = getCppExpression2(exp)
             print(f'const float {name} = {expStr};')

         print('')

         for (name, exp) in commonExp:
             expStr = getCppExpression2(exp)
             print(f'const float {name} = {expStr};')

         print('')

         expStr = getCppExpression2(varBarUpdateOptimized2);
         print(f'un[l] = {expStr};')
```

```
const float c0 = powf(alphaav, -8.0);
const float c1 = k * sigma0;
const float c2 = powf(alphaav, 8.0);
const float c3 = powf(k, 2.0);
const float c4 = powf(alphaav, 6.0);
const float c5 = (1.0 / (h));
const float c6 = powf(alphaav, 4.0);
const float c7 = powf(kappa, 2.0);

const float x0 = powf(p[l], 3.0);
const float x1 = powf(p[l-1], 2.0);
const float x2 = sqrt(p[l-2]);
const float x3 = powf(p[l+1], 2.0);
const float x4 = sqrt(p[l+2]);
const float x5 = powf(pp[l], 1.5);
const float x6 = sqrt(pp[l-1]);
const float x7 = sqrt(pp[l+1]);
const float x8 = powf(p[l], 2.5);
const float x9 = powf(p[l-1], 2.5);
const float x10 = sqrt(p[l]);
const float x11 = (1.0 / (x10));
const float x12 = sqrt(p[l-1]);
```

```
const float x13 = x10 + x12;
const float x14 = powf(x11 * x13 * (1.0 / (x12)), 2.0) * (x8 + x9);
const float x15 = powf(p[l+1], 2.5);
const float x16 = sqrt(p[l+1]);
const float x17 = x10 + x16;
const float x18 = powf(x11 * x17 * (1.0 / (x16)), 2.0) * (x15 + x8);
const float x19 = sqrt(pp[l]);
const float x20 = powf(c5, 2.0);
const float x21 = x5 * x6 * x7 * powf(p[l], 2.0);
const float x22 = -1.0 * u[l+1];
const float x23 = x12 * x17;
const float x24 = x13 * ((-1.0 * u[l-1]) + u[l]);
const float x25 = x16 * x24;
const float x26 = x14 + x18;
const float x27 = (-1.0 * u[l]) + u[l+1];
const float x28 = x8 * ((-1.0 * x25) + (x23 * x27));

un[l] = (1.0 / (x0)) * (1.0 / (x1)) * (1.0 / (x2)) * (1.0 / (x3)) * (1.0 / (x4)) * (1.0 / (x5)
```

2 The Hammer

The hammer force is defined by

```
In [16]: psi, g, m, k = symbols('psi g m k')
         w = StateSymbol('w')
         w.isDistributed = False
         v = StateSymbol('v')
         v.isDistributed = False
         eta = u - w
         Fh = ((k * g / 2) * dtc(eta) + utb(v)) * g;
         display(Fh)
```

$$g\left(0.5v + 0.5vp + \frac{g}{4}\left(un - up - wn + \wp\right)\right)$$

The FDS for the hammer is

```
In [17]: hammerFds = Eq(dtt(w), - (1 / m) * Fh)
         hammerFds
```

Out[17]:

$$\frac{1}{k^2}\left(-2w + wn + \wp\right) = -\frac{g}{m}\left(0.5v + 0.5vp + \frac{g}{4}\left(un - up - wn + \wp\right)\right)$$

The hammer force can be plugged into the bar FDS to get

```
In [18]: Jh, rho = symbols('J_h rho')
         varBarFdsWithHammer = Eq(varBarFds.lhs, varBarFds.rhs + (1 / rho) * Jh * Fh)
         varBarFdsWithHammer
```

Out[18]:

$$\frac{1}{k^2}\left(-2u+un+up\right)\mu_{x+}\left(\mu_{x-}^{2.0}\left(p^{-0.5}\right)\mu_{x-}\left(p^{2.5}\right)\right)=F+\frac{J_h g}{\rho}\left(0.5v+0.5vp+\frac{g}{4}\left(un-up-wn+\wp\right)\right)-\frac{\kappa^{2.0}}{\alpha_{av}^{4.0}}\delta_{x+}\left(\delta_{x-}\right.$$

Which can be solved as two equations with two unknowns for u^{n+1} and w^{n+1}

In [19]:
```
solutionSet = solve((hammerFds, varBarFdsWithHammer), nextt(w), nextt(u))

hammerUpdate = expandx(solutionSet[nextt(w)])
varBarWithHammerUpdate = expandx(solutionSet[nextt(u)])

display(simplify(hammerUpdate))
display(varBarWithHammerUpdate)
```

$$\frac{1}{pb^2\sqrt{pbb}pf^2\sqrt{pff}pp\sqrt{ppb}\sqrt{ppf}\alpha_{av}^4h^4\left(ppbpfJ_hg^4k^4+\left(g^2k^2-4.0m\right)\left(ppbpfk\left(-J_hg^2k+4.0\rho\sigma_0\right)+0.25\rho\left(pb\left(\sqrt{\right.\right.\right.\right.}$$

$$\frac{1}{\sqrt{p}\sqrt{pp}\alpha_{av}^4\left(J_hg^4k^4+\left(g^2k^2-4.0m\right)\left(-J_hg^2k^2+4.0k\rho\sigma_0+4.0\rho\left(0.5\left(0.5p^{\frac{5}{2}}+0.5pb^{\frac{5}{2}}\right)\left(\frac{0.5}{\sqrt{pb}}+\frac{0.5}{\sqrt{p}}\right)^2+0.5\left(0.5p\right.\right.\right.\right.}$$

The C++ expression for the bar update at the hammer point is

In [20]:
```
(commonExp, varBarWithHammerUpdateOptimized) = cse(varBarWithHammerUpdate)

varBarWithHammerUpdateOptimized = varBarWithHammerUpdateOptimized[0]

for (name, exp) in commonExp:
    expStr = getCppExpression2(exp, indexName='lh')
    print(f'const float {name} = {expStr};')

print('')

expStr = getCppExpression2(varBarWithHammerUpdateOptimized, indexName='lh');
print(f'un[lh] = {expStr};')
```

```
const float x0 = sqrt(p[lh]);
const float x1 = (1.0 / (x0));
const float x2 = sqrt(pp[lh]);
const float x3 = (1.0 / (x2));
const float x4 = powf(alphaav, 4.0);
const float x5 = 4.0 * m;
const float x6 = powf(g, 2.0);
const float x7 = powf(k, 2.0);
```

```
const float x8 = x6 * x7;
const float x9 = x8 + (-1.0 * x5);
const float x10 = 4.0 * k * rho * sigma0;
const float x11 = powf(p[lh], 2.5);
const float x12 = 0.5 * x11;
const float x13 = powf(p[lh-1], 2.5);
const float x14 = 0.5 * x1;
const float x15 = 0.5 * powf(p[lh-1], -0.5);
const float x16 = x14 + x15;
const float x17 = powf(p[lh+1], 2.5);
const float x18 = 0.5 * powf(p[lh+1], -0.5);
const float x19 = x14 + x18;
const float x20 = (0.5 * powf(x16, 2.0) * (x12 + (0.5 * x13))) + (0.5 * powf(x19, 2.0) * (x12 +
const float x21 = 4.0 * rho * x20;
const float x22 = x0 * x2 * x4;
const float x23 = 2.0 * g * x7;
const float x24 = x23 * v;
const float x25 = x23 * vp;
const float x26 = x8 * wp;
const float x27 = x8 * up[lh];
const float x28 = x0 * x2 * x4 * up[lh];
const float x29 = Jh * x0 * x2 * x4;
const float x30 = (1.0 / (h));
const float x31 = x0 * x30;
const float x32 = powf(alphaav, 2.0);
const float x33 = 0.5 * x3;
const float x34 = x19 * x30;
const float x35 = x34 * ((-1.0 * u[lh]) + u[lh+1]);
const float x36 = x16 * x30;
const float x37 = x36 * ((-1.0 * u[lh-1]) + u[lh]);
const float x38 = x35 + (-1.0 * x37);
const float x39 = x11 * x30 * x38;

un[lh] = x1 * x3 * (1.0 / (x4)) * (1.0 / ((x9 * (x10 + x21 + (-1.0 * Jh * x8))) + (Jh * powf(g
```

The C++ expression for the hammer update is

```
In [21]: (commonExp, hammerUpdateOptimized) = cse(hammerUpdate)

         hammerUpdateOptimized = hammerUpdateOptimized[0]

         for (name, exp) in commonExp:
             expStr = getCppExpression2(exp, indexName='lh')
             print(f'const float {name} = {expStr};')

         print('')
```

```
        expStr = getCppExpression2(hammerUpdateOptimized, indexName='lh');
        print(f'wn = {expStr};')
```

```
const float x0 = sqrt(p[lh]);
const float x1 = (1.0 / (x0));
const float x2 = sqrt(pp[lh]);
const float x3 = (1.0 / (x2));
const float x4 = powf(alphaav, 4.0);
const float x5 = 4.0 * m;
const float x6 = powf(k, 2.0);
const float x7 = x6 * powf(g, 2.0);
const float x8 = 4.0 * k * rho * sigma0;
const float x9 = powf(p[lh], 2.5);
const float x10 = 0.5 * x9;
const float x11 = powf(p[lh-1], 2.5);
const float x12 = 0.5 * x1;
const float x13 = 0.5 * powf(p[lh-1], -0.5);
const float x14 = x12 + x13;
const float x15 = powf(p[lh+1], 2.5);
const float x16 = 0.5 * powf(p[lh+1], -0.5);
const float x17 = x12 + x16;
const float x18 = (0.5 * powf(x14, 2.0) * (x10 + (0.5 * x11))) + (0.5 * powf(x17, 2.0) * (x10 +
const float x19 = 4.0 * rho * x18;
const float x20 = x19 + x8 + (-1.0 * Jh * x7);
const float x21 = x0 * x2 * x4;
const float x22 = 2.0 * g * x6;
const float x23 = x22 * v;
const float x24 = x22 * vp;
const float x25 = x7 * wp;
const float x26 = x7 * up[lh];
const float x27 = x0 * x2 * x4 * up[lh];
const float x28 = Jh * x0 * x2 * x4;
const float x29 = (1.0 / (h));
const float x30 = x0 * x29;
const float x31 = powf(alphaav, 2.0);
const float x32 = 0.5 * x3;
const float x33 = x17 * x29;
const float x34 = x33 * ((-1.0 * u[lh]) + u[lh+1]);
const float x35 = x14 * x29;
const float x36 = x35 * ((-1.0 * u[lh-1]) + u[lh]);
const float x37 = x34 + (-1.0 * x36);
const float x38 = x29 * x37 * x9;

wn = x1 * x3 * (1.0 / (x4)) * (1.0 / ((x20 * (x7 + (-1.0 * x5))) + (Jh * powf(g, 4.0) * powf(k
```

Appendix B

Derivations

B.1 Interpolation and Spreading Operators

Discretized spatial models are by default only accessible at grid points (multiples of h). Access to values in between grid points is handled by *interpolation operators* [5, chapter 5].

Given some real-valued position x along the domain of the model the simplest, zeroth order, interpolation operator is

$$I_0(x)u_l = u_{l_0}, \tag{B.1}$$

where $l_0 = \text{floor}(x/h)$. A first order interpolation operator uses linear interpolation and is defined by

$$I_1(x)u_l = (1-\alpha)u_{l0} + \alpha u_{l_0+1} \tag{B.2}$$

where $\alpha = (x/h) - l_0$. Higher order interpolators can be constructed as one wishes, see e.g. [5, chapter 5] for the definition of cubic interpolators.

Related to interpolation are spreading operators which allow one to spread some value over a grid function at some fractional index and can be thought of as the discrete version of the Dirac delta function. The zeroth order spreading operator is defined as [5, chapter 5]

$$J_0(x) = \frac{1}{h}\begin{cases} 1 & l = l_0 \\ 0 & \text{otherwise} \end{cases} \tag{B.3}$$

that is, the desired value is multiplied by $1/h$ and applied to the grid point at the truncated index. The first order spreading operator is

$$J_0(x) = \frac{1}{h}\begin{cases} (1-\alpha) & l = l_0 \\ \alpha & l = l_0 + 1 \\ 0 & \text{otherwise} \end{cases} \tag{B.4}$$

where the value is scaled by $1-\alpha$ at point l_0 and α at l_0+1. As with interpolation operators higher order spreading operators can be constructed, see [5, chapter 5].

B.2 Bar of Variable Cross Section with Change of Variables

Given the definition of α from Equation (2.62), using the fundamental theorem of calculus, we have

$$\alpha' = \frac{1}{\alpha_{\text{av}}}\theta^{-1/2} \tag{B.5}$$

Focusing on the stiffness term of Equation (2.51) we have

$$\kappa_0^2(\theta^3 u_{xx})_{xx} \quad \rightarrow \quad \kappa_0^2\alpha' \left(\alpha'(\theta^3\alpha'(\alpha' u_\alpha)_\alpha)_\alpha\right)_\alpha . \tag{B.6}$$

Substituting α' for Equation (B.5) we get

$$\kappa_0^2 \frac{1}{\alpha_{\text{av}}}\theta^{-1/2}\left(\frac{1}{\alpha_{\text{av}}}\theta^{-1/2}\left(\theta^3\frac{1}{\alpha_{\text{av}}}\theta^{-1/2}\left(\frac{1}{\alpha_{\text{av}}}\theta^{-1/2}u_\alpha\right)_\alpha\right)_\alpha\right)_\alpha$$

$$\frac{\kappa_0^2}{\alpha_{\text{av}}^4}\theta^{-1/2}\left(\theta^{-1/2}\left(\theta^3\theta^{-1/2}\left(\theta^{-1/2}u_\alpha\right)_\alpha\right)_\alpha\right)_\alpha$$

$$\frac{\kappa_0^2}{\alpha_{\text{av}}^4}\theta^{-1/2}\left(\theta^{-1/2}\left(\theta^{5/2}\left(\theta^{-1/2}u_\alpha\right)_\alpha\right)_\alpha\right)_\alpha .$$

B.3 Non-iterative Solution of Mallet

Starting with ϕ_u we have

$$\phi_u = \frac{d\phi}{du} = \frac{d\phi}{du}\frac{dt}{dt} = \frac{d\phi}{dt}\frac{dt}{du} = \frac{\phi_t}{u_t} \tag{B.7}$$

Letting

$$\psi = \sqrt{2\phi} \tag{B.8}$$

one may write

$$\phi_u = \psi\psi_u = \psi\frac{\psi_t}{u_t} \tag{B.9}$$

because, by the chain rule,

$$\psi\psi_u = \sqrt{2\phi}\frac{\sqrt{2}}{2\sqrt{\phi}}\phi_u = \phi_u. \tag{B.10}$$