UNIVERSITY OF CALIFORNIA, SANTA BARBARA

Hearing the Infinitesimal:

From Classical to Quantum Music

A dissertation submitted in partial satisfaction of the

requirements for the degree

Doctor of Philosophy in Music

by

Rodney DuPlessis

Committee:

Professor João Pedro Oliveira, Chair

Professor Curtis Roads

Professor JoAnn Kuchera-Morin

December 2021

The Dissertation of Rodney DuPlessis is approved.

Curtis Roads

JoAnn Kuchera-Morin

João Pedro Oliveira, Committee Chair

December 2021

Hearing the Infinitesimal:

From Classical to Quantum Music

by

Rodney DuPlessis

ACKNOWLEDGMENTS

This dissertation was supported by a generous Graduate Division Dissertation Fellowship. I thank the UCSB Graduate Division for their interest and investment in my work.

The journey of science and of music is a communal effort. Only through collaboration, sharing, and humility can we advance in our collective understanding. I am eternally grateful to my cohort and colleagues for our countless stimulating conversations and for holding space for inspiration and creativity to flourish. Thank you to Clarence Barlow for cultivating this community and ethos at UCSB, for countless long nights of libation and good company, and for showing me what unrestrained curiosity in composition can achieve. *Come up and see me sometime.* Thank you to Curtis Roads for offering advice, providing encouragement, opening opportunities, cat-sitting, and for showing me the stress-mitigating power of sailing.

I thank my committee for their support and guidance. Thank you to João Pedro Oliveira for comments on an early draft of this document, for being instrumental in my application for the Graduate Division Dissertation Fellowship, and most of all for helping to sharpen my musical instinct that is referenced often in this document.

I also thank Andres Cabrera, JoAnn Kuchera-Morin, and the Allosphere Research Group for their work on Allolib, which provided a foundation for the software I present in this document. A special thank you to Karl Yerkes who taught me nearly everything I know about programming, empowering me to create tools to realize my musical vision and to continue learning on my own.

I thank my mother for teaching me empathy and open-mindedness and I thank my father for teaching me the joy of problem solving and for encouraging my curiosity. This upbringing made me who I am and guided me here.

This dissertation is dedicated to my wife, without whom I would not have discovered a place where my musical goals would be nurtured and I would not be earning a PhD today.

VITA OF RODNEY DUPLESSIS

December 2021

EDUCATION

2021 Doctor of Philosophy, Music Composition
University of California, Santa Barbara, USA

2021 Master of Science, Media Arts & Technology
University of California, Santa Barbara, USA

2019 Master of Arts, Music Composition
University of California, Santa Barbara, USA

2013 Bachelor of Arts, Interdisciplinary Major in Music and Psychology (with distinction)
St. Thomas University, Fredericton, Canada

AWARDS, GRANTS, FELLOWSHIPS

2021 Dorothy and Sherrill C. Corwin Award for Excellence in Composition
1st prize in solo/chamber category for Coacervate

2020 Musica Nova International Electroacoustic Music Competition 2020
Finalist (De Rerum Natura)

Destellos International Electroacoustic Competition 2020
Honorable Mention (De Rerum Natura)

SIMEC Electroacoustic Music Competition 2020
Finalist (De Rerum Natura)

Graduate Division Dissertation Fellowship

MAT Merit Supplemental Support Research Stipend

ASCAP/SEAMUS Award
Finalist (De Rerum Natura)

2019 Summer Culture and Community Grant
UCSB Summer Music Festival

Dorothy and Sherrill C. Corwin Award for Excellence in Composition
1st prize in Percussion category for Sisyphe Heureux

Dorothy and Sherrill C. Corwin Award for Excellence in Composition
2nd prize in Electroacoustic category for Glossopoeia

Peter Racine Fricker Research Stipend

2018 International Doctoral Recruitment Fellowship

Summer Cultural and Enrichment Grant
UCSB Summer Music Festival

Corwin Grant
UCSB Summer Music Festival

AWARDS, GRANTS, FELLOWSHIPS (cont.)

2017 The Peter Racine Fricker Fellowship for Studies in Composition
UCSB Regents Departmental Fellowship
International Doctoral Recruitment Fellowship
Peter Racine Fricker Research Stipend

2016 The Peter Racine Fricker Fellowship for Studies in Composition
UCSB Regents Departmental Fellowship

ACADEMIC APPOINTMENTS

2021 Teaching Associate - Intermediate Composition (Instructor of Record)
University of California, Santa Barbara
Teaching Associate - Class Composition (Instructor of Record)
University of California, Santa Barbara

2018-2020 Teaching Assistant – Computer Music
University of California, Santa Barbara

2010-2012 Tutor – Music Department
St. Thomas University, Fredericton, CAN
Tutor – Psychology Department
St. Thomas University. Fredericton, CAN

PROFESSIONAL ACTIVITIES

2021- Co-Creative Director–Nomadic Soundsters
Online
CREATE Technical Coordinator
University of California, Santa Barbara

2020 A1 Engineer–Alliance of Women in Media Arts and Science Conference
University of California, Santa Barbara

2019 Co-Artistic Director–UCSB Summer Music Festival
University of California, Santa Barbara
A1 Engineer – Alliance of Women in Media Arts and Technology Conference
University of California, Santa Barbara

2018 Executive Director – UCSB Summer Music Festival
University of California, Santa Barbara
Technical Director – Alliance of Women in Media Arts and Technology Conference
University of California, Santa Barbara

PROFESSIONAL ACTIVITIES (cont.)

2017 Composition Forum Coordinator
University of California, Santa Barbara

2016 Research Assistant – First-Year Seminars
University of California, Santa Barbara – Supervisor: Clarence Barlow

CONFERENCE PRESENTATIONS & GUEST LECTURES

Sep 23, 2021 Visiones Sonoras
Performance: Coacervate

Sep 22, 2021 Visiones Sonoras
Presentation: CHON: A Physics-Based LFO Array

Aug 28, 2021 2021 SCI National Student Conference
Performance: Coacervate

Apr 24, 2021 SEAMUS 2021 National Conference
Performance: Coacervate

Mar 19, 2021 Sonic Art (Colby College)
Guest Lecture: Algorithmic Composition

Feb 23, 2021 Composition LAB (Estonian Academy of Music and Theatre)
Guest Lecture: CHON: from physics to musical gesture

Oct 30, 2020 Composition Forum (University of California Santa Barbara)
Presentation: Controlling EmissionControl2

Apr 22, 2020 Earth Day Art Model 2020
Performance: HeatWaves (Co-Presenters: SUDO Ensemble)

Mar 12, 2020 SEAMUS 2020 National Conference
Performance: Dimensionless

Feb 4, 2020 Alliance of Women in Media, Arts, & Sciences Conference 2020
Workshop: Introduction to Computer Music Programming in Pure-Data (Co-Presenter: Elizabeth Hambleton)

Jan 18, 2020 Womxn/Hacks 2.0
Workshop: Intro to Computer Music Programming

Nov 8, 2019 NowNet Arts Conference 2019
Performance: Trans-Pacific Concert between University of California Santa Barbara and the Elder Conservatorium of Music in Adelaide

May 2, 2019 Sound + Science Symposium 2.0, UCLA
Performance & Panel: Artsmesh Network Music Performance and Discussion (Co-Presenters Ken Fields, Joel Ong, Gil Kuno)

CONFERENCE PRESENTATIONS & GUEST LECTURES (cont.)

Oct 30, 2018 Art22: Introduction to Programming for the Arts (UCSB)
Guest Lecture: Coding as Compositional Process

Aug 8, 2018 International Computer Music Conference 2018, Daegu, South Korea
Performance: Bachflip

PUBLICATIONS

2021 "Architecture for Real-Time Granular Synthesis: EmissionControl2"
(co-author with Curtis Roads and Jack Kilgore)
Unpublished manuscript (submitted).

"CHON: From physical simulation to musical gesture" Unpublished thesis submitted in partial fulfillment of the Master of Science degree in Media Arts and Technology, UC Santa Barbara.

SELECTED LIST OF WORKS & PERFORMANCES

2021 *Psi* (Electroacoustic)
Dec 2021: Rodney DuPlessis, PhD Recital, Santa Barbara (online), Premiere

Coacervate (Violin and electronics)
1st prize Dorothy and Sherrill C. Corwin Award in Solo/Chamber category
Sep 2021: Chelsea Edwards, Visiones Sonoras, Morelia, Michoacán, Mexico (online)
Aug 2021: Chelsea Edwards, SCI National Student Conference, Online
Jun 2021: Chelsea Edwards, Synesthesias: New Music at UCSB 2, Santa Barbara (online)
Apr 2021: Chelsea Edwards, SEAMUS 2021, Online
Dec 2020: Chelsea Edwards, ECM Concert, Santa Barbara (online), Premiere

2020 *Pandæmonium* (Piano four hands)
Aug 2020: HOCKET, UCSB Summer Music Festival 2020, Santa Barbara (Online), Premiere

Oscilla (Interactive exhibit)
Feb 2022 – May 2022: MOXI, Santa Barbara
Oct 2020 – Ongoing: The Museum of Sensory and Movement Experiences, Santa Barbara

SELECTED LIST OF WORKS & PERFORMANCES (cont.)

2020 *De Rerum Natura* (Electroacoustic)
ASCAP/SEAMUS Award 2020 Finalist
SIMEC Electroacoustic Music Competition 2020 Finalist
Destellos International Electroacoustic Competition 2020 Honorable Mention
Musica Nova International Electroacoustic Music Competition 2020 Finalist
Oct 2021: Rodney DuPlessis, Musiques Démesurées, Clermont-Ferrand, France
Dec 2020: Rodney DuPlessis, Musica Nova Concert of finalists, Prague, Czech Republic (online)
Mar 2020: Rodney DuPlessis, SEAMUS 2020, Charlottesville, Virginia, (online)
Aug 2019: Rodney DuPlessis (live diffusion on the Musiques & Recherches Acousmonium), Influx Summer Festival, Brussels, Belgium, Premiere

2019 *Mysterium Cosmographicum* (Oscilloscope)
Aug 2019: UCSB Summer Music Festival 2019, Santa Barbara
Jun 2019: MAT End of Year Show, Santa Barbara
Mar 2019: First Thursday @ SBCAST, Santa Barbara, Premiere

2018 *Sisyphe Heureux* (Percussion quartet and electronics)
1st prize Dorothy and Sherrill C. Corwin Award in Percussion category
Aug 2018: Los Angeles Percussion Quartet, Santa Barbara, Premiere

Disconnect (Saxophone and live electronics)
Nov 2018: Henrique Portovedo, I Encontro de LiveLoopists, Aveiro, Portugal
Jun 2018: Henrique Portovedo, ICLI 2018, Porto, Portugal
Mar 2018: Henrique Portovedo, Santa Barbara, Premiere

Quinto Suono (String quartet)
Jan 2018: Formalist Quartet, Santa Barbara, Premiere

2017 *2fl* $\vee$ $\neg$*2fl* (Flute duo)
Aug 2017: Adriane Hill and Cynthia Vong, Santa Barbara, Premiere

BachFlip (Electroacoustic)
Aug 2018: International Computer Music Conference (ICMC) 2018, Daegu, Korea
Jun 2017: Santa Barbara, Premiere

SELECTED LIST OF WORKS & PERFORMANCES (cont.)

2017 *Feedback* (Double bass and live electronics)
May 2017: Scott Worthington, Santa Barbara, Premiere

2016 *Glossopoeia* (Electroacoustic)
2nd prize Dorothy and Sherrill C. Corwin Award in Electroacoustic category
Apr 2018: CEMEC, CCRMA/Stanford
Apr 2018: CEMEC, Santa Barbara
Nov 2017: CREATE presents: Sound Resistance, Santa Barbara, Premiere

Surface Tension (Electroacoustic)
Apr 2017: CEMEC, Mills College
Apr 2017: CEMEC, San Diego, Premiere

ABSTRACT

Hearing the Infinitesimal: From Classical to Quantum Music

by

Rodney DuPlessis

Scientists such as Einstein, Boltzmann, Leibniz, and Kepler understood that musical and scientific thought are deeply linked. Composers have incorporated science into music in literal and metaphorical ways, especially within the last century. In this same span of time, new models in science and music have proliferated. Aside from a few projects, little has been done to meaningfully link quantum physics and music. This dissertation documents my efforts to represent science and, in particular, quantum physics in music through theory, software development, and interpretation. A broad discussion of scientific metaphor in music using two of my compositions as case studies is followed by a look at classical physics in composition and in designing the musical software CHON. In the following chapters, quantum physics and music are brought together through an introduction and examination of my new software application, QHOSYN, and my newest piece, *Psi*.

Contents

List of Figures

List of Media Examples[1]

[1]Examples in this list refer to examples in the text. If you click an item in this list, you will be taken to the example in the text. If you click the example in the text, you will be taken to the media example online. If you are reading a paper copy of this document, or if your pdf reader does not support hyperlinks, you may navigate to the media examples manually in an internet browser at the following address: https://rodneyduplessis.com/dissertation/media

Chapter 1

Introduction

At the smallest scales, everything in the universe is oscillating. Elementary particles are characterized by a wavefunction that oscillates at a rate proportional to their mass. Even spacetime oscillates, as confirmed by recent measurements of gravitational waves[1]. Each in their own way, scientists and musicians probe our oscillating universe to reveal resonating patterns.

This dissertation is broadly an account of the gradual and inexorable assimilation of scientific metaphors into my music. The process has entailed roughly equal parts mathematical and scientific study and aesthetic experimentation and reflection. The culmination of this union thus far is reflected in my music and software based on quantum mechanics.

I employ two complementary methods in composing music based on science: sonification and interpretation. These form a duality that mirrors the difference in quantum physics between theory and interpretation. Put simply, quantum theory has given us lasers and modern electronics, while quantum interpretation has given us Schrödinger's undead cat and parallel universes. The latter is useful for more than just science fiction, however, as it exerts a steering force on new research and theories.

Between sonification and interpretation there exists a gradient. Sonification can become

progressively more interpretive as data is manipulated, transformed, transposed, and squeezed. Interpretive methods of representing science in music can range from iconic and representative, such as using the sound of a Newton's cradle to represent classical physics, to impressionistic and abstract, such as making synthesized sounds interact in physical ways. A composer can also strive to remove themselves from the sonification as much as possible and rely solely on algorithms that generate the music form data or simulations. Even if this is desirable, this is an unachievable ideal, as the composer's influence permeates the project form the moment they choose a strategy for mapping number to sound.

Music composition is not science, but there is a great deal to be gained from a union between the two. There is no real objective ideal to strive for in music as there is in science. Though composers and scientists do share some similarities in their working methods, the composer's final goal is an aesthetic/philosophical one. An artist or a philosopher takes what they can from science and plays with it, shines different light on it, translates it to different senses, and re-frames it in different perspectives.

Ultimately, my goal in connecting science and music is twofold. First, as a composer, I aim to to discover new means of musical expression through exploration of scientific models. Second, I strive to inspire and provoke new perspectives by creating music and software instruments that encode something fundamental about the world. As with all of my work, I see this journey as a communal effort.

The work presented here did not emerge from a vacuum. Two overarching themes are at the core of this research: A quantum conception of sound, and more broadly the idea of generating sound from data and simulations.

Dennis Gabor was the first to introduce a quantum notion of sound[2][3]. Inspired by this, Iannis Xenakis developed a theory for music composition using sound grains[4]. Xenakis conceived of arranging these sound grains according to probabilistic laws derived

from information theory and thermodynamics. Later, Curtis Roads greatly expanded the theory of microsound and granular synthesis[5]. Much of this work in formalizing a granular approach to sound focused on the novel idea of treating sound as a mass of particles. This has opened an enormous field of possibilities in sound analysis, synthesis, and composition[6]. Conversely to how treating an electron as a wave was a paradigm shift for physicists in the early 20th century, treating the traditionally wave-like phenomenon of sound as a particle was a paradigm shift for musicians in the mid- to late-20th century.

In my research I have returned to Gabor's original meaning by treating sound as quantum particles in the most literal sense. These quantum particles, governed by the mathematics of quantum mechanics, are well known to exhibit characteristics of both particles and waves. The research presented in chapter 4, 5, and 6 expands on this. I make use of microsound theory in addition to several other methods of synthesis. At the core of this is a stochastic theory based on the time-varying probability distributions of quantum wavefunctions. Thus, this work is close in spirit to Xenakis' pioneering work incorporating stochastic theory into musical composition[4].

The process of turning gathered or simulated data into sound is commonly called data sonification. By now the field of sonification has established itself as an important interdisciplinary endeavor (see [7] for a glimpse of the scope if the field). Sonification, or auditory display, can be used as an alternative to visualization for assistive technology and making sense of data, among other applications.

In my research, sonification is only a starting point. The sonification of the simulations provides material with which I can compose. Carla Scaletti has written about this distinction between sonification and data-driven music[8]. She defines sonification as:

> ... a mapping from data generated by a model, captured in an experiment, or otherwise gathered through observation to one or more parameters of an audio signal or sound synthesis model for the purpose of better understanding,

communicating or reasoning about the original model, experiment or system.

On the other hand, data-driven or algorithmic music need not share the same goals. It is music first and foremost. At an even lower level, before the final musical result is even considered, the sonification tools I describe here are not designed as simple data printing machines but as instruments to be played. In this way the mark of the composer is embedded at every level, sculpting, but not obfuscating, data into convincing metaphors and compelling narratives.

In this dissertation, I set out from my previous work in sonifying simulations of classical physics to venture into the realm of quantum mechanical simulations[9]. This work in classical simulations bears some relation to physical modeling (PM) synthesis. The mass-spring method of PM synthesis was pioneered by researchers such as Lejaren Hiller, Pierre Ruiz, and Claude Cadoz[10][11]. It is a way to simulate a string, membrane, or other construction using atomic oscillating nodes connected by spring-like forces. I have a different goal in using the mass-spring model. In my software, CHON, I simulate coupled oscillators in order to extract data about the motion of each individual oscillator. In this way, a new kind of low frequency oscillator paradigm is made available for physically coupled musical gestures.

The subject of quantum physics in sonification has attracted relatively little attention. Rajmil Fischman wrote a piece of software and an accompanying musical experiment that uses Schrödinger's equation to derive the form of a piece[12]. Bob Sturm made a foray into quantum mechanics when he attempted a sonification based on particles in an atom trap[13]. Using MATLAB, Sturm set up a simulation where 50 "particles" were represented by sine waves swooping up and down in frequency, corresponding to their energy levels, and varying the amplitude as a particle approaches and recedes from a stationary observer. The sounds were rendered offline. According to Sturm, rendering 1 minute of the simulation took about 7 hours complete. The result of this work is the piece

50 Particles in a Three-Dimensional Harmonic Potential: An Experiment in 5 Movements (1999) in which 50 sine waves drift up and down in frequency and amplitude for ten minutes.

Sturm avoided the quantum wave function, choosing to focus on the duality of energy-frequency in quantum particles and otherwise staying within the realm of Newtonian Mechanics:

> Initial work attempted to sonify the non-stationary wave functions derived from Schrödinger's equation, but this immediately led to interpretation problems. Specifically, what relation can be created between an energy–frequency distribution and a momentum–probability distribution for a particle in some quantum state? The mapping is not clear enough. A circuitous route instead, remaining in the safe clutches of classical mechanics and borrowing the wave–particle duality of QM, proves much more immediately productive. [13, p. 132]

I find no such problem in interpretation, though I am admittedly approaching the quantum system from a different, more fundamental perspective. A quantum harmonic oscillator expressed as a time-varying wave function immediately suggests to me a wealth of musical potential. I expand on this potential in Chapter 5.

The Allosphere Research Group, led by JoAnn Kuchera-Morin, has done some of the most compelling work to date in sonifying and visualizing quantum processes[14]. Using the 3-story immersive Allosphere system, the Allosphere Research Group turns a hydrogen-like atom into an audiovisual instrument that can be controlled, explored, and performed. Kuchera-Morin has composed an audiovisual piece, *Myrioi* (2020), using this instrument.

1.1 Overview

This chapter serves to introduce the subject matter of this dissertation and to establish the background of the work.

In chapter 2, I discuss the relationship of science and music in general. I also present two of my own pieces, *De Rerum Natura* and *Coacervate*, as case studies in scientific sonification and science-based musical metaphor.

Chapter 3 presents my research into the use of classical Newtonian physics in music. This includes a brief description of classical physics, with a focus on the classical model of coupled harmonic oscillators. I also discuss advances I have made in my software for sonifying such models, CHON.

My research and software based on quantum physics is described in chapter 4. The chapter begins with some background information regarding quantum physics that is needed to understand the foundations of the software and music I have created. The software, QHOSYN, is then described in terms of the physics, the code, the interface, and the audio engine.

Chapter 5 is an account of composing the piece "Psi" using the results of both my classical and quantum mechanical musical experiments. This chapter serves as a case study and a proof of concept that demonstrates the musical potential of the research presented in this document.

Chapter 6 is a reflection, from an aesthetic standpoint, on the musical results of my work composing using classical and quantum models.

Chapter 7 concludes the document and points to future applications and developments of these techniques that have yet to be explored.

Chapter 2

Science and Music

> *From here on nothing prevents us from foreseeing a new relationship between the arts and sciences, especially between the arts and mathematics: where the arts would consciously "set" problems which mathematics would then be obliged to solve through the invention of new theories.* — Iannis Xenakis[15]

Music and Science form a reciprocal, symbiotic relationship. Musicians and scientists inspire and learn from one another and the musician-scientist stands to gain the most from both worlds. Many have previously pointed out the strong links between mathematics, science, and music[16][17]. As Einstein said[18]:

> The theory of relativity occurred to me by intuition, and music is the driving force behind this intuition. My parents had me study the violin from the time I was six. My new discovery is the result of musical perception.

Einstein is one example from a long lineage of musically-minded giants on whose shoulders we stand. Kepler, Galileo, Boltzmann, Planck, Leibniz; these great thinkers understood the human musical instinct as fundamentally linked to scientific instinct[19]. While good science is certainly not done using instinct alone (and indeed has safeguards against

drawing conclusions from such subjective reasoning), it is undeniably a vital part of hypothesis forming, model building, and experimental design. Ultimately, science needs to satisfy pure logical criteria, and music has to satisfy aesthetic criteria, but the methods used to these ends can be closely related.

The composer is first sparked by an idea (a melody, harmony, text, theme, form, etc.) or question (what would it sound like if...? How can this data be sonified? Is it possible to write a compelling duo for crumhorn and theremin?). The spark may be inspired by other music, in which case the composer can adapt and modify an existing compositional system. However, if the spark comes from outside of music and is quite novel, the composer might design a new system from scratch. This can involve creating new computational tools or creating music-theoretical rules for the work to follow.

This process mirrors the way a scientist comes to an idea or question and designs an experiment to test it. The initial spark for an experiment or theory can emerge from a combination of sources. The scientific method involves testing and re-testing hypotheses, which leads to experiments modeled after other experiments, with perhaps only one or two variables changed. Other scientific pursuits require the creation of entirely new theoretical frameworks or new tools and machinery to test them.

The process of designing and running an experiment, and subsequently assessing the results, is subject to scientific rigor. Throughout the compositional process, the musical work must stand up to the rigors of aesthetic judgment. An analysis of the components of this judgment system can be left for another time. Suffice to say that the aesthetic criteria of the composer stand in for the scientist's statistical analysis. Finally, of course, the work is subject to "peer-review" of sorts in a concert.

What follows in his chapter is a look at two of my pieces in which the "spark" or inspiration was drawn from science. The two pieces together show a broad range of possibilities in deriving music from science. The first piece, *De Rerum Natura* was composed largely

using intuitive methods. It does not make extensive use of algorithm or data sonification. Rather, it consists of sonic metaphors at various levels that are based on my interpretation of the concepts at hand. It is also a meditation on the very concept of algorithmic methods vs. intuitive methods. The second piece, *Coacervate*, is an extension of this work and what I consider to be a turning point in my musical practice. In this piece, for the first time, I struck a balance I had been seeking between algorithmic and intuitive methods. Direct data sonification and interpretive techniques are integrated and entwined to weave a network of sonic metaphor.

2.1 De Rerum Natura

In her book *Lost in Math*, Sabine Hossenfelder draws attention to the trap that some physicists fall into of being blinded by beautiful mathematical models[20]. Scientists in general study the universe and pursue objective truth using the scientific method. However, scientists are human and they can have a bias in favor of satisfying aesthetic models with symmetry and simplicity. This bias leads some scientists to pursue untenable, unlikely, and untestable theories in their research. It is not necessarily (though occasionally is) a problem of unsound science, but a problem of spending time and resources (millions of dollars) on pointless endeavors, guided by aesthetic criteria.

One of these criteria is the principle of *naturalness*. In physics, a natural theory ought to have dimensionless ratios of order 1 (e.g. a ratio of $1:3.7$ is more desirable than a ratio of $1:37000$ or $1:0.000037$). Pi ($3.14159\ldots$) is a nice natural ratio while the ratio between the weak nuclear force and the gravitational force ($1.73859(15) \times 10^{33}$) is not[1]. A natural theory should also avoid fine-tuned parameters, which require extreme precision to account for our universe. Essentially, the idea of naturalness boils down to scientists wanting to pursue hypotheses that lead to models with nice numbers that don't

[1] this discrepancy is known as the hierarchy problem in fundamental physics.

seem overly contrived. There have even been attempts to quantitatively calculate the naturalness of a theory [21].

Nobel laureate Frank Wilczek takes a differing stance, citing all of the achievements that scientists have made while following aesthetic ideals[22]. He acknowledges that this can steer scientists wrong, such as in the case of Johannes Kepler's Mysterium Cosmographicum model of the solar system(S2.1🔊). However, he argues that even these missteps are valuable on a global level. The thirst for unveiling beautifully symmetrical, orderly, economical secrets in the universe acts as a libido driving physicists toward new discoveries and theories.

There is a parallel conflict in music composition between algorithmic/formal methods and intuitive/free methods. Scientists pursuing beautiful numbers at the expense of other theories that may fit more closely to reality are akin to composers pursuing beautiful numerical ratios, algorithms, and formalisms in their music without regard for what is ultimately *heard* in the music. On the other hand, composer can just focus on the music as such and miss out on a world of inspirational structures and forms. Of course, in either case, the composer is not pursuing the same goal as a scientist. The bottom line of science is clearly to discover objective ground truths about reality. But what is the bottom line for music? In this analogy, it may seem that I have assumed that the phenomenological result, what a listener perceives, is the ground truth of a piece of music, but this isn't a foregone conclusion. Rather than tackle this charged question, I would like to simply take for granted that there exist at least two perspectives on the source of music's value, namely: what the composer puts into the music, and what a listener extracts from the music.

De Rerum Natura is inspired by the concept of "naturalness" in physics; this conflict between truth and beauty. The tension between this widely applied aesthetic concept on the one hand, and the promise of science to shed all bias in pursuit of truth on the

other, guided my shaping of the piece. The blending and processing of sounds reflects the tension between nature and naturalness; the way things are and the way we want them to be.

I recorded the sound material in Australia (Alpine National Park), Paris, Siena, and California. I composed the piece at Musiques & Recherches in Belgium and at CREATE in California (2019-2020). *De Rerum Natura*, unlike the other pieces discussed in this document, does not feature any data sonification. I recorded the raw sound material and tried to intuitively weave a tapestry out of them. While composing, I was focused on two main ideas that guided the form: creating a musical dialogue around the idea of naturalness and designing physical metaphor into the musical gestures.

It occurred to me that there could be several definitions of naturalness in the context of music. One could take natural in the sense of sounds of nature in opposition to industrial or human-made sounds. I had recorded plenty of examples of both. I had in my sound archive rain, thunder, birds, trees, leaves, branches, footsteps, voices, musical instruments, metal objects, bikes, cars, electromagnetic fields... I created a gradient from natural to unnatural and played with the dynamic between the two. I also used techniques such as convolution and granulation to blur the distinction. This also evokes the sense of naturalness that refers to whether something has been altered or not. As an example, various sounds of rain are heard throughout the piece, and at 3:20, a clip of rain is convolved with a very high metallic sound resulting in an uncertain degree of naturalness(S2.2🔊).

Another sense of naturalness contrasts with synthetic or automatic sounds. For example, the sound of a violin played manually by a human can be described as more natural than a synthesized sawtooth wave. Any sound that is manually performed by a human may gain this natural quality. This contradicts the previously discussed definition of naturalness in which human intervention decreases naturalness. In the sounds I recorded

for *De Rerum Natura*, I was sometimes a passive observer and sometimes an agitator, activating objects in the soundfield with hits, scrapes, bending, and shaking. A stark juxtaposition of this synthetic-physical duality is heard at 5:25. A drone generated from pulsar synthesis dominates until the sound of a branch being broken is heard(S2.3🔊). Manually created sounds have a sense of physicality, whereas automatic or synthetic sounds can feel more abstract.

In exploring degrees and dimensions of naturalness in *De Rerum Natura*, especially in the physical-synthetic dichotomy, I was inspired to try to create tangible gestures even when using abstract material. I found a rich resource of sound material by convolving sounds with recorded sounds to achieve certain sound transformations with a more tactile, physical result. Convolution reverb is an obvious example, but I also used convolution to granulate sounds. Instead of using standard granular synthesis tools, I would convolve the sound with a recorded sound that has an inherent granular texture such as rainfall, snow or ice crunching, fire crackling, or wood creaking and breaking(S2.4🔊).

I manipulated some sounds to design spatial gestures and changes in energy that felt like they had inertia and momentum, but some of the recorded material already had an inherent physicality. One example is the stereo recording I took of a man shaking a box filled with some small objects at the Stravinsky fountain in Paris. This recording is a prominent motif of the piece. It has very distinct granular, oscillatory, and spatial characteristics that make it feel very tangible. I used it in various ways from inserting it unchanged, to convolving it with other sounds (imposing its spatial and temporal shape onto the sound), and transforming the original sound with time stretching, pitch shifting, and spectral manipulation (S2.5🔊).

With all of this I created gestural sequences that evoke physical metaphors such as can be heard in the first minute of the piece. First, a metallic sound crashes onto the soundfield, throwing up a flurry of shrapnel that slowly settles onto a calm glassy surface. A new

sound then swirls into the scene and kicks up the dust again. Emerging from the debris, a fluttering sound seems to strike a match that heats up another sound that builds up to an explosion. The fallout of the explosion is swept away unexpectedly by another clattering sound which sets up a domino effect leading to a door opening into a noisy soundscape(S2.6🔊).

2.2 Coacervate

Certain mixtures of polyelectrolytes can spontaneously form dense liquid droplets (dense phase) suspended in water (dilute phase). These liquid droplets, called *coacervates*, are often filled with complex molecules, proteins, polymers, and nucleic acids. Coacervation occurs when polyanion and polycation polymers coalesce into a dense phase within a dilute solution (Like liquid bubbles suspended in liquid). Coacervate formation has been suggested as a possible mechanism through which the first simple cells formed on earth[23]. I worked closely with violinist and chemical engineer Chelsea Edwards to create a sonic narrative from this chemistry.

In Coacervate, I used a variety of techniques to encode the chemistry of coacervation into music. I used some of my original software applications in the process including Prism-Dilate[24] and NMR-spectroscope. I also made extensive use of EmissionControl2[25], a granulation software created by Jack Kilgore, Curtis Roads, and myself.

The sound material consists of audio recordings from a chemistry lab, recordings of the violin, and synthetic sounds based on chemical structure. One of the methods of synthesis involved using Nuclear Magnetic Resonance Spectroscopy data to sonify the spectral characteristics of the polyelectrolytes. NMR data encodes the structure of the complex molecules by recording the resonant frequencies of the constituent carbon or hydrogen atoms[26]. It achieves this by inducing state changes in the quantum spin of elementary particles. This method is the state of the art in studying the structure of molecules. It is

a valuable tool for a scientist and it is a source of interesting spectra and sound material for a composer (See Figure 2.1 for spectral peak values and S2.7 for audio examples).

a) Adenosine Triphosphate 13C NMR

Frequency	Normalized Amplitude
15930.248	0.486
15648.896	0.865
15272.675	0.325
14353.066	0.867
12208.41	0.295
8999.529	0.826
8721.06	0.625
8712.92	0.582
7746.125	1
7342.039	0.991
6819.692	0.704

b) Adenosine Triphosphate 1H NMR

Frequency	Normalized Amplitude
2514.21	0.0613
2517.29	0.0812
2518.63	0.0879
2521.76	0.0816
2525.93	0.1011
2529.03	0.1199
2530.4	0.1278
2533.53	0.097
2560.7	0.0873
2563.52	0.1065
2567.17	0.093
2570.05	0.1053
2572.22	0.08
2575.24	0.0662
2578.89	0.0568
2581.77	0.0548
2631.19	0.13
2634.07	0.1757
2636.67	0.137
2701.57	0.0035
2763.42	0.0089
2768.68	0.0118
2774.2	0.0058
3669.62	0.4931
3675.43	0.492
4938.43	0.976
5110.3	1

c) Allylamine Hydrochloride 13C NMR

Frequency	Normalized Amplitude
4517.501	0.88
11389.334	0.781
14136.056	0.99

d) Allylamine Hydrochloride 1H NMR

Frequency	Normalized Amplitude
548	1.0
760	0.4
1164	0.9

Figure 2.1: Frequency and amplitude values for ATP and PAH using both 13C and 1H NMR techniques. ATP is a much more complex polyelectrolyte.

NMR spectroscopy produces values representing the chemical shift of atoms (the difference in resonant frequency based on electron cloud density) in units of ppm (parts per million), notated as δ. An NMR reading will return many δ in the form of peaks on a frequency-amplitude graph (see Figure 2.2)[26]. Each δ value reveals how the the resonant frequency of an atom in a molecule differs from its characteristic resonant frequency in a reference molecular standard (often the resonance of tetramethylsilane), thus revealing information about its chemical bonds. These values in ppm are determined by:

$$\delta = (v - v_r)/v_o$$

where v is the measured resonance signal, v_r is the reference resonance of the molecular standard, and v_o is the frequency of the driving magnetic field. For sonification purposes, the resonant frequency in Hertz is v, and the spectrum can be transposed down several octaves to put it within audible range. Another sonification method might take the chemical shift $(v - v_r)$ as the spectrum, in which the frequencies are already in an audible range.

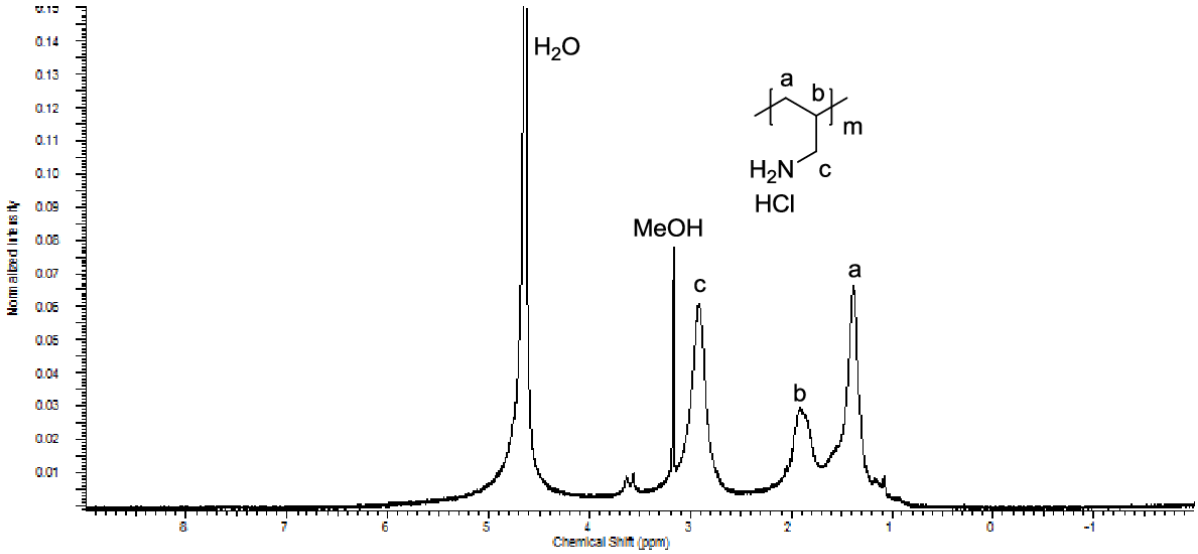

Figure 2.10. 400MHz ^{1}H NMR spectrum of PAA-HCl in D_2O.

Figure 2.2: An NMR reading of Polyallylamine Hydrochloride[27]

This new method of the sonification of molecular structure is a very direct translation from science to music as the molecules literally resonate like an instrument.

The field recordings from the chemistry lab captured pipettes, glass vials, vortex mixers, and other machinery. Unfortunately I was not able to enter the lab due to COVID-19 restrictions, so I had to ask Chelsea to take (noisy) recordings on her phone. The artifacts from noise reduction are evident, but actually imbued a desirable shimmering timbral quality into the sounds (S2.8🔊).

Some motifs for the the violin were constructed by first translating the constituent molecules of the mixture to pitch-classes according to the atomic number of their atoms

and then reading through the molecular structure bond-by-bond. For example, H_2O became G-D-G, a prominent motif throughout the piece because water is the the basis into which the other components are added. $MgCl$ (C-Dflat) is added to create a saline solution, before the much more complex polyelectrolytes are added. For much of the piece, the sounds in this piece tend to float through a diffuse musical texture, in order to evoke the dilute phase of the solution(S2.9🔊).

This diffuse soundscape is occasionally disturbed by a new component being added to the solution/soundscape triggering sudden bonding and coalescing. When the polycation Polyallylamine Hydrochloride (PAH) is added, the coacervation reaction occurs and the music becomes very active[23]. PAH and ATP coalesce into coacervate droplets, represented by melodic chains and dense harmonies formed from the constituent pitch-classes (S2.10🔊).

Chapter 3

Classical Physics

Classical physics encompasses a range of theories that emerged before the quantum revolution of the 20th century, including classical mechanics (Newtonian, Lagrangian, and Hamiltonian), classical electrodynamics, and classical thermodynamics. Classical mechanics describes the deterministic motions of objects such as a baseball struck by a bat, a raindrop falling from the sky, or a planet orbiting a star. This framework is largely founded on Newton's laws of motion[1]. Classical electrodynamics extends Newton's laws to explain electrical charges, magnetism, and currents. Classical thermodynamics describes the transference of energy, heat, and work and is suited to describe a steam engine or a chemical reaction. All of these fields were later extended and supplemented in modern physics when physicists encountered limits to their descriptive power[28].

Classical physics describes much of the macroscopic world that we are able to directly experience. At our human scale (larger than atoms), mass (less massive than black holes), and speeds (much slower than the speed of light), classical physics provides a very useful framework for understanding and predicting the movement and interactions of objects. At

[1]Newton's Laws can be stated as follows:
1) A body's velocity remains constant unless acted upon by an external force.
2) The force exerted on a body is the product of mass times acceleration.
3) A body will exert an equal and opposite force to any force exerted upon it.

the limits of classical physics lies the transition into relativistic and quantum mechanics. For example, to describe the behavior of very small objects, at the scale of atoms and sub-atomic particles, quantum physics is needed; and for objects approaching the speed of light, special relativity explains the phenomena that classical physics cannot. We rarely experience these "extra-classical" physics directly and consciously, so our faculties of sense, perception, and reasoning are ill-equipped to fully grasp them. Classical mechanics, conversely, is much easier to understand and is even largely intuitive (see chapter 6 for an expansion of this point).

The physics of sound and of musical parameters such as harmony, scale, and timbre have been studied for millennia. The Pythagorean school studied the ratios of scale and harmony, and by Plato's time harmony was considered a branch of physics[29]. Physics has entered into the musical thinking of 20th century composers such as Edgard Varèse and Györgi Ligeti. Varèse described his conception of music using physical language such as "sound-mass," "attraction-repulsion," and "shifting planes"[30]. Ligeti used a literal physical system (100 metronomes) in his Poéme Symphonique [31].

The metronomes in Ligeti's Poéme Symphonique act as independent oscillators. They each progress at their own rate, ticking at a regular frequency. This is an analog version of the Low Frequency Oscillators (LFOs) and modulators that are regularly used by composers of electronic music. Periodic oscillators are natural and they are of fundamental importance in physics. However, an infinite sine wave (or other waveform) that doesn't decay or vary in time is an abstract concept. Real oscillators exist in networks of causality and influence, they are acted upon and they act on other objects. This is why the model of coupled harmonic oscillators is of such profound interest to scientists.

Coupled oscillators (in the form of mass-spring models) also play a central role in the field of physical modeling. Physical modeling synthesis is largely focused on re-synthesizing believable facsimilies of real instruments[32][33]. It is also less commonly used for

creating fantastical virtual instruments of unreal proportions or configurations[34] [35]. Claude Cadoz and others at the Association pour la Création et la Recherche sur les Outils d'Expression) in Grenoble pioneered the use of the mass-spring model in physical modeling[11].

A basic coupled harmonic oscillation system consists of two masses attached with springs, such as the pendulums in Figure 3.1. In such a system, energy is passed back and forth between oscillators and complex motion can arise from the superposition of modes, especially as the number of oscillators in the system increases.

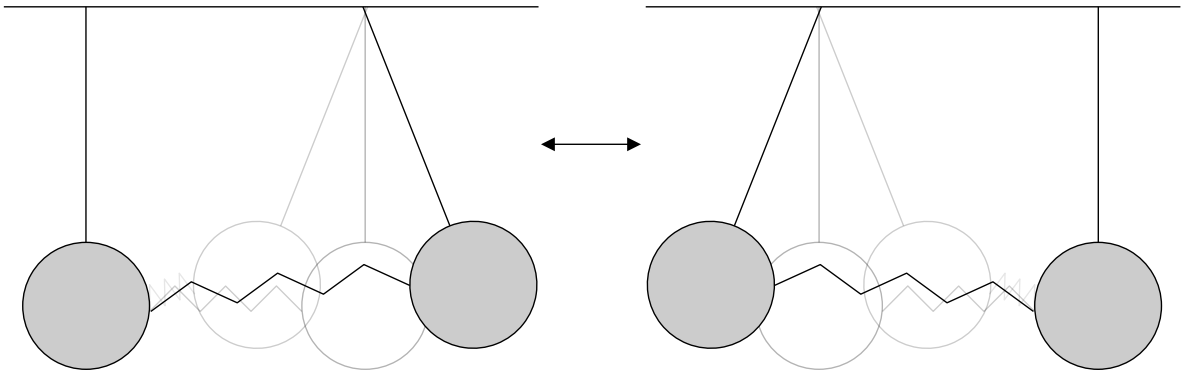

Figure 3.1: Coupled Pendulum - the amplitude of one pendulum's oscillation increases while the other decreases. Then, the process reverses.

Though this kind of system is well-known in physical modeling, the perspective has most often been focused on how the sum total of the system simulates an object such as a string. It had never before been applied in composition by using the individual oscillators as voices in their own right to create, for example, contrapuntal structures and cascading antiphonic forms. I am speaking of using a coupled oscillator system as a new kind of LFO, or an LFO network, that is tangible, physical, and causal. With an LFO system such as this, a composer can drive synthesis parameters or musical score generation with a modulation scheme grounded in real phenomena. It was my interest in this application of coupled harmonic oscillators that led me to create CHON[36].

3.1 CHON

Coupled Harmonic Oscillator Network (CHON) is a real-time interactive application for generating musical material and control signals by using a simulation of coupled harmonic oscillators as an interface. I have already documented this software extensively in my masters thesis[9], but I will briefly expand on what I wrote there and note some advancements I have made with the software.

CHON is used by directly clicking and dragging oscillators on screen. This intervention sets the coupled oscillator system into a chain reaction. The user can then let the system evolve on its own, change parameters of the simulation such as the stiffness of springs or the damping coefficient, or click an oscillator to interrupt the flow of the system directly.

The balance of the precise simulation of a physical system against the direct user interaction makes CHON a hybrid algorithmic-heuristic tool, allowing intuitive intervention in an otherwise rigidly deterministic system. The musical gestures that one can create with CHON range from trite (a sine wave) to chaotic (dozens of mutually dependent oscillators), but they seem nonetheless tangible because of their connection to the physical world.

The motion of each oscillator can drive synthesis parameters within CHON or be sent out via OSC to control other software or hardware. CHON can be used for non-musical purposes as well. I have heard from an artist who used CHON to control parameters in a visualization. In the newly released version 1.3[36], CHON can send more OSC data, reporting the number of oscillators in the system and their absolute positions in space as well as their relative positions (relative to their equilibrium point).

In addition to the new OSC messages for external synthesis control, the internal synthesis engine of CHON has received an overhaul. There is now a stereo mode, which pans each oscillator's synthesis from left to right according to their position in the simulation. The tuning system, which sets the pitch of each oscillator, has been greatly expanded. The

user can now select one of five scales along with a root pitch, and the entire CHON system will be re-tuned accordingly. With this new update, the Additive Synth engine can also be re-tuned to the scale(S3.1🔊).

Beyond these global controls, the amount of control over each individual oscillator's tuning and volume has been improved by adding the ability to right-click on an oscillator to edit its parameters directly (see Figure 3.2). An oscillator's tuning can be set by inputting a frequency in Hz or a scale step-number, to quantize it to the currently selected scale.

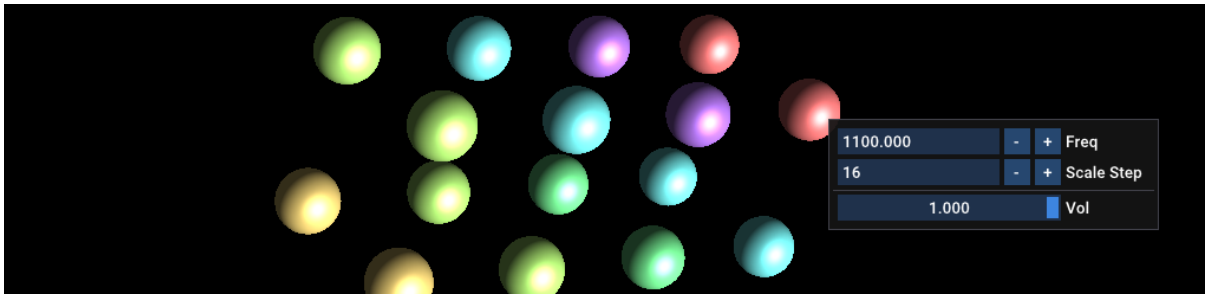

Figure 3.2: The new right-click menu, allowing the user to tweak each individual oscillator's parameters.

To increase the usefulness of the internal synthesis engine of CHON, an audio recording module has been added. With the audio recorder, the user can turn on recording and perform on CHON. When they are finished, the recorded file can be saved by stopping the audio recorder. The audio input and output of CHON can also be configured with the new Audio IO Settings module.

Another completely new feature is audio input driving. CHON could previously simulate a simple harmonic oscillator, damped oscillators, and coupled oscillators. Now, with this new feature, CHON can also simulate another fundamental oscillation model in physics: the driven oscillator. A driven oscillator can be thought of as a simple harmonic oscillator that receives an external driving force. A driving force can also be added to a coupled oscillator system, one need only decide which oscillators in the system are driven and the force will propagate throughout the network.

With audio input driving, the driving force acting on CHON is an audio signal from a microphone or another application. Using the Audio IO Settings module, the user can select an audio input source and CHON will begin listening. The Input module, pictured in Figure 3.3, allows the user to configure the input driving parameters.

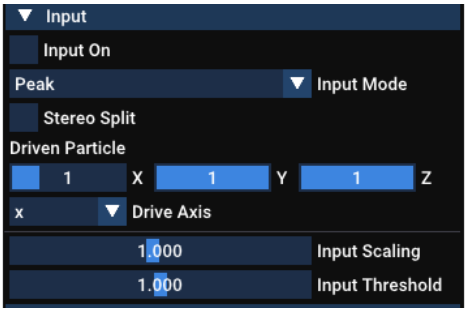

Figure 3.3: The CHON input driving module

"Input On" toggles the input driving feature on/off. The "Input Mode" drop-down menu allows the user to select between three modes: Peak, RMS, and Frequency. In Peak mode, the instantaneous sample value CHON reads from the input signal is the amount of force applied to the oscillator. In RMS mode, CHON averages the signal magnitude over a user-configurable number of samples, so that the driving force is smoothed. In Frequency mode, CHON performs Short-Time Fourier Transform (STFT) analysis on the input signal, driving a different oscillator with each frequency bin. In Peak and RMS mode, the user can select which oscillator out of all active oscillators in CHON will recieve the driving force using the "Driven Particle" parameter. The "Stereo Split" parameter can be turned on to drive one oscillator with the left channel of the audio input and another oscillator with the right channel. The "Drive Axis" parameter determines the direction (X, Y, or Z) in which the driving force will be applied. The user can scale the amplitude of the input signal with the "Input Scaling" parameter and set a minimum amplitude threshold for CHON to register the signal using the "Input Threshold" parameter.

With this new feature, CHON can respond to audio input and be set in motion with sonic energy. This motion could then activate the internal synthesis engine of CHON (S3.2🔊).

Alternatively, the data of the motion of the oscillators can be sent out via OSC to another application or synthesizer. One can set up CHON for music with a live performer in which the sound of the performer drives CHON in sympathetic resonance. This could lead to performance network topologies such as a feedback loop where CHON generates control data that then generates sound that drives CHON, or a duet where one or more performers drive CHON with the sound of their instruments.

CHON can simulate coupled oscillators in a variety of spatial configurations. Up to 100 particles can be arranged in rows and columns of whatever proportions the user wishes. In version 1.3, the oscillators can now be arranged in a 3-dimensional cube in addition to the previously allowed configurations of a 1-dimensional string and a 2-dimensional plane (see Figure 3.4).

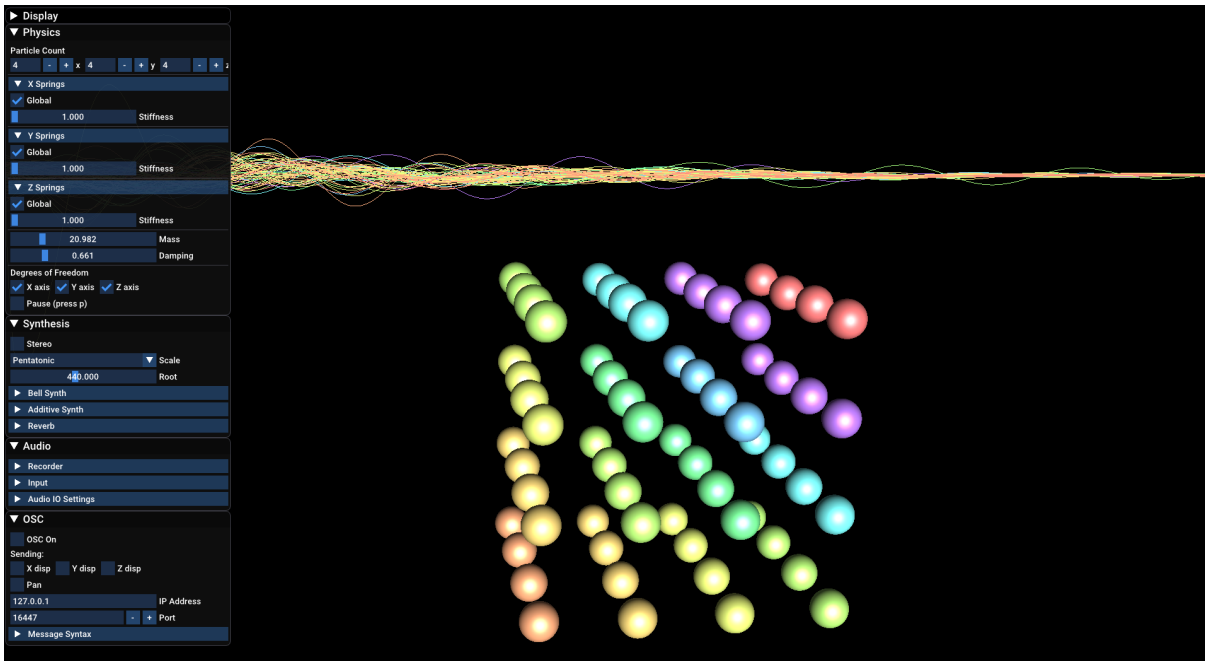

Figure 3.4: A screenshot of CHON in a 3-dimensional oscillator network configuration.

Chapter 4

Quantum Physics

Beyond the classical physics that governs the macroscopic world we are familiar and comfortable with, reality can become very strange and unintuitive. The laws of classical physics cease to be applicable at scales beneath the size of an atom. At very small sizes, the fundamental particles that make up the universe behave in very non-classical ways. Determinism and causality seem to break down, particles behave like waves, objects can travel through barriers, and the states of very distant objects can be intertwined through entanglement.

The formulae and frameworks of quantum physics have been extraordinarily successful in application, leading to countless advancements in technology and material science [37]. On the other hand, the philosophy of quantum physics has proven to be extraordinarily befuddling despite many great minds taking up the task of untangling it. Nevertheless, because the results of quantum experiments and theories lead to surprising and seemingly paradoxical consequences, many people tend to be drawn to solve the puzzle of what quantum theories really *mean*.

Frameworks for understanding the consequences of quantum physics intuitively are commonly called interpretations. The most widely taught interpretation has been the

Copenhagen interpretation[38]. We can add to this the many-worlds interpretation, quantum information theory, de Broglie–Bohm theory or pilot wave theory, and many others. These interpretations grapple with such philosophical quandaries as determinism, the ontology of the wavefunction, observer agency, and locality.

The interpretations of quantum physics should not be confused with quantum physics itself, however. While each of these interpretations have certain weaknesses and make wild claims about the nature of reality, quantum physics itself, the mathematics and the theoretical frameworks that physicists use on a regular basis, are essentially airtight. Interpretations of quantum physics have led to to science fiction like Schrodinger's undead cat, parallel universes, and human consciousness shaping reality. Quantum physics itself, on the other hand, has brought us lasers, transistors, and modern medical scanning. This is why physics teachers often choose to ignore these interpretive issues and instead teach the so-called "shut up and calculate" (instrumentalist) interpretation: the math is consistent with the math, don't worry about if it conforms to your intuition about reality[39].

This should not be taken to imply that the philosophy of quantum physics is less important than the physics itself, only that the distinction is important. Interpretations of quantum physics help guide physicists their intuitions, in experimental design, and in hypothesis forming. Furthermore, the philosophical ramifications of quantum physics are important in shaping our understanding of the world. Insofar as philosophy has been integral to the development of society in the past, quantum philosophy can do the same for the future. A physicist may be able to get by with only the mathematical formalisms of quantum theory, but the field is enriched through new perspectives and intuitions. The pure mathematics of quantum physics can be good enough for the physicist, but not for the artist. It is up to the physicist to apply these concepts, it is up to artists and philosophers to play with them.

Thus I aim to use quantum physics as an artist uses perspective, proportion, and color theory: not necessarily to represent the world in perfect reflection, but to play with reality and perception to cultivate intuition and create meaning, narrative, and wonder. In a sense, I wish to postulate my own interpretation of quantum physics: a sonic interpretation. Quantum mechanical effects and all of their strange consequences are ultimately predicated on and borne from the underlying mathematics, so a good interpretation of quantum mechanics must start from the mathematics. I will clarify some of the most salient of these technical points in the following sections.

4.1 Hilbert Space

To understand a quantum system, we may need to analyze a multitude of parameters at once. If these parameters are mutually orthogonal (if they correspond to unique eigenstates), then we can avail ourselves of mathematical tools that allow for calculations in higher dimensional spaces.

A Hilbert space can be used to represent vectors in any number of dimensions. Hilbert spaces are not real physical spaces (though they can represent and be used to analyze real space), they are simply abstract mathematical spaces with convenient and useful properties. Among these advantages is that they allow for the types of math used in Euclidean space to be extended into any number of dimensions. In fact, Euclidean space is itself an example of a 3-dimensional Hilbert space.

Imagine an object with 3 properties: sweetness, bitterness, sourness. We could represent these attributes in a 3-dimensional grid with the x-axis, y-axis, and z-axis corresponding to sweetness, bitterness, and sourness, respectively. Let us say the object has a sweetness of 5, a bitterness of 2, and a sourness of 8. We can then imagine a vector in our taste-space with coordinates (5, 2, 8), meaning that to arrive at this coordinate, we travel 5 on the x-axis, 2 on the y-axis, and 8 on the z-axis (figure 4.1). Now that we have arranged the

data so, we gain access to certain analytical abilities. For example, we could measure something we might call "flavorfulness" by calculating the length (magnitude) of the vector using basic geometrical rules (with thanks to Pythagoras). The further the vector extends from the origin (0,0,0), the more flavorful it is. One could come up with other ways to analyze the data, such as taking the angle of the vector to indicate the complexity of the taste or representing a multi-course meal as several vectors forming a polygon.

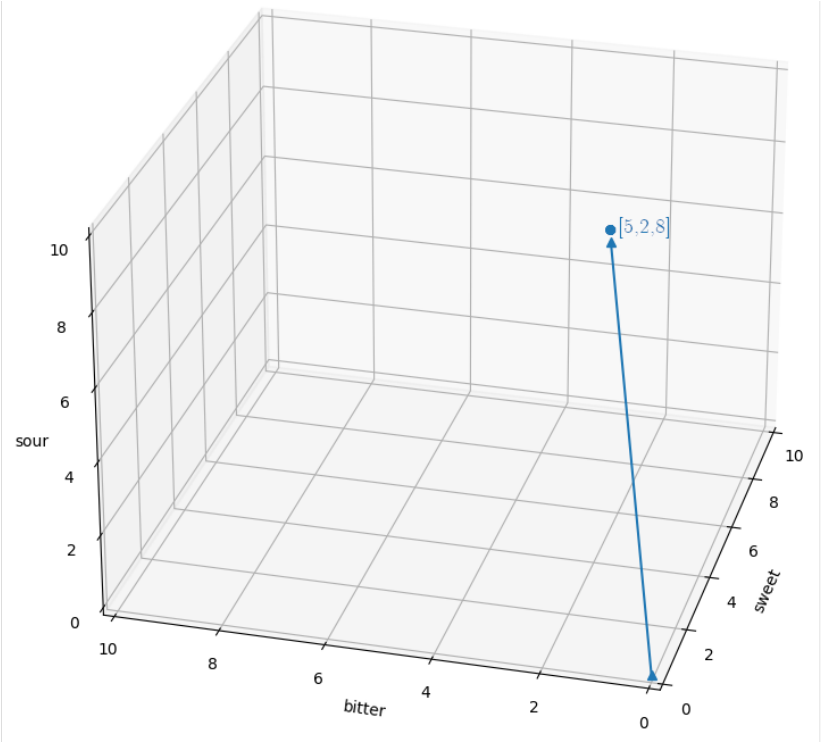

Figure 4.1: An example of a 3D Hilbert space representing taste with a vector [5,2,8]. The magnitude of this vector can give us a quantitative measure of "flavorfulness".

This 3D space is fairly simple to understand, but what if we had more types of flavor? If we add a 4th flavor - savoriness - then we need to have a 4-dimensional space. This is where higher-dimensional Hilbert space becomes useful. We can now represent the flavor of the object as a point in 4D space, say (5, 2, 8, 1), or, 5 on the sweet axis, 2 on the bitter axis, 8 on the sour axis, and 1 on the savory axis. This 4D space might be hard to draw or visualize, but the important thing is that we can perform the same kind of geometrical mathematics in this higher-dimensional space as we can in 3D space (such as

finding the magnitude of a 4D vector).

In music, multi-dimensional geometric thinking has been applied by composers such as Iannis Xenakis, James Tenney, and Clarence Barlow. Xenakis, with his architectural education, used geometric thinking extensively in his work, including higher-dimensional spaces such as in the composition of Herma [4]. Tenney's harmonic spaces [40] are a kind of Hilbert space, though he never explicitly used the term to describe his methods. Barlow's harmonicity and multi-dimensional scaling techniques approach Hilbert space theory [41].

Hilbert spaces are related to music in other ways as well. The Fourier transform, one of the foundational tools of sonic analysis, processing, and synthesis, is an isomorphism of Hilbert space with each term in the Fourier series representing one of the dimensions of the space. The modes of a string or drumhead can also be analyzed using Hilbert space methods[42].

In quantum physics, the possible states of a system make up a state space. We call the possible measurable states *eigenstates* to differentiate them from probabilistic superposed states[1]. This space of eigenstates is a complex-valued Hilbert space that describes the probability of the system to be in each of these eigenstates. Or, looking at it another way, each possible eigenstate of the system can be seen as an orthogonal vector in an abstract Hilbert space. We can then derive a wavefunction of the system from a projection of the Hilbert space into two or three dimensions.

4.2 Wavefunction

A particle is a special kind of quantum thing that is at once like a wave and like an object with a shape, size, and mass. It is a common misconception that particles exhibit

[1]Eigenstates refer to definite and measurable values of the variables of a particle, like a position or momentum. Eigenstates are relative. A particle does not have an eigenstate, but it can have eigenstates of position or eigenstates of momentum.

a wave-particle duality in which the particle is sometimes a particle and sometimes a wave. It is not accurate to say that it is sometimes one and sometimes the other, as some explanations seem to suggest. It is always a particle and a particle has wave-like properties[43].

Louis de Broglie was the first to propose the wave-like nature of particles in his PhD thesis[44]. It has been confirmed experimentally that particles can be diffracted like a wave[45] and can interfere constructively and destructively[46]. This leads to the idea of superposition, which is a confounding subject to many people, but shouldn't be foreign to musicians.

The de Broglie relations give us a bridge between the wave-like and particle-like properties of matter in the same way as Einstein's $e = mc^2$ gives us a bridge between matter and energy. They state:

$$\lambda = \frac{h}{p}$$

$$f = \frac{E}{h}$$

where λ is the wavelength of the particle, h is the Planck constant, p is the momentum of the particle, f is the frequency of the particle, and E the total energy. This has been foundational to quantum theory as Schrödinger built upon it with his wave mechanics model in a higher-dimensional space.

So, it is important to keep in mind that the "particle" in quantum physics is not akin to the macroscopic objects we might visualize when we think of a particle. It is not like a particle of sand or a dust particle. It is something far more alien to our mind's eye and it is distinctly "quantum" in nature. Throughout this text, when I use the word *particle*, I

am referring to this quantum thing.

A particle can be mathematically represented by a wavefunction that is a superposition of its eigenstates. This wavefunction is complex-valued and the square of the absolute value of the wave gives us the probability distribution describing the possible states of the particle. The wavefunction is commonly referred to by the Greek letter psi (Ψ). The wavefunction corresponds to a vector in our system's Hilbert space.

4.3 Decoherence

When a particle interacts with its environment (i.e. with other particles or fields) it gradually loses its quantum properties and begins acting more classical. This is called *decoherence*. In the case of a quantum harmonic oscillator, this could mean, for example, that the system begins to behave more like a classical harmonic oscillator with a definite position and momentum, losing its probabilistic, wave-like properties.

A measurement is one such interaction that causes decoherence. When measured, the wavefunction collapses, meaning that the probability of the eigenstate we found is 1, while the probability of all other eigenstates is 0. This seems obvious, as though we are simply updating our knowledge when we learn more information. However, the collapse is not just a reduction of uncertainty, rather it alters the very nature of the state of the particle. If we measure the position of the particle, the position of the particle turns from something spread out and wave-like, into something more point-like. This affects other properties of the particle as well. Position and momentum are complementary variables, so the Heisenberg uncertainty principle states that if position is measured, the momentum of the particle becomes more uncertain, and if the momentum is measured, the position becomes more uncertain. So, while the position collapses to a point, the momentum probability distribution spreads out like a wave.

After position is measured and the wavefunction is collapsed, the particle's position starts to gradually smear out. This smearing out is called *dispersion*. Intuitively, we can understand this as being due to the fact that we do not know what the momentum was when we measured the position. The particle could be in a number of locations after we measure it, depending on its momentum.

In my research and simulations, I have chosen to ignore the decoherence effect. It was a conscious choice made primarily due to the fact that what I aim to sonify here are the quantum effects of a particle in superpositions of energy states. This is a somewhat idealized system, but it does not create a completely unrealistic picture of the quantum situation. Researchers working on quantum computing, for example, spend considerable effort to keep particles in coherent states in order to preserve the quantum effects that are important to their work. Similarly, I am interested in the quantum properties of the quantum harmonic oscillator, so I have created systems in which decoherence is not a factor.

4.4 QHOSYN

QHOSYN (Quantum Harmonic Oscillator Synthesizer) was created to simulate a quantum harmonic oscillator for the purpose of exploring the possibilities of sonifying such a system. Like its counterpart in classical mechanics, the quantum harmonic oscillator is a fundamental model in quantum mechanics as it forms the basis of many systems[47]. Unlike the classical harmonic oscillator, the quantum harmonic oscillator describes a time-varying wavefunction, rather than a classical object. The ground state of this wave, like all quantum systems, is greater than zero. In other words, it cannot be totally static and must always be in some state of fluctuation.

A quantum harmonic oscillator can be described by its eigenstates, or possible energy levels. The wavefunction of the quantum harmonic oscillator turns out to be a linear

combination (or superposition) of these eigenstates. The wavefunction is complex-valued and rotates in the imaginary plane (phase increments). Each eigenstate, then, can interfere based on their phase components in a superposed state. This creates a time-dependent morphing wave and it was this evolving wave pattern that inspired me to create QHOSYN.

QHOSYN can simulate the quantum harmonic oscillator in a superposition of up to 15 eigenstates. It is both a simulation and a tool for generating quantum-based sound. I will describe the software in detail here and discuss some its applications in the following chapter.

4.4.1 Physics

To begin with the physical simulation, we first calculate the eigenbasis of the quantum harmonic oscillator. This will simplify the process of calculating new wavefunctions, as we need only multiply the wavefunction with each basis vector to obtain a new quantum harmonic oscillator.

The eigenbasis forming our Hilbert space is given by:

$$\psi(x,n) = \frac{1}{\sqrt{2^n n!}} \left(\frac{m\omega}{\pi\hbar}\right)^{1/4} e^{-\frac{m\omega x^2}{2\hbar}} H_n\left(\sqrt{\frac{m\omega}{\hbar}}x\right)$$

where n (an integer) is a unique eigenstate of the system and corresponds to the orthogonal axes in our Hilbert space, x is a position, m is the mass of the oscillator, ω is the angular frequency of the oscillator, and H_n are the physicist's Hermite polynomials, which orthogonalize our eigenstates:

$$H_n(x) = (-1)^n e^{x^2} \frac{d^n}{dx^n} e^{-x^2}$$

Using nondimensionalization, we can normalize the units and simplify the problem. We

can factor out the reduced Planck constant $\hbar$ as well as the mass and angular frequency of the oscillator, resulting in the simplified equation:

$$\psi(x,n) = \frac{\pi^{1/4}}{\sqrt{2^n n!}} \cdot e^{-(x^2/2)} \cdot H_n$$

This is the form that QHOSYN uses to determine the basis. QHOSYN calculates a 15-dimensional Hilbert space as the eigenbasis for our wavefunction, since we plan to allow up to the first 15 eigenstates to be simulated. This Hilbert space is calculated using the formula described above.

The `WaveFunction` class takes as arguments in its constructor an object of class `HilbertSpace` and an std::function that takes a double precision float and returns a double precision float. In its constructor, the `WaveFunction` then calculates an orthogonal basis projection from this `HilbertSpace`.

The wavefunction $\langle\psi|$ given by the user is projected onto the orthogonal vectors $|\phi\rangle$ of the Hilbert space. This is entails a matrix multiplication for each energy eigenstate $|\phi\rangle$ that we wish to simulate:

$$\langle\psi_x|\phi_x\rangle = \psi_0\phi_0 + \psi_1\phi_1 + \psi_2\phi_2 \cdots \psi_n\phi_n$$

The `orthogonalBasisProjection()` method creates a lambda function for each dimension of the Hilbert space. This lambda function takes a float value, samples the Hilbert space in that basis, gets the complex conjugate of the sampled result, and returns the magnitude of that conjugate multiplied by a sample of the wavefunction. In pseudo-code this looks like:

```
ValueAtX = conjugate(hilbertSpace[dimension][x]) * waveFunction[x]
```

This function is then integrated over the range of -10 to 10 in each basis to find the

coefficient for each eigenstate. A normalization factor is then calculated to bring the resulting wavefunction into a usable amplitude range. The resulting wavefunction is a complex valued projection of the superimposed eigenstates. Specifically it is a discretized version of what would be a continuous function, in order to be computable. The spatial resolution of this discretization is 256. This number was chosen as it is a reasonable length for a wavetable synthesis basis waveform and it is lean enough to enable fast calculations and sending via OSC.

Once all of the above is finished, the wavefunction is now stored in memory. Every step of the simulation needs only to lookup the value for each position along the x axis at time step t in each orthogonal vector and add the results:

$$\langle \psi(x,t)| = \sum_{n=0}^{N} c_n \phi_{n,x} t \, (0.5+n) \, i$$

Time t is multiplied by the eigenstate number which is offset by 0.5 because according to Heisenberg's uncertainty principle it cannot be zero. This gives us the imaginary part of our complex number. The value of x at time t for each $|\phi\rangle$ is multiplied by the coefficient c_n for that vector derived from the orthogonal basis projection or given manually by the user.

4.4.2 Code structure

QHOSYN was coded in C++. It was designed as an audiovisual, cross-platform, standalone application for Linux, MacOS, and Windows. The primary concerns in designing the current version of the software were to make it efficient enough to simulate and sonify a quantum wavefunction and to make it a usable sandbox for me to explore the possibilities of this sonification. General user-friendly design was not a primary concern in this iteration, but a more approachable and streamlined version of QHOSYN is currently in production.

QHOSYN is supported largely by by the Allolib library[48], which is powered by various other C++ libraries. Allolib is a C++ library developed by the Allosphere Research Group at UCSB. It provides a framework for cross-platform development of interactive multimedia applications and tools. QHOSYN takes advantage of Allolib's convenient wrapping of low-level functions and libraries, its simple structure for initializing an application and running audio and video threads (allowing for communication between threads), and its cross-platform libraries that make it simple to compile builds for Linux, MacOS, and Windows.

The FFTW library was used for its very efficient and hardware accelerated IFFT algorithm[49]. The RtAudio library handles the audio input and output streams for QHOSYN[50]. RtAudio provides an API for cross-platform, real-time audio input and output. Window creation and graphics API are provided by OpenGL and GLFW[51]. The GUI is provided by the ImGui (Dear ImGui) library by Omar Cornut[52].

The structure of the project separates the code of QHOSYN into five files: The primary cpp file "QHOSYN.cpp" and its header file "QHOSYN.hpp," a header file containing various helper functions called "utility.hpp," a header file called "shadercode.hpp" containing the fragment shader for the visual display of the wavefunction, and "wavefunction.hpp," which contains the classes and functions relevant to the calculation of the wavefunction. The code of QHOSYN is open source. It is included in this document as Appendix A and can also be examined online at https://github.com/rodneydup/QHOSYN

4.4.3 User Interface

The visual interface of QHOSYN is comprised of the visualization of the quantum harmonic oscillator and several control panels that can be used to configure the simulation, the visualization and sonification, and manage data flow (see figure 4.2). The visual interface updates at 60 frames per second. User input is accepted via mouse and keyboard.

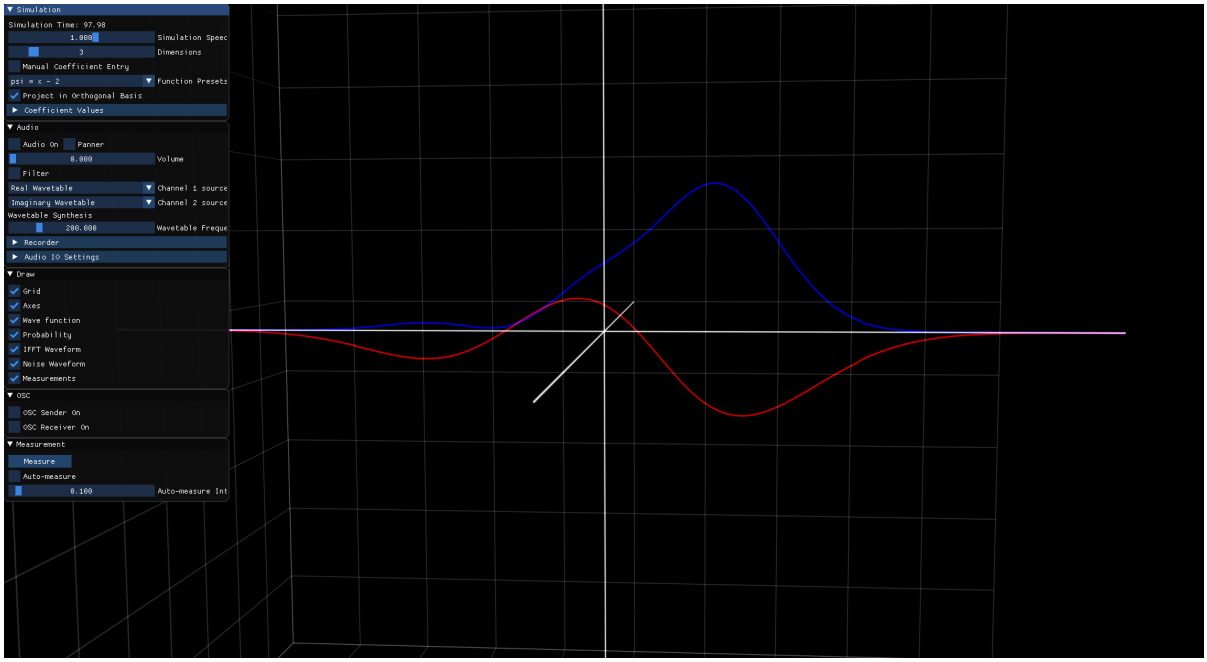

Figure 4.2: The visual interface of QHOSYN.

The visualization of the quantum harmonic oscillator is the dominant element of the interface. The wavefunction itself is represented by a red line. The probability distribution calculated from the wavefunction is a blue line. The three axes (x, y, z) are white lines and a grid is drawn in dark gray lines. If the probability noise band or inverse Fourier transform modes are activated, then the waveform of those sonifications appear in yellow and cyan, respectively. Any of these visualizations can be activated or deactivated in the "Draw" control panel(figure 4.3). This allows the user to customize the visualization according to their preference.

The wavefunction evolves over time, its phase rotating and, if the wavefunction is not in a stationary state, its shape changing. The speed fo the simulation can be adjusted in the "Simulation" panel(figure 4.4). The speed can be set between -5 and 5 and is a multiplier on the simulation speed. A speed of 0 will freeze the simulation.

In the simulation panel, the user can also change the number of eigenstates being simulated (between 1 and 15). The wavefunction can be changed from the "Function

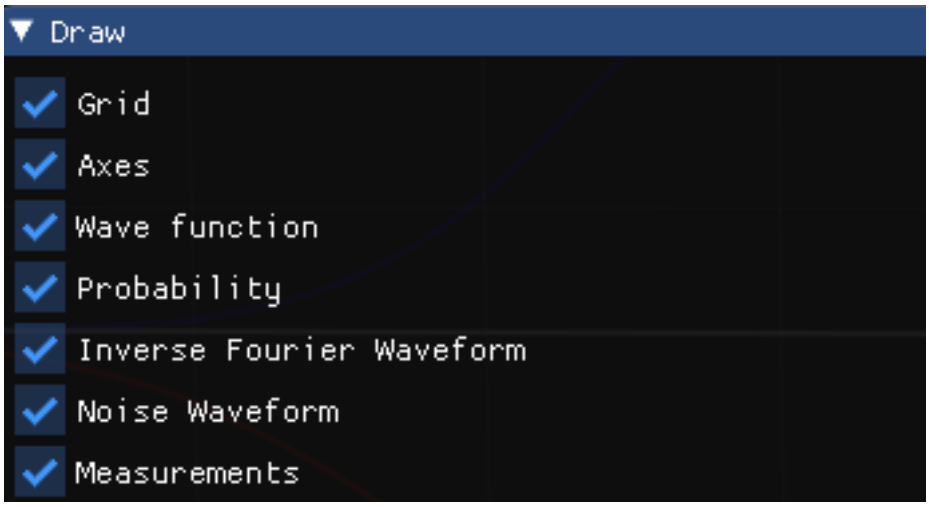

Figure 4.3: The Draw Panel.

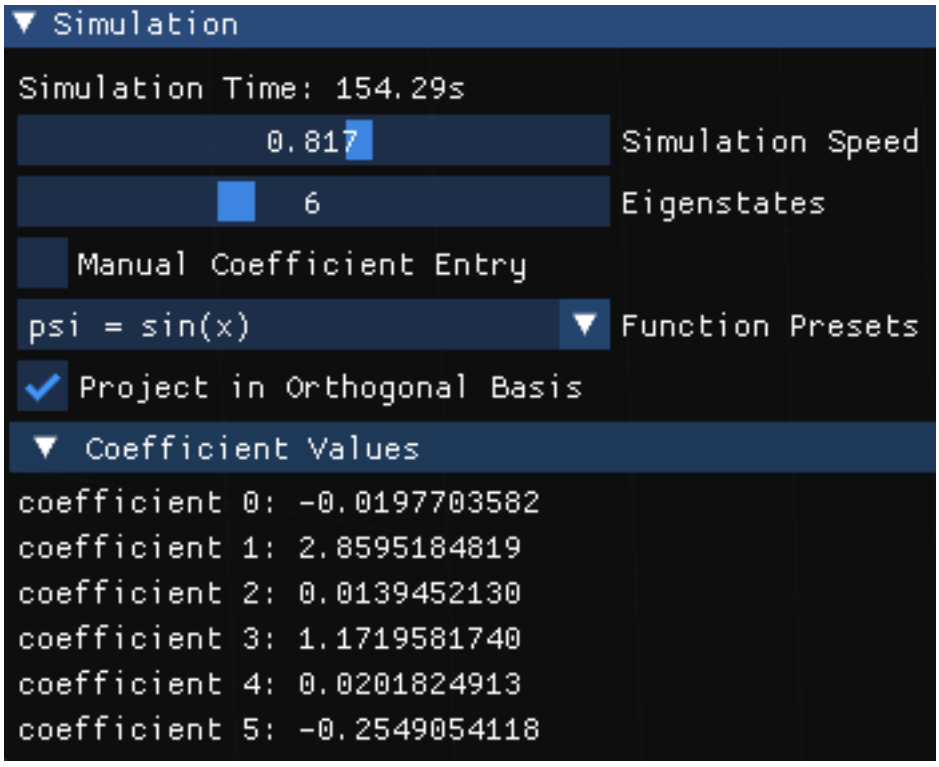

Figure 4.4: The Simulation Panel.

Presets" dropdown menu. There are currently five presets representing the following wavefunctions:

$$|\psi(x)\rangle = sin(x)$$

$$|\psi(x)\rangle = 1if[x = highestEigenstate]else0$$

$$|\psi(x)\rangle = random$$

$$|\psi(x)\rangle = x - 2$$

$$|\psi(x)\rangle = \frac{1}{x+1}$$

The Simulation panel also allows the user to set the coefficients of each eigenstate manually if the "Manual Coefficient Entry" checkbox is selected. The user can also choose to apply the selected function directly to the eigenstate coefficients without orthogonal basis projection by de-selecting the "Project in Orthogonal Basis" checkbox.

Measurements of the wavefunction can be taken by using the controls in the "Measurement" panel (figure 4.5). A single measurement can be taken by clicking the Measure button, or measurements can automatically be taken at a regular interval by activating the Auto-measure checkbox and selecting a measurement interval using the Interval slider. When taking measurements, the probability curve of the wavefunction provides a random discrete weighted distribution function. The result of this random (weighted) sampling is visually displayed on the interface by a white glowing point on the x-axis, indicating the measured position of the particle. This measurement can in turn be sent out via OSC to another application, or control the panner within QHOSYN's sound engine.

The "OSC" panel provides controls for configuring the OSC sending and receiving capabilities of QHOSYN (figure 4.6). By selecting the OSC Sender On checkbox, the controls for OSC sending are revealed (same for the OSC Receiver On checkbox). These controls allow the user to configure the IP address and Port to send the OSC messages on, the

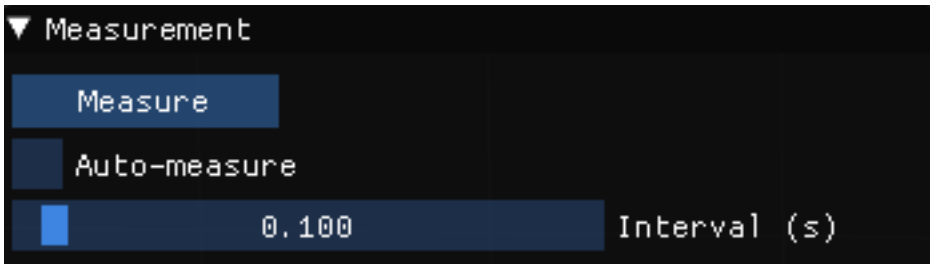

Figure 4.5: The Measurement Panel.

argument to use to identify measurement messages from QHOSYN, whether to send measurement data, and whether to send the entire waveform (as an array).

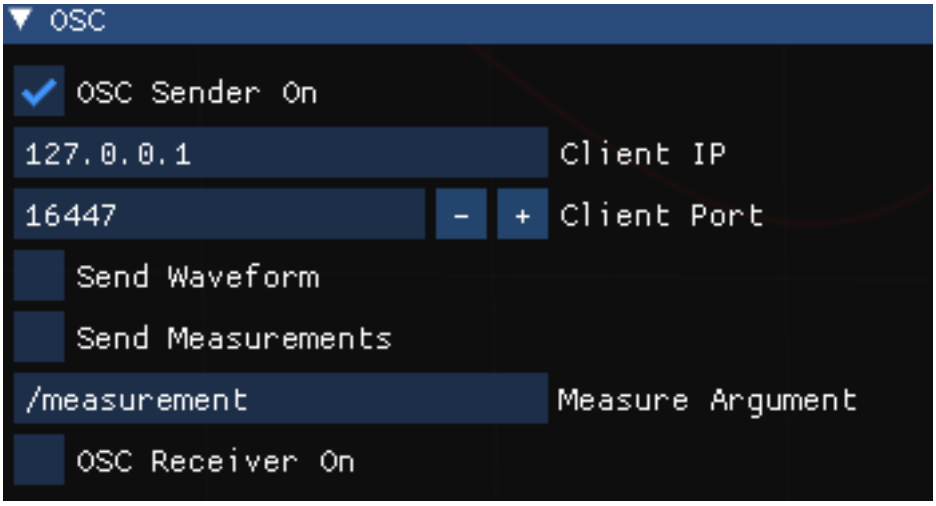

Figure 4.6: The OSC Panel.

The "Audio" panel contains the controls for QHOSYN's sound engine (figure 4.7). Audio can be toggled using the Audio On checkbox, and the volume of the audio output can be set with the Volume slider. The Panner checkbox activates the panner, which pans the sound according to the measurement results (auto-measure must be on). Both audio channels are panned independently. Channel 1 Source and Channel 2 Source dropdown menus allow the user to select the sources for these channels from the following options: none, Real Wavetable, Imaginary Wavetable, Probability Wavetable, Probability Noise Band, and Inverse Fourier Transform. The frequency slider controls the frequency of wavetable synthesis, and the Start Bin and Bandwidth sliders control the Fourier transform.

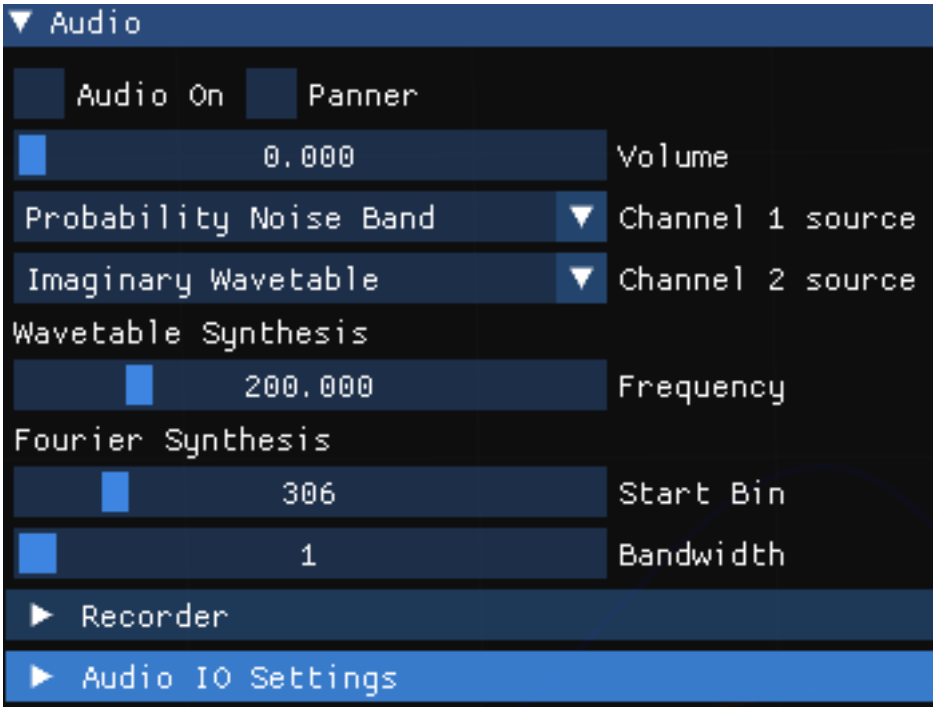

Figure 4.7: The Audio Panel.

4.4.4 Sound Engine

The internal synthesis engine of QHOSYN converts the wavefunction into sound in three ways: wavetable synthesis, inverse Fourier transform synthesis, and noise band synthesis. More possibilities emerge by pairing QHOSYN with an external application through OSC communication.

The wavetable synthesizer treats the wavefunction as a 256 sample waveform and reads through the waveform a number of times per second determined by the value of the Frequency slider set by the user(S4.1). The wavetable can be derived from the real values of the wavefunction (Y axis), the imaginary values (the z axis), or the probability distribution (the magnitude squared of the wavefunction). The wavetable reader always proceeds from left to right on the x axis. When the synthesizer read head reaches the end of the waveform, it wraps back around to the beginning. If the read head lands between sampled values of the wavefunction, it linearly interpolates between the preceding and succeeding sample. The waveform is updated 60 times per second, at the visual frame

rate of QHOSYN.

The wavefunction is stored in memory as an array of 256 complex numbers for each time step. The inverse Fourier transform method of synthesizing the wavefunction treats these arrays as frames of a Short-Time Fourier Transform (STFT). An STFT breaks a signal into blocks and applies a Fast Fourier Transform algorithm to each block, returning a series of complex-valued arrays that represent slices of the signal in time. These complex arrays provide information about the frequency, magnitude, and phase content of the signal. One can convert this STFT representation of the signal back into a time-domain signal through an Inverse Short-Time Fourier Transform (ISTFT) process. The inverse Fourier synthesis of QHOSYN takes advantage of the morphological similarity between the data of the wavefunction and the STFT.

Directly applying an STFT to the wavefunction resulted in an unusable sound because the data was too correlated. The solution is much more intensive and involves doing an FFT of the data, convolving the time-domain signal by a filter kernel (hamming window), and then converting it to the frequency domain and back once again. The final result resembles a band-limited impulse train that evolves over time(S4.2🔊).

An extension of this technique leads to the noise band synthesis method of QHOSYN. This method follows the same procedure of taking the IFFT of the wavefunction data, except that it randomizes the phase. First, the magnitude of each complex number is taken ($mag = \sqrt{im^2 + re^2}$). Then a random phase is calculated and the real part of the bin is calculated ($re = cos(phase) * mag$) and the imaginary part is calculated ($im = sin(phase) * mag$). The IFFT is then taken form this data. The result of this method is colored noise that is shaped by the probability distribution curve of the wavefunction, shifting and morphing in time(S4.3🔊).

Beyond these synthesis algorithms, another sonic effect was derived from the wavefunction. The panner can be applied to all of the synthesis methods described here. The panner

relies on the measurement sampler, which returns a value from the weighted probability distribution of the wavefunction. Each of the two channels gets a different value. Regardless of which synthesis method is chosen, the sound will be panned in the stereo field according to the position measurement(S4.4🔊).

Chapter 5

Psi

In November 2021 I completed work on the stereo fixed media piece *Psi*. It is the most complete musical representation of the compositional methods I have developed over the past 5 years. The majority of the final sonic result was realized over the course of a year between Fall 2020 and Fall 2021, but pre-compositional work began even earlier. With *Psi*, I thoroughly explored methods of composition using quantum mechanics and classical mechanics. I utilized my original software and the internalized intuition I have gained in creating the software and researching the relevant concepts.

The composition of *Psi* stands on the shoulders of years of research, software development, and sonic exploration. I used no fewer than six of my original software applications in the process of creating and transforming the sounds in the work. These include: QHOSYN[53], CHON[9], Pulsar~[54], PRISM-Dilate, PRISM-Mask[24], and EmissionControl2 (a joint project between Curtis Roads, Jack Kilgore, and myself)[25].

There are many ways in which quantum harmonic oscillators can be used in musical composition. In Psi, I explored at least six methods. One of the foundational musical ideas of *Psi* is a spatial one. The indeterminate position of quantum particles is among their most conspicuous qualities. QHOSYN can take periodic measurements of its quantum

harmonic oscillator simulation to generate a stream of evolving stochastic panning data, imbuing any sound with this indeterminate quantum position quality. This contrasts with classically-based panning data generated from CHON. The form of *Psi* depends in part on this spatial duality.

5.1 Form

There are three main structural features that define the formal architecture of *Psi*: an overarching metaphorical narrative, transformations and interactions between sounds, and the sound material itself.

In constructing the narrative, I employed methods that I have developed over the course of the past several years. These methods are a synthesis of screenwriting frameworks[55], narratology[56][57][58], and metaphors derived from thermodynamical laws. The narratological strategies shape the macro-structure of the piece from the top-down, while the thermodynamic metaphors carry the mesostructure from one sonic event or phrase to the next.

The narrative of *Psi* unfolds from the contrast between classical and quantum systems. The piece begins with the sound of a Newton's Cradle; the quintessential classical system. Some synthetic sounds based on the same physics are added gradually while leaving the focus on the sound of the Newton's cradle. This material is explored extensively from various perspectives, diving deeper into the sounds before zooming in to a level where quantum effects begin to emerge. In this quantum sound world, particles become smeared out into droning waves and jump around in the stereo field unpredictably. Eventually, echoes of the classical return as processed versions of the cradle sounds from the beginning, now unstable and flitting about the sound field haphazardly. Quantum particles dance like tiny corpuscular creatures popping in and out of our perception, spinning, collapsing, dispersing, and entangling with one another. The activity grows as we begin to zoom

out and widen our perspective once again. We finally emerge from the quantum sound world to re-examine the Newton's Cradle with the remnants of the quantum effects now coloring our aural perception.

On a broad scale, the piece essentially consists of three sections we can call Classical, Quantum, and Unification. The Classical section is relatively orderly, deterministic, and tangible. Sounds follow linear paths in their transformations. They attract, repel, collide, and oscillate like typical Newtonian objects. This applies to the spatial movement of sounds as well as other parameters. For example, granular textures change density, sometimes gradually thinning out, and other times being compressed (in the sense of a gas being compressed) as some new sound enters the field and increases the pressure. The Quantum section, on the other hand, presents the deeply weird and intangible nature of quantum objects. The quantum sound objects in this section jump discontinuously in the stereo field and, as with the classical sounds, other parameters of some sounds are similarly governed by this indeterminacy. The final Unification section brings the two worlds together in juxtaposition to round out the journey. These three sections bleed into one another to some extent, but they roughly begin at 0:00, 3:50, and 9:15, respectively.

On the level of phrase to phrase evolution, thermodynamic metaphors provide a framework for shaping the energy of the piece. In essence, these methods treat sounds as thermodynamic systems, and their transformations as thermodynamic processes. A sound can represent any of the three main types of thermodynamic systems: isolated (no matter or energy can enter or leave), closed (matter can not enter or leave, but energy can), or open (energy and matter freely enter and leave). Sound interactions and transformations can be of any of the 4 main types of thermodynamic processes: adiabatic (no heat transfer), isobaric (pressure remains constant), isothermal (temperature remains constant), and isochoric (volume remains constant). In this metaphor, the sounds themselves contain matter and an amount of energy determined by various sonic parameters. As an example,

two *open system* sounds can merge to increase pressure (sonic density) in an *isochoric process* when they come into contact in the soundfield.

Just as a character in a story can be motivated by external or internal forces, sonic crisis can spring from interactions between sounds or from within a sound itself. A sound can be expanded an contracted in an adiabatic process by various means (such as using PRISM-Dilate, described below). The characteristics of the sound itself can suggest a progression to the music. The sound of a Newton's cradle clicking suggests an inexorable stream of energy transfers in gradual decay into entropy.

5.2 Sound material

The sound material of *Psi* is largely synthetic except for two 96kHz/32-bit stereo recordings I took of a Newton's cradle[1]. The first sound recording of the Newton's cradle was a full process of the device from initially setting it into motion until it stopped, which lasted 1 minute and 32 seconds. In the second recording, I manually performed the Newton's cradle, creating gestures with varied energy envelopes with my hands(S5.1🔊).

The exposition of *Psi* allows the full progression of the first Newton's cradle recording to unfold. Various inserts and textures adorn the overall entropic trajectory, but it is not interrupted. When the Newton's cradle finally comes to rest, a gestural section begins with excerpts from the second recording and other sounds generated from the same material.

The majority of the pitched material in *Psi* is tuned to the Bohlen-Pierce scale. The Bohlen-Pierce scale is a 13-tone macrotonal scale that has a period of 3:1 (1900 cents) rather than an octave. A semi-tone in the Bohlen-Pierce scale is about 146 cents, about 50% larger than the semitone in standard equal-temperament. I have found this scale to be full of beautiful intervallic colours unheard of in other tuning systems. Some of these

[1] Additionally, some sounds were convolved with my own recorded impulse responses.

colours are more pure and iridescent than standard intervals can achieve, while others are rougher and muddier.

I have invented my own harmonic language in the Bohlen-Pierce scale that has allowed me to construct functional relationships between chords as well as melodic tension and release. The chord progression used at various times in *Psi* was derived from an earlier piece(S5.2🔊).

For the quantum material, over 90 unique sound files were generated one by one using QHOSYN(see Figure 5.1). Some of the sound files use QHOSYN's internal synthesis, while others use QHOSYN to control another application. Many sound files were also created from the classical material. This includes sounds generated from CHON, CHON controlling EmissionControl2, and CHON controlling Pulsar~.

5.3 Wavetable Synthesis

The first method of synthesis I tried with QHOSYN was wavetable synthesis. This possibility was immediately apparent as I watched the undulating quantum harmonic oscillator. This method simply entails reading the wavefunction as a wavetable. It seemed somewhat trivial at first, but it proved to be quite musically inspiring and was further enhanced through spatial effects using stochastic data generated by sampling the wavefunction's probability curve(S5.3🔊).

In fact the most obvious aural effect of the wavetable synthesis is the shifting momentum of the particle, not the position. Consider that the Fourier transform of a time-varying signal gives us a frequency domain signal from which we can derive the magnitude of a frequency in the original signal. On the other hand, the Fourier transform of a position-varying wave function gives us a momentum curve, the magnitude of which gives us the probability distribution of possible momenta. Time is to frequency as position is to momentum. The

Filename	Synthesis	Src1	Src2	Panner	Freq	Note /bin	Octv/ width	Eigenstates	Function	Project	Sim Speed
QHOS_RI_NP-30(C1)-13d(x-2)P	QHOS	Real	Imag	no	30.0	C 1	13	*x-2*	*sinx*	yes	1
QHOS_PI_P-50(G1)-6d(rand)	QHOS	Prob	Imag	yes	50.0	G 1	6	*rand*	*rand*	no	1
QHOS_RI_P-70(A1)-10d(rand)	QHOS	Real	Imag	yes	70.0	A 1	10	*rand*	*rand*	no	0.367
QHOS_RP_P-32(C#1)-3d(x-2)P	QHOS	Real	Prob	yes	32.4	C# 1	3	*x-2*	*x-2*	yes	1
QHOS_IP_P-36(D1)-4d(rand)P	QHOS	Imag	Prob	yes	35.7	D 1	4	*rand*	*rand*	yes	2.5
QHOS_RI_NP-39(E1)-5d(0ifnmax)	QHOS	Real	Imag	no	38.6	E 1	5	*0ifnmax*	*0ifnmax*	no	0.5
QHOS_RI_P-42(F1)-8d(0ifnmax)	QHOS	Real	Imag	yes	42.0	F 1	8	*0ifnmax*	*0ifnmax*	no	0.55
QHOS_PI_P-46(F#1)-7d(rand)P	QHOS	Prob	Imag	yes	45.9	F# 1	7	*rand*	*rand*	yes	1.5
QHOS_PP_P-50(G1)-9d(rand)P	QHOS	Prob	Prob	yes	50.0	G 1	9	*rand*	*rand*	yes	1.3
QHOS_PP_P-54(H1)-9d(x-2)P	QHOS	Prob	Prob	yes	54.0	H 1	9	*x-2*	*x-2*	yes	1.3
QHOS_RI_NP-54(H1)-9d(sinx)P	QHOS	Real	Imag	no	54.0	H 1	9	*sinx*	*sinx*	yes	1.75
QHOS_PI_P-59(H#1)-5d(sinx)	QHOS	Prob	Imag	yes	58.8	H# 1	5	*sinx*	*sinx*	no	4.68
QHOS_PR_P-64(J1)-5d(rand)P	QHOS	Prob	Real	yes	64.3	J 1	5	*rand*	*rand*	yes	1
QHOS_PR_P-70(A1)-7d(rand)	QHOS	Prob	Real	yes	70.0	A 1	7	*rand*	*rand*	no	1
QHOS_PP_P-76(A#1)-10d(rand)P	QHOS	Prob	Prob	yes	75.6	A# 1	10	*rand*	*rand*	yes	1
QHOS_RI_P-83(B1)-11d(rand)P	QHOS	Real	Imag	yes	83.3	B 1	11	*rand*	*rand*	yes	1.05
QHOS_PP_P-83(B1)-11d(rand)	QHOS	Prob	Prob	yes	83.3	B 1	11	*rand*	*rand*	no	1.05
QHOS_RI_NP-90(C2)-9d(rand)P	QHOS	Real	Imag	no	90.0	C 2	9	*rand*	*rand*	yes	1.25
QHOS_RI_NP-90(C2)-2d(sinx)P	QHOS	Real	Imag	no	90.0	C 2	2	*sinx*	*sinx*	yes	1
QHOS_RI_NP-97(C#2)-8d(sinx)P	QHOS	Real	Imag	no	97.20	C# 2	8	*sinx*	*sinx*	yes	1
QHOS_RI_P-107(D2)-6d(sinx)	QHOS	Real	Imag	yes	107.14	D 2	6	*sinx*	*sinx*	no	2.39
QHOS_RI_P-116(E2)-7d(0ifnmax)	QHOS	Real	Imag	yes	115.71	E 2	7	*0ifnmax*	*0ifnmax*	no	1
QHOS_PN_P-126(F2)-10d(0ifnmax)	QHOS	Prob	None	yes	126.00	F 2	10	*0ifnmax*	*0ifnmax*	no	1
QHOS_RI_P-138(F#2)-9d(rand)	QHOS	Real	Imag	yes	137.76	F# 2	9	*rand*	*rand*	no	1.5
QHOS_RI_P-150(G2)-7d(rand)	QHOS	Real	Imag	yes	150.00	G 2	7	*rand*	*rand*	no	1
QHOS_RI_NP-150(G2)-2d(sinx)P	QHOS	Real	Imag	no	150.00	G 2	2	*sinx*	*sinx*	yes	var
QHOS_PN_P-162(H2)-5d(sinx)	QHOS	Prob	None	yes	162.00	H 2	5	*sinx*	*sinx*	no	2
QHOS_RI_P-176(H#2)-4d(x-2)	QHOS	Real	Imag	yes	176.40	H# 2	4	*x-2*	*x-2*	no	1
QHOS_IN_P-193(J2)-9d(x-2)P	QHOS	Imag	None	yes	192.86	J 2	9	*x-2*	*x-2*	yes	1
QHOS_PR_P-210(A2)-9d(rand)P	QHOS	Prob	Real	yes	210.00	A 2	9	*rand*	*rand*	yes	1
QHOS_PN_P-227(A#2)-7d(rand)P	QHOS	Prob	None	yes	226.80	A# 2	7	*rand*	*rand*	yes	1
QHOS_RI_P-227(A#2)-5d(rand)P	QHOS	Real	Imag	yes	226.80	A# 2	5	*rand*	*rand*	yes	1

Figure 5.1: Inventory of quantum-based sound files created using QHOSYN and other software.

Filename	Synthesis	Src1	Src2	Panner	Freq	Note /bin	Octv/ width	Eigenstates	Function	Project	Sim Speed
QHOS_RI_NP-250(B2)-9d(rand)P	QHOS	Real	Imag	no	250.00	B	2	9	*rand*	yes	1.5
QHOS_PP_P-270(C3)-3d(rand)	QHOS	Prob	Prob	yes	270.00	C	3	3	*rand*	no	1
QHOS_RI_NP-270(C3)-5d(357)	QHOS	Real	Imag	no	270.00	C	3	5	*357*	no	1
QHOS_RI_NP-270(C3)-7d(357)	QHOS	Real	Imag	no	270.00	C	3	7	*357*	no	0.5
QHOS_RI_P-292(C#3)-7d(357)	QHOS	Real	Imag	yes	291.60	C#	3	7	*357*	no	0.5
QHOS_PI_P-321(D3)-9d(rand)P	QHOS	Prob	Imag	yes	321.43	D	3	9	*rand*	yes	1
QHOS_PI_P-347(E3)-11d(rand)P	QHOS	Prob	Imag	yes	347.14	E	3	11	*rand*	yes	1
QHOS_RI_P-378(F3)-9d(rand)P	QHOS	Real	Imag	yes	378.00	F	3	9	*rand*	yes	1
QHOS_PI_P-413(F#3)-15d(rand)	QHOS	Prob	Imag	yes	413.27	F#	3	15	*rand*	no	0.5
QHOS_PN_P-450(G3)-14d(rand)P	QHOS	Prob	None	yes	450.00	G	3	14	*rand*	yes	1.25
QHOS_RI_P-486(H3)-9d(x-2)P	QHOS	Real	Imag	yes	486.00	H	3	9	*x-2*	yes	0.6
QHOS_RI_P-529(H#3)-3d(rand)	QHOS	Real	Imag	yes	529.20	H#	3	3	*rand*	no	1
QHOS_RI_P-579(J3)-5d(rand)P	QHOS	Real	Imag	yes	578.57	J	3	5	*rand*	yes	1
QHOS_RI_P-630(A3)-7d(rand)	QHOS	Real	Imag	yes	630.00	A	3	7	*rand*	no	1.5
QHOS_RI_P-680(A#3)-9d(rand)	QHOS	Real	Imag	yes	680.40	A#	3	9	*rand*	no	1
QHOS_PI_P-750(B3)-11d(rand)P	QHOS	Prob	Imag	yes	750.00	B	3	11	*rand*	yes	1
QHOS_RI_NP-810(C4)-1d(rand)P	QHOS	Real	Imag	no	810.00	C	4	1	*rand*	yes	2
QHOS_PN_P-810(C4)-15d(rand)	QHOS	Prob	None	yes	810.00	C	4	15	*rand*	no	1
PULSAR_RI_P-12&13()-5d(rand)P	PULSAR	Real	Imag	yes	12&13			5	*rand*	yes	1
QHOS_NN_NP-1(1)-3d(rand)P	QHOS	Noise	Noise	no	1	1		3	*rand*	yes	0.5
QHOS_NN_P-1(1)-13d(sinx)	QHOS	Noise	Noise	yes	1	1		13	*sinx*	no	0.413
QHOS_NN_P-1(1)-13d(rand)	QHOS	Noise	Noise	yes	1	1		13	*rand*	no	0.7
QHOS_FN_P-114(1)-6d(rand)	QHOS	Fourier	Noise	yes	114	1		6	*rand*	no	1
QHOS_FN_P-489(1)-7d(rand)	QHOS	Fourier	None	yes	489	1		7	*rand*	no	1
QHOS_PF_P-30(C1)-10d(x-2)P	QHOS	Prob	Fourier	yes	30	C	1	10	*x-2*	yes	1.5
QHOS_NN_NP-1(1)-9d(sinx)	QHOS	Noise	Noise	no	1	1		9	*sinx*	no	0.4
QHOS_FN_P-1(1)-14d(x-2)P	QHOS	Fourier	Noise	yes	1	1		14	*x-2*	yes	0.3
QHOS_FN_P-400(5)-8d(x-2)P	QHOS	Fourier	None	yes	400	5		8	*x-2*	yes	0.8
PULSAR_Basicwaves_P-var(var)-5d(sinx)P	PULSAR	Basic	Waves	yes	var			5	*sinx*	yes	0.5
PULSAR_RI_P-var(gestures)-7d(1÷x+1)P	PULSAR	Real	Imag	yes	var			7	*1÷(x+1)*	yes	1.2
PULSAR_RI_P-var(gestures)-9d(1÷x+1)P	PULSAR	Real	Imag	yes	var			9	*1÷(x+1)*	yes	0.7
PULSAR_RI_P-var(textures)-11d(1÷x+1)P	PULSAR	Real	Imag	yes	var			11	*1÷(x+1)*	yes	1

Figure 5.2: Inventory cont.

Filename	Synthesis	Src1	Src2	Panner	Freq	Note /bin	Octv/ width	Eigenstates	Function	Project	Sim Speed
EC2_NewtonFilt_P-7d(x-2)P	EC2	Newton	Filter	yes				7	*x-2*	yes	0.4
EC2_Newton_P-14d(rand)P	EC2	Newton		yes				14	*rand*	yes	1.5
EC2_NewtonRev_P-5d(0ifnmax)P	EC2	Newton	Rev	yes				5	*0ifnmax*	yes	1
EC2_NewtonLow_P-11d(sinx)P	EC2	Newton	Low	yes				11	*sinx*	yes	2
EC2_NewtonLowFilt_P-8d(1÷x+1)P	EC2	Newton	LowFilt	yes				8	*1÷(x+1)*	yes	0.2
EC2_Newton_P-9d(1÷x+1)P	EC2	Newton		yes				9	*1÷(x+1)*	yes	0.95
EC2_QHOS-BP1(2)-Dense_P-5d(1÷x+1)P	EC2	QHOS	Dense	yes	BP1		2	5	*1÷(x+1)*	yes	0.35
EC2_QHOS-BP1(1)-Sparse_P-7d(0ifnmax)P	EC2	QHOS	Sparse	yes	BP1		1	7	*0ifnmax*	yes	1
EC2_QHOS-BP1(1)-Sp-Dn_P-3d(1÷x+1)P	EC2	QHOS	Sp-dn	yes	BP1		1	9	*1÷(x+1)*	yes	0.5
EC2_QHOS-BP1(1)-var_P-3d(rand)P	EC2	QHOS	var	yes	BP1		1	3	*rand*	yes	0.5
EC2_QHOS-BP2(2)-Dense_P-15d(rand)P	EC2	QHOS	Dense	yes	BP2		2	15	*rand*	yes	0.3
EC2_QHOS-BP2(1)-Sparse_P-7d(1÷x+1)P	EC2	QHOS	Sparse	yes	BP2		1	7	*1÷(x+1)*	yes	0.7
EC2_QHOS-BP2(1)-Sp-Dn_P-5d(0ifnmax)P	EC2	QHOS	Sp-dn	yes	BP2		1	5	*0ifnmax*	yes	1
EC2_QHOS-BP3(2)-Dense_P-9d(rand)P	EC2	QHOS	Dense	yes	BP3		2	9	*rand*	yes	2
EC2_QHOS-BP3(1)-Sparse_P-3d(1÷x+1)P	EC2	QHOS	Sparse	yes	BP3		1	3	*1÷(x+1)*	yes	1
EC2_QHOS-BP3(1)-Sp-Dn_P-7d(1÷x+1)P	EC2	QHOS	Sp-dn	yes	BP3		1	7	*1÷(x+1)*	yes	1
EC2_QHOS-BP4(2)-Dense_P-7d(rand)P	EC2	QHOS	Dense	yes	BP4		2	7	*rand*	yes	0.5
EC2_QHOS-BP4(1)-Sparse_P-15d(rand)P	EC2	QHOS	Sparse	yes	BP4		1	15	*rand*	yes	0.5
EC2_QHOS-BP4(1)-Sp-Dn_P-9d(sinx)P	EC2	QHOS	Sp-dn	yes	BP4		1	9	*sinx*	yes	1.5
EC2_QHOS-BP1-4(2-1)_P-9d(x-2)P	EC2	QHOS		yes	BP1-4		2-1	3	*x-2*	yes	0.3
EC2_QHOS-BP1-4(1)_P-3d(1÷x+1)P	EC2	QHOS		yes	BP1-4		1	9	*1÷(x+1)*	yes	0.5
EC2_QHOS-BP1-4(1-w)_P-5d(sinx)P	EC2	QHOS		yes	BP1-4		1-w	5	*sinx*	yes	0.3
EC2_QHOS-BP1-4(w-w)_P-3d(x-2)P	EC2	QHOS		yes	BP1-4		w-w	3	*x-2*	yes	0.5
EC2_3QHOS-PSI_P-2-5-7d(1)	EC2	3QHOS	Psi	yes	var			2-5-7	*x-2 & 1÷(x+1)* & 1÷(x+1)	yes	0.016
EC2_3QHOS-PSI_P-5-5-7d(2)	EC2	3QHOS	Psi	yes	var			5-5-7	x-2 & sinx & 1÷(x+1)	yes	0.5
EC2_3QHOS-PSI_P-5-5-7d(3)	EC2	3QHOS	Psi	yes	var			5-5-7	x-2 & sinx & 1÷(x+1)	yes	0.08
EC2_3QHOS-PSI_P-5-5-7d(4)	EC2	3QHOS	Psi	yes	var			5-5-7	*x-2 & 1÷(x+1)* & 1÷(x+1)	var	0

Figure 5.3: Inventory cont.

phase of a sound signal varies over time whereas the phase of a wavefunction varies over position. The derivative of phase with respect to time (the rate of change of phase over time) is equal to frequency. The derivative of phase with respect to position (the rate of change of phase over position) is equal to spatial frequency.

Treating the wave function as a table for wavetable synthesis essentially converts the position axis to a time axis. One way to imagine a real physical situation where this could occur is to imagine the wave propagating toward you in space as a pressure wave. Over time, you will encounter each "position" in the wave function as it hits your ear. Of course, this is not how a real quantum particle works, but such is the concession made by translating the wave function into sound.

The spatial frequency of the Fourier transform becomes temporal frequency. Thus, the evolving spectrum we hear is analogous to the evolving momentum distribution over time. The spectrum is harmonic due to the quantized nature of the energy states of a particle.

In *Psi*, much of the pitched material was generated using wavetable synthesis. The section from ~3:50-5:20 especially focuses on this material.

5.4 Wavefunction as spectrum

The nature of the waveform being complex-valued and changing in time suggested Short-Time Fourier Transform (STFT) methods for sonification. The wavefunction consists of 256 complex-valued data points per frame, so I thought of using that as 256 bins of a frequency domain signal which I could then treat with an inverse short-time Fourier transform to create a time-domain audio signal.

The results were lackluster at first. Because so many "bins" contained a good deal of energy, I expected a noisy audio signal in the result, which may have been interesting in its own right. However, what emerged was essentially a single frequency with large sidebands.

I surmise that this is a result of the phase of each bin being too closely correlated to all of the others. I did make some attempts at modifying the phase of each bin, but the result was essentially pure white noise which had no correlation to the initial wave function.

I found two methods that produced interesting results. The first was two randomize the phase and to apply the magnitude envelope of the wavefunction to the bins, as described in the previous chapter. In effect, this created filtered noise with the spectrum of the noise mirroring the shape of the probability distribution. The noise generated this way is dynamic and colorful thanks to the evolving patterns of the wavefunction(S5.4🔊).

The other method kept the phase information of the wavefunction, but solved the phase correlation problem by using a process of convolution with a filter kernel. This produced a kind of clicking sound like an impulse train(S5.5🔊). I found this clicking impulsive sound generated from the quantum harmonic oscillator to be a useful counterpart to the Newton's cradle sounds in *Psi*.

The sounds from the IFFT and wavetable synthesizers of QHOSYN were sometimes further processed with various audio effects and were also used as a basis for granular synthesis.

5.5 Granular synthesis

Granular synthesis is an apt framework for the sonification of quantum processes. Just as quantum physics explores the most infinitesimal objects in the universe, granular synthesis allows composers to create music using the smallest sonic particles. For my granular explorations of quantum physics in *Psi*, I used two software applications in conjunction with QHOSYN: EmissionControl2, and Pulsar~.

EmissionControl2 provided a broad field of possibilities to explore with its 15 parameters, unlimited sound file granulation, and flexible modulation scheme[59]. I was able to send

measurements of the particle's position in QHOSYN via OSC to control parameters of EmissionControl2. One of the most direct mappings was to apply these measurements to panning, so that spatial measurements applied to spatial effects in the sound.

Granular synthesis also afforded a metaphor with wave-particle duality. Sonic particles, after all, are actually waves, though they have finite bounds in time and space. Additionally, sound particles make up a Gestalt in granular synthesis that itself might be perceived as wave-like or particulate. Throughout *Psi* wave-particle duality is evoked by sounds and grain streams in the frequency range at the edge of perception between rhythm and pitch (15-25hz). In this range, the lister sometimes perceives individual grains, and sometimes perceives a low frequency wave formed from the fusing of grains.

5.5.1 Pulsar Synthesis

Pulsar synthesis is a method of granular synthesis that emits grains at a regular interval. The pulsar is made up of a "pulsaret" followed by a silent interval. The composer can generate myriad musical sounds by controlling the proportion of pulsaret to silence in the pulsar, the waveform of the pulsaret, the amplitude envelope, and the frequency of the pulsar stream[60]. I created Pulsar~ in 2018 to explore the possibilities of this technique in Pure Data. A major benefit of Pulsar~ in the case of composing for *Psi* was the ability to flexibly apply data to the parameters thanks to the Pure Data environment.

Several sounds in the piece were created by using Pulsar~ with CHON(S5.6🔊). Up to 4 instances of Pulsar~ were instantiated and each instance was controlled by the data from a different oscillator in CHON. I set up the system so that the displacement of each particle in the x axis would control the panning of its corresponding pulsar. I also explored configurations in which the y and z axes of the particles were mapped to the formant frequency (pulsaret period) and or masking of the particles. I controlled other aspects of the Pulsars manually to create active gestures and textures. I used this method

to generate some of the pitched material in the classical sections of *Psi* (notably the gesture at 0:26).

Pulsar synthesis presents an especially compelling analogy for the perception of the quantum mechanical system in question. With it's short, regular pulses, a sampling of the quantum oscillator applied to some of the parameters of pulsar synthesis quickly immerses the listener into the quantum world. I set up two Pulsar~ objects in Pure Data, and routed the measurement data to each of them to control their panning. I alternated the routing so that the two pulsars did not change position at the same time. I also sent the waveform itself to the pulsars. I mapped the real part of the wavefunction to one pulsaret waveform, and the imaginary part to the other (see figure 5.4 and media example S5.7🔊). In this way, I was able to explore pulsar synthesis with a new feature: pulsaret waveforms that evolve in time.

5.5.2 3D QHO granular clouds

I ran three instances of QHOSYN to simulate a three-dimensional quantum harmonic oscillator, representing the x, y, and z axes, respectively. This is a valid approach to simulating a 3D quantum harmonic oscillator since the dimensional terms are separable in the three-dimensional Schrödinger equation[61]. EmissionControl2 was then configured to listen to the OSC output from all three of these QHOSYN instances (see Figure 5.5). The x-axis measurement results governed the stereo panning of grains. The y-axis measurement controlled the pitch of grains by modulating the Playback Rate parameter. The z-axis measurements affected the a sense of "depth" of the grains by modulating the Grain Duration and at the same time modulating the filter Resonance (higher z value resulted in longer and more filtered grains, lower z created shorter and less filtered grains).

Using this configuration, I was able to create three-dimensional quantum granular clouds(S5.8🔊). This method approached a kind of sonification of a particle in 3D

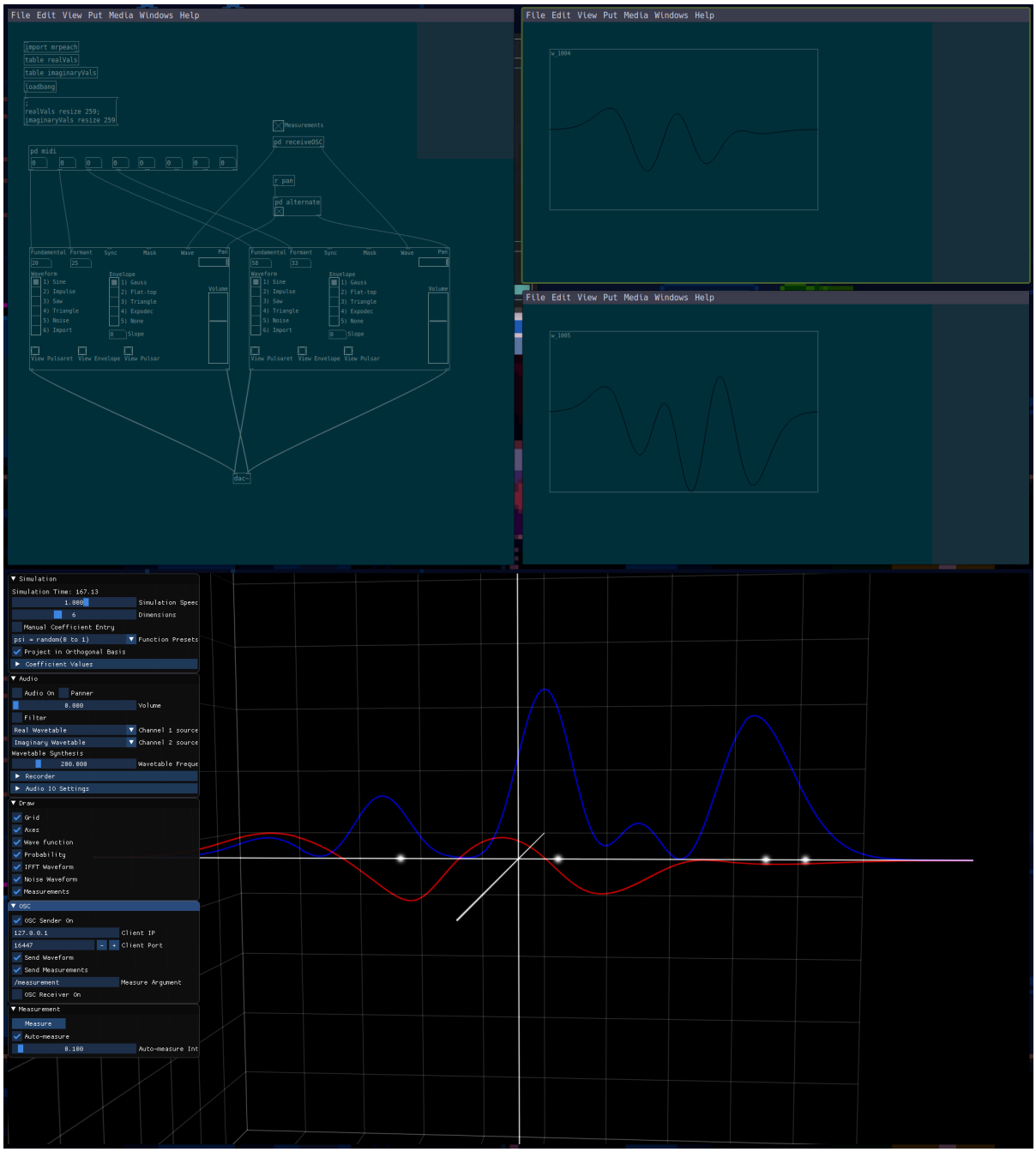

Figure 5.4: Creating quantum pulsars with QHOSYN and two Pulsar~ objects in Pure Data. The real and imaginary parts of the wavefunction provide the waveforms for the two pulsarets. The pulsaret waveforms can be seen in the top right.

Figure 5.5: Using three instances of QHOSYN to control three parameters in EC2 and achieve 3-dimensional quantum granular textures.

space measured at regular intervals. Under this correspondence, each grain represents the particle in the spatial location it is found each time it is measured. While each measurement represents the particle at a time and place, a collapsing of the positional potential of the particle, the sonic result is not a well-defined particle, the perceptual result of so many rapid measurements is a Gestalt mass or cloud of sound that loses definition and takes on a fuzzy character. Thus, the resulting granular clouds contain traces of the quantum probability cloud.

By manually controlling other parameters such as Grain Rate, Asynchronicity, Intermittency, Filter Center, and Envelope Shape, I was able to create a variety of quantum cloud textures. Naturally, the choice of which sound file to granulate had an impact on the quantum cloud texture as well. For this, I used several of the source recordings for *Psi* as well as a complete draft of *Psi* itself.

5.6 The PRISM Plugins

In processing the sounds of *Psi*, I made use of two plugins from my forthcoming suite of novel spectral processing effects. These VST plugins are coded in C++ using JUCE. They allow composers to process the spectrum of a sound in unconventional ways.

The first plugin, PRISM-Dilate, is a re-imagining of my software PISCES, which I created in 2017 and used to write my piece *BachFlip*. It employs an algorithm I call spectral dilation, that takes inspiration from the concept of a homothetic transformation in affine geometry. It applies a similar transformation to the spectrum of an input sound, pulling the spectrum to a point, pushing it away from a point, or flipping it around the point(S5.9🔊). A vizualization of the effect can be seen in Figure 5.6

The second plugin, PRISM-Mask, was created during the composition of *Psi* to realize a specific effect I wanted to achieve. I imagined a sound fading out or fading in by gradually

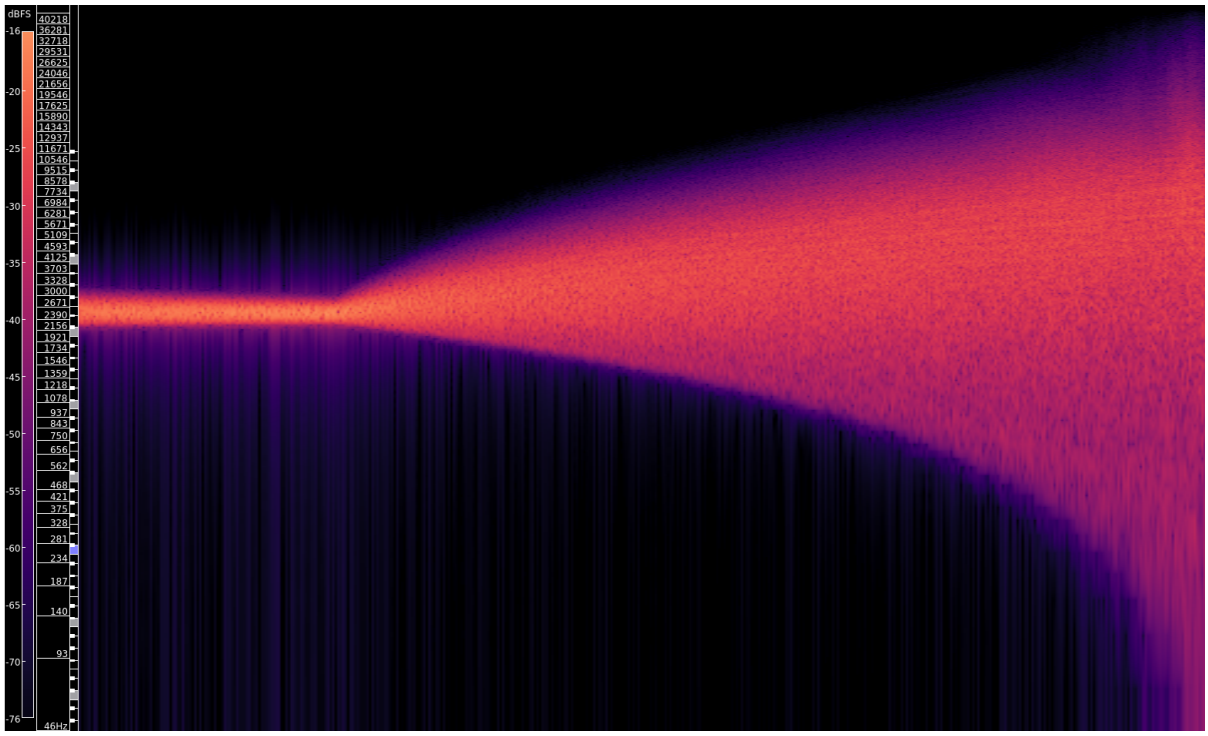

Figure 5.6: A spectrogram showing the effect of spectral dilation. A noisy signal is initially dilated toward the focal point of 2500Hz and then the effect is gradually lessened, returning the sound to its unprocessed state.

masking or unmasking frequency bins(S5.10). The effect is used notably to create the first sound in *Psi*. A Newton's cradle click is passed through a reverb and subsequently reversed, generating a sound like white noise crescendo that peaks at the first click of the Newton's cradle. However, rather than crescendo by growing in amplitude, the white noise fades in by gradually unmasking frequency bands one at a time until the whole spectrum is revealed (see Figure 5.7). The effect is also used in an extended 1 minute long gesture at the end of the piece, where frequency bins are masked and unmasked in waves and crashes.

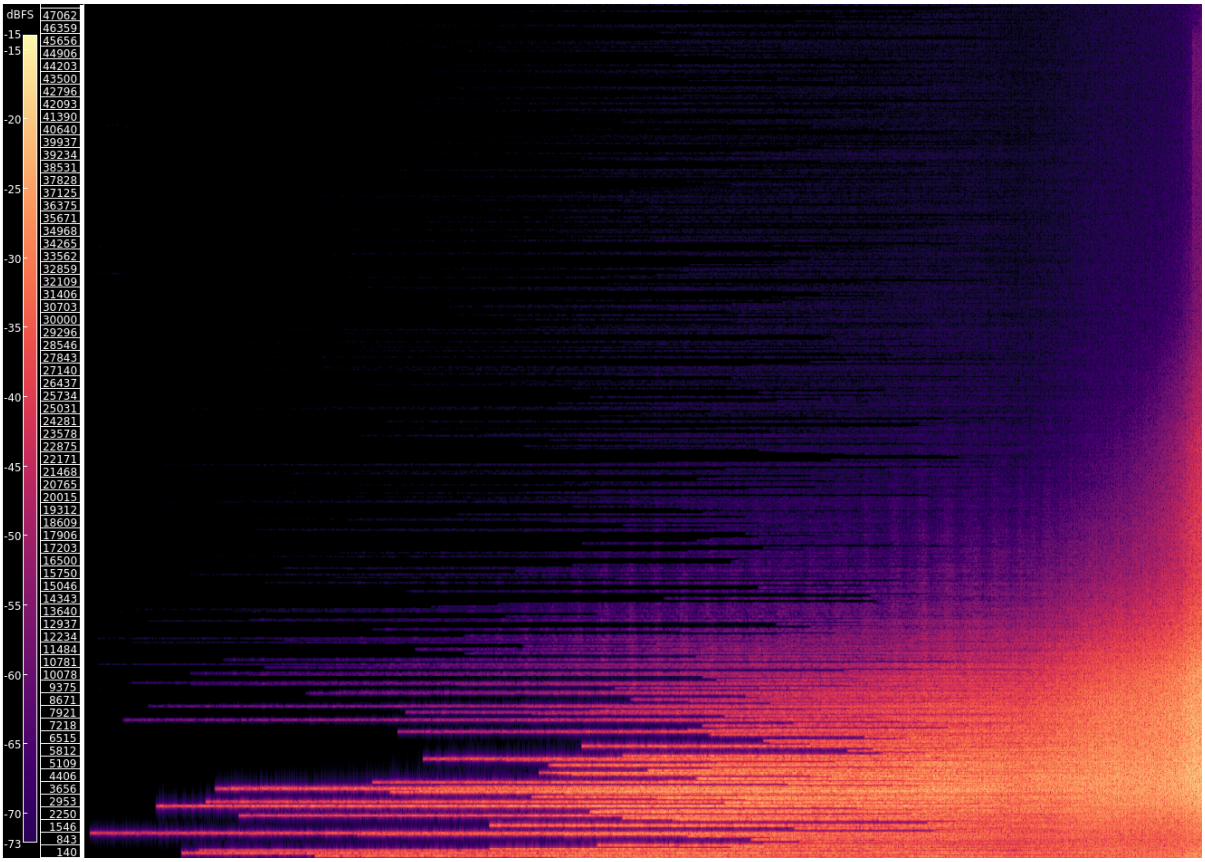

Figure 5.7: A spectrogram showing spectral masking. All bins are initially masked and subsequently unmasked one-by-one.

Chapter 6

Evaluation of Classical and Quantum Aesthetics

Sonification and musification of scientific data and frameworks creates opportunities and poses challenges for the composer. Physics in particular is perhaps the most accommodating of the translation into music of all the sciences. It may be because of the tangibility of the theoretical models, though that aspect does not apply to quantum physics. Certainly any dynamical time-varying theory will lend itself to sonification as music is a dynamic, time-varying art form. Some models in science are static descriptions of a system, and these models tend to lead to musical dead-ends. The most musical scientific models describe the way multiple objects or agents interact over time. From a model such as this, contrapuntal relationships, dynamic gestures, and shifting hierarchies can be derived that naturally propel music forward.

Classical mechanics and quantum mechanics both describe the motion and other physical properties of objects. They are dynamic by definition. This suggests a musical potential, but it is still up to the composer to mold these forces into a compelling piece. In composing *Psi*, I found that these models provide a starting point, sometimes as musical material to

be arranged and manipulated, other times as a framework for shaping the form of the music. Ultimately, a composer's touch is needed to tell the story.

Working with the material also carved new intuitions onto my psyche, so that I was subconsciously applying these models according to heuristically learned patterns. Eventually, the algorithm may become superfluous when the intuitive shaping of the music produces the same perceived results with less effort. For now, I find the most aesthetically satisfying process to be a combination of algorithmic and heuristic models.

6.1 Classical Physics as a Musical Model

Classical physics offers compelling models for music. It provides a familiar anchor for listeners that is deterministic, causal, and logical. If anything, the composer using classical physics as a model needs to be careful not to create music that is *too predictable* to the point of becoming tedious. On the other hand, musical genres like acousmatic music, which lack the physical, embodied basis that live ensemble music engenders by its very nature, can benefit from a grounding in classical physics.

I have previously discussed the value of the classical mechanical model as a visceral and universal logical framework for music[9]. I have found the logic of classical physics to be a common ground to which any listener can relate. Unlike other frameworks for music such as scales, metric hierarchies, and stylistic tropes, classical physics is universal and not culturally learned. It is the logic of an apple falling to the ground when it grows too heavy for its stem, or a billiard ball begin struck and rolling in a predictable direction based on the angle of impact.

A sound can tell a story as vivid as a visual scene or a word.

A narrative is most often associated with linguistic communication, but narrative itself is an abstract concept that can manifest in many media. Byron Almén [58, p. 13] even

suggests that musical narrative is one of the most pure forms of narrative because of its abstract nature allowing it to express meaning without reference to specific characters and settings. This can be heard in *Psi*, for example, in the first 3 minutes of the piece, where many sounds enter and leave the scene, but they generally always move in a spatially logical trajectory. Some of these sounds bear no obvious identity with a real object, but the metaphor of motion and causality is still understood on a deep level(S6.1🔊).

Composers can construct an illusion of causality from any sonic parameter, but by using principles from classical physics, the causality comes pre-built in the mind of the listener. The necessity of a sense of causality can't be overstated. Curtis Roads [62, p. 328] points out:

> An impression of causality implies predictability. If listeners are not able to correlate sonic events with any logic, the piece is an inscrutable cipher.

In *Psi*, the first 3 minutes and 30 seconds are rooted in classical mechanics in various ways. Sounds are spatialized so that they follow classically consistent trajectories, the Newton's cradle sound symbolizes and exemplifies Newtonian mechanics, and CHON is used to create complex but causal relationships between sounds. This is set up so that it can be ripped away when the piece dives into quantum territory in the rest of the piece. The exploration of causality itself is a core component of *Psi*, and this only comes across if overtly causal sound worlds are presented before entering the non-deterministic quantum sound world.

Hearing causal music based on physical metaphors can elicit empathy, understanding, and a sense of immersion or participation in the sound world. In recent years, scientists have done a great deal of research into mirror neurons, which are neurons that fire in a similar manner when performing an action and when observing that same action being done[63][64]. The sounds of actions can also trigger these mirror neurons, and there is even some activation when listening to music, suggesting a sense of empathy with

the "motion" of the music[65][66]. So, by presenting a sound that is somehow physical and familiar to the listener, a composer can induce in the listener a sense of actually performing an action, thereby intensifying immersion into the listening experience.

There are several ways of presenting physical metaphors in music. These methods range from intuitive and imagined to algorithmic. I wrote of the first method[9]:

> One can treat a sound as if it had physical properties such as mass, momentum, inertia, internal energy, heat, entropy, gravity, and others. Physical metaphor involves shaping the evolution of a sound, or a group of sounds, according to how real physical objects with these properties behave. For example, Newton's first law of motion states that an object at rest will remain at rest and an object in motion will remain in motion unless acted upon by an external force. This is the law of inertia. If we follow this law with sounds, then, for example, a sound at rest will remain at rest unless it is excited by some force, which could be another sound entering the field and colliding with it. The momentum of a sound and the metaphorical collision of sounds can take shape entirely in the imagination of the composer, and this is a rich domain for creativity.

In the second method, the physical metaphor is sonified algorithmically by mapping data directly to sound. Since classical mechanics in large part is a description of motion, the most compelling mapping tends to be from the position of the objects in space to the panning of sounds in a stereo field or multichannel surround system. Applying the data to other parameters of sound can produce very interesting results, but the correspondence to the physical model tends to break down.

I have found the most compelling results by combining this free intuitive method with algorithm and simulations. In *Psi*, I used CHON to simulate a real physical system, but I also intervened in the simulation and played it like an instrument. Additionally,

I arranged sounds using micro-montage and other techniques according to what felt physically plausible based on an internalized constellation of artistic and scientific instincts. The result sets all available resources to the task of creating a sonic narrative that is both intellectually honest and aesthetically moving.

6.2 Quantum Physics as a Musical Model

If classical mechanical models in music provide an anchor to keep the listener grounded, a quantum mechanical model is a rocket that carries the listener into the stratosphere. We do not directly observe quantum effects in our daily lives so quantum effects tend to baffle our intuition.

The theory of embodied cognition tells us that the mind is intrinsically linked to the body and perceptual faculties and that we understand the world through embodied metaphors called schemata that are gathered in our interactions with the world[67]. We learn these schemata before we learn language. For example, we may learn the center-periphery schema by observing repeatedly that things can be in the center of our perception (our eyes pointed toward a point, our ears trained on a sound) or the edge of our perception, in the background. This metaphor then becomes a way of constructing meaning. We use it to make sense of organizational plans, we feel it in our social circles and cliques, and we see and hear this pattern in the the interplay of foreground and background in paintings or music. Within the framework of embodied cognition, even abstract concepts are conceptualized using physical and embodied metaphors[68]. The use of aesthetics and embodied schemata in auditory display has been receiving increased attention in recent years and has been proposed as a solution to the mapping problem of sonification[69].

How, then, can anyone hope to understand quantum mechanical effects that seem to have no corollary in our lived experiences? How can someone truly grok an object that has simultaneously particle-like and wave-like properties? Physicists can sometimes form

intuitions about the quantum world by immersing themselves in the mathematics and through interpretations, but it nevertheless remains intangible.

Music may offer one humble path forward. Indeed, it may be one of the best-suited art forms to encode quantum mechanics. Music, after all, exhibits its own kind of wave-particle duality. It is a wave-based phenomenon that relies on longitudinal pressure waves carried through a vibrating medium. The frequency, amplitude, and shape of these waves are integral to our perception of sound. The idea of superposition, often presented as a shocking property of quantum particles, is mundane in the context of sound and music. It is common knowledge that the sound of a piano string producing a C at 261Hz is actually a superposition of many states (or vibrational modes), the harmonics of C: 261Hz, 522Hz, 783Hz, 1044Hz, 1305Hz, and so on. The idea of a note, like a particle, is a quantization of a wave-phenomenon. The note is bound in space (it comes from a particular point) and time (it has a fixed duration). A note is still a wave, however.

Electroacoustic music offers a medium in which sonic objects can behave in more "quantum" ways. The point of origin of a sound can be diluted, duplicated, and rapidly shifted. A sound particle can even seem to exceed the speed of light by instantaneously panning from one side of a concert hall to another. Beyond this, there are practically an endless variety of effects that can serve as abstract metaphors for even the most intangible of ideas[1]. A sound, like the piano string, can collapse from a superposition of states (partials) to a single one. The PRISM-Dilate plugin achieves one version of this effect. Granular synthesis presents another particle-wave metaphor, spectra can be synthesized in any configuration, and sounds can be arranged in time in probabilistic distributions. Furthermore, if music can activate mirror neurons, it may induce embodied experiences of the otherwise intangible quantum processes being sonified.

Turning from what music can offer in terms of quantum intuition to what quantum physics

[1]This is not in fact exclusive to Electroacoustic music. One need only look to the programmatic music of the romantic era to see examples of music evoking abstract concepts like emotions and ideas.

can offer music, we find an abundance of inspiring material for the composer to work with. The undulating wavefunctions provide a dynamic to synthesis either by directly using the wave as such, or by treating the wave as an evolving spectrum to be converted to a time-domain signal via Fourier methods. The time-varying probability distributions of the wavefunction offer an extension to the stochastic techniques pioneered by Iannis Xenakis[4]. Further possibilities may yet emerge as composers begin to explore and play with the mathematics, data structures, and paradigms offered by quantum physics.

There is no shortage of awe, wonder, and curiosity to be drawn from the strange world of quantum physics. The challenge lies in how to map this world to sound. This challenge is difficult enough from a sonification perspective, but for the composer it is a two-headed beast with the sub-challenges of firstly maintaining the integrity and validity of the data and mapping strategy, and secondly crafting a compelling sonic narrative with aesthetic meaning. In the worst case, fixating only on the former leads to the most clinical and uninspired algorithmic music, while focusing too much on the latter results in trite and superficially programmatic music. To make the task more difficult, these two goals can clash when the raw data is aesthetically banal or the composer's narrative vision threatens to distort the data beyond recognition. I have opted for a unified approach to scientific sonification, using symbolic-iconic sounds, direct data mapping, and phenomenological intuitive composition to evoke metaphors that guide the listener through the uncharted sonic territory of quantum physics.

Quantum physics, in contrast to classical physics, does not bring any tangibility or universal framework to music. Instead, it offers surprise, intrigue, and novelty in form and structure. It also has a higher barrier to entry than classical physics. It is difficult to grasp the concepts and mathematics of quantum physics to a degree where one can be comfortable enough to compose with them. I hope that the work I did in this regard to create QHOSYN helps to mitigate this barrier for future composers and researchers.

Chapter 7

Conclusion

The methods described in this document have been successful for my purposes, but there is more work to be done. As it stands, I have released what I am calling QHOSYN v0.9 as a standalone application for Linux, MacOS, and Windows. In this state, it was suitable for my personal work, but it requires some refinement in the user interface and functionality to be of use to other musicians. It might also be worthwhile to develop a VST plugin to conform to the workflow of more musicians. Beyond QHOSYN, there is a vast range of models that remain to be adapted from quantum mechanics. For this, scientist-artist collaboration would be beneficial.

QHOSYN was largely created based on my own studies in quantum physics. I am not a physicist, so I took an autodidactic approach that required considerable effort for over a year to find an appropriate quantum model, disentangle the mathematics, and translate this into a software simulation. Though I gained a significant amount of intuition through this struggle, I would have embraced the chance to collaborate with someone more knowledgeable on the subject than myself. I have found in past projects, such as *Coacervate*, that collaboration with scientists is hugely beneficial in accelerating and enriching the process.

The quantum harmonic oscillator simulated by QHOSYN is a fundamental quantum mechanical model. With the initial version, it was desirable to focus on a very fundamental model, but I have several ideas for extending the simulation to create more interesting situations. First, I plan to implement a collapse mechanic to the simulation, so that taking a measurement can collapse the wavefunction, followed by dispersion. Second, I would like to extend QHOSYN into higher spatial dimensions (2D and 3D) to better simulate a real particle. Third, I hope to create coupled quantum oscillators that might approach a simulation of phonons or quantum fields.

In the near future, I hope to collaborate with scientists to mutually benefit from improving and broadening the scope of my physical simulations. I am looking into other numerical solutions to the time-dependent Schrödinger equation that allow for real-time user interference in the wave function's evolution. These interferences could cause wave function collapse or trigger creation and annihilation operations (adding or removing energy eigenstates).

There seems to be a great deal of promise in using quantum mechanical simulations to generate novel stochastic patterns for musical control. Research that extends the quantum harmonic oscillator model described here or that explores other quantum models could reveal a whole new class of time-varying stochastic musical forms. Certainly there is potential in exploring this on the quantum computing machines that are becoming more feasible today.

There are many ways to sonify quantum physics beyond what has been presented here. The methods I used to generate quantum granular clouds and wavefunction pulsar streams suggest further possibilities in combining quantum mechanics and granular synthesis. While QHOSYN's OSC output makes it flexible, an integrated software solution that includes the simulation and granular synthesis engine in one app would be advantageous. QHOSYN updates the wavefunction at a rate of 60 frames per second. An expansion of

the granular synthesis capabilities of this approach would benefit from a much higher refresh rate. Aside from, or perhaps parallel to, this granular method, I can also imagine an application simulating a 2D quantum harmonic oscillator that extends the wavetable synthesis to wave terrain synthesis. Other topologies are possible in even higher dimensions.

Science has been, and will continue to be, at the core of my work. From about the age of 5 until 12 years old, when anyone asked me what I wanted to be when I grew up, I would answer definitively "a scientist." I was captivated by scientific theories, models, and experiments. Beyond just a fascination with typical science experiments for kids such as model volcanoes or potato batteries (enthralling as those are), I loved math and I possessed seemingly endless stamina for disentangling complex systems and discovering how things work. Eventually, my fascination shifted to musical systems and I redirected my analytical energy. As is evident throughout this document, the scientific spirit never truly left.

Curiosity and discovery drive the musician and the scientist alike. I have extensively documented here my explorations and experiments in bringing scientific sonification and metaphor into my music. Every new method that leads to compelling sound is a musical breakthrough. QHOSYN and Psi represent the latest breakthrough in my work; a breakthrough that opens up a whole field of possibilities.

Every great scientific breakthrough results in a paradigm shift that takes some time and effort for scientists to accept, and even longer for the general public. Quantum theories represent probably the most significant (and challenging) paradigm shift in science since Galileo's heliocentric model of the solar system. The difficulty of this paradigm shift to quantum intuition can be seen in the breadth of competing interpretations put forth by scientists and philosophers. Advances in quantum physics has resulted in numerous tangible technologies and scientific breakthroughs in spite of this lack of a singular interpretation. However, it may be that these breakthroughs occur in part because of

this plethora of intuitions about quantum physics. In either case, it seems clear that contributions to new intuitions about the quantum world can aid scientists in forming and testing new hypotheses and designing effective experiments that result in scientific breakthroughs. We have ample testimony from great scientific thinkers and increasing evidence from psychology to assume that music can contribute in some small way to this goal. As Dennis Gabor noted[3]:

> Acoustical phenomena are discussed by mathematical methods closely related to quantum theory. While in physical acoustics a new formal approach to old problems cannot be expected to reveal much that is not already known, the position in subjective acoustics is rather different.

Indeed as with subjective acoustics, subjective musical perception also stands to gain a great deal from quantum theories. Quantum physics reveals new musical possibilities. Novel synthesis methods and formal structures are generated and new intuitions are formed based on quantum sensibilities. The infinitesimal vibrations of the universe become audible.

Bibliography

[1] LIGO Scientific Collaboration and Virgo Collaboration *et al.*, "Observation of Gravitational Waves from a Binary Black Hole Merger," *Physical Review Letters*, vol. 116, no. 6, p. 061102, Feb. 2016, doi: 10.1103/PhysRevLett.116.061102.

[2] D. Gabor, "Theory of communication," *Journal of the Institution of Electrical Engineers - Part III: Radio and Communication Engineering*, vol. 93, no. 26, pp. 429–457, Nov. 1946, doi: 10.1049/ji-3-2.1946.0074.

[3] D. Gabor, "Acoustical Quanta and the Theory of Hearing," *Nature*, vol. 159, no. 4044, pp. 591–594, May 1947, doi: 10.1038/159591a0.

[4] I. Xenakis, *Formalized Music: Thought and Mathematics in Composition*. Pendragon Press, 1992.

[5] C. Roads, *Microsound*. Cambridge, MA, USA: MIT Press, 2002.

[6] B. L. Sturm, C. Roads, A. McLeran, and J. J. Shynk, "Analysis, Visualization, and Transformation of Audio Signals Using Dictionary-based Methods †," *Journal of New Music Research*, vol. 38, no. 4, pp. 325–341, Dec. 2009, doi: 10.1080/09298210903171178.

[7] T. Hermann, *The Sonification Handbook*. Berlin: Logos Verlag, 2011.

[8] C. Scaletti, "Sonification [does not equal] Music," *The Oxford Handbook of Algorithmic Music*. Feb. 2018, doi: 10.1093/oxfordhb/9780190226992.013.9.

[9] R. DuPlessis, "CHON: From physical simulation to musical gesture," Master's thesis, University of California Santa Barbara, 2021.

[10] L. Hiller and P. Ruiz, "Synthesizing Musical Sounds by Solving the Wave Equation for Vibrating Objects: Part 2," *Journal of the Audio Engineering Society*, vol. 19, no. 7, pp. 542–551, Jul. 1971, Accessed: Mar. 26, 2021. [Online]. Available: https://www.aes.org/e-lib/browse.cfm?elib=2156.

[11] C. Cadoz, A. Luciani, and J.-L. Florens, "CORDIS-ANIMA: A Modeling and Simulation System for Sound and Image Synthesis - The General Formalism," *Computer Music Journal*, vol. 17, nos. 1, Spring 1993, pp. 19–29, 1993, Accessed: Mar. 28, 2021. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01234319.

[12] R. Fischman, "Clouds, Pyramids, and Diamonds: Applying Schrödinger's Equation to Granular Synthesis and Compositional Structure," *Computer Music Journal*, vol. 27, no. 2, pp. 47–69, Jun. 2003, doi: 10.1162/014892603322022664.

[13] B. L. Sturm, "Composing for an ensemble of atoms: The metamorphosis of scientific experiment into music," *Organised Sound*, vol. 6, no. 2, pp. 131–145, Aug. 2001, doi: 10.1017/S1355771801002102.

[14] L. J. Putnam, J. Kuchera-Morin, and L. Peliti, "Studies in Composing Hydrogen Atom Wavefunctions," *Leonardo*, vol. 48, no. 2, pp. 158–166, Apr. 2015, doi: 10.1162/LEON_a_00912.

[15] I. Xenakis, *Arts/Sciences: Alloys*. Pendragon Press, 2010.

[16] J.-C. Risset, "Computing Musical Sound," in *Mathematics and Music*, G. Assayag, H. G. Feichtinger, and J. F. Rodrigues, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 215–231.

[17] R. Root-Bernstein, "The Art of Innovation: Polymaths and Universality of the Creative Process," in *The International Handbook on Innovation*, Elsevier, 2003, pp.

267–278.

[18] S. Suzuki, *Nurtured By Love: A New Approach to Education*, 1st Edition. Exposition Press, 1975.

[19] R. S. Root-Bernstein, "Music, Creativity and Scientific Thinking," *Leonardo*, vol. 34, no. 1, pp. 63–68, 2001, Accessed: Feb. 27, 2020. [Online]. Available: https://www.jstor.org/stable/1576986.

[20] S. Hossenfelder, *Lost in math: How beauty leads physics astray*, First edition. New York: Basic Books, 2018.

[21] G. F. Giudice, "Naturally Speaking: The Naturalness Criterion and Physics at the LHC," in *Perspectives on LHC Physics*, WORLD SCIENTIFIC, 2008, pp. 155–178.

[22] F. Wilczek, *A Beautiful Question: Finding Nature's Deep Design*, Reprint edition. New York, NY: Penguin Books, 2016.

[23] E. A. Frankel, P. C. Bevilacqua, and C. D. Keating, "Polyamine/Nucleotide Coacervates Provide Strong Compartmentalization of Mg2+, Nucleotides, and RNA," *ACS Publications*. Feb. 2016, doi: 10.1021/acs.langmuir.5b04462.

[24] R. DuPlessis, "PRISM." Aug. 2021, Accessed: Nov. 04, 2021. [Online]. Available: https://github.com/rodneydup/PRISM.

[25] J. Kilgore, C. Roads, and R. DuPlessis, "EmissionControl2." 2020, [Online]. Available: https://github.com/EmissionControl2/EmissionControl2.

[26] T. Koch, "NMR Theory Tutorial." 2011, Accessed: Nov. 14, 2021. [Online]. Available: https://www.orgchemboulder.com/Spectroscopy/nmrtheory/NMRtutorial.shtml.

[27] D. Bertuzzi *et al.*, "General Protocol to Obtain D-Glucosamine from Biomass Residues: Shrimp Shells, Cicada Sloughs and Cockroaches," *Global Challenges*, vol. 2, Aug. 2018, doi: 10.1002/gch2.201800046.

[28] D. Morin, *Introduction to classical mechanics : With problems and solutions*. Cambridge, UK ; New York : Cambridge University Press, 2008.

[29] Plato, *The Republic*, 2nd edition. London ; New York: Penguin Classics, 2003.

[30] E. Varese and C. Wen-chung, "The Liberation of Sound," *Perspectives of New Music*, vol. 5, no. 1, p. 11, 1966, doi: 10.2307/832385.

[31] G. Ligeti, "Poéme Symphonique." 1962.

[32] Modartt, "Pianoteq," *Madartt: Virtual instruments, physically modelled*. Accessed: Mar. 26, 2021. [Online]. Available: https://www.modartt.com/.

[33] Arturia, "V-Collection." Accessed: Mar. 28, 2021. [Online]. Available: https://www.arturia.com/products/analog-classics/v-collection/overview.

[34] Eckel, Iovino, and Caussé, "Sound synthesis by physical modelling with modalys," *Proceedings of the International Symposium on Musical Acoustics*, pp. 479–482, 1995.

[35] AAS, "AAS—Instruments, synthesizer, and effect plug-ins based on physical modeling." Accessed: Mar. 28, 2021. [Online]. Available: https://www.applied-acoustics.com/.

[36] R. DuPlessis, "CHON." 2021, [Online]. Available: https://github.com/rodneydup/CHON.

[37] C. Orzel, "What Has Quantum Mechanics Ever Done For Us?" *Forbes*. Accessed: Nov. 03, 2021. [Online]. Available: https://www.forbes.com/sites/chadorzel/2015/08/13/what-has-quantum-mechanics-ever-done-for-us/.

[38] S. Siddiqui and C. Singh, "How diverse are physics instructors' attitudes and approaches to teaching undergraduate level quantum mechanics?" *European Journal of Physics*, vol. 38, no. 3, p. 035703, Mar. 2017, doi: 10.1088/1361-6404/aa6131.

[39] N. D. Mermin, "Could Feynman Have Said This?" *Physics Today*, vol. 57, no. 5, pp. 10–11, May 2004, doi: 10.1063/1.1768652.

[40] J. Tenney, *From Scratch: Writings in Music Theory*, 1st edition. Urbana: University of Illinois Press, 2015.

[41] C. Barlow, *On Musiquantics*. Musikwissenschaftliches Institut der Johannes Gutenberg-Universität Mainz, 2012.

[42] K. Dovstam, "Real modes of vibration and hybrid modal analysis," *Computational Mechanics*, vol. 21, no. 6, pp. 493–511, Jun. 1998, doi: 10.1007/s004660050328.

[43] H. Nikolic, "Quantum mechanics: Myths and facts," *Foundations of Physics*, vol. 37, no. 11, pp. 1563–1611, Nov. 2007, doi: 10.1007/s10701-007-9176-y.

[44] L. de Broglie, "Recherches sur la théorie des Quanta," PhD thesis, Migration - université en cours d'affectation, 1924.

[45] G. P. Thomson and A. Reid, "Diffraction of Cathode Rays by a Thin Film," *Nature*, vol. 119, no. 3007, pp. 890–890, Jun. 1927, doi: 10.1038/119890a0.

[46] C. J. Davisson and L. H. Germer, "Reflection of Electrons by a Crystal of Nickel," *Proceedings of the National Academy of Sciences*, vol. 14, no. 4, pp. 317–322, Apr. 1928, doi: 10.1073/pnas.14.4.317.

[47] S. J. Ling, J. Sanny, and W. Moebs, "The Quantum Harmonic Oscillator," Sep. 2016, Accessed: Nov. 15, 2021. [Online]. Available: https://opentextbc.ca/universityphysicsv3openstax/chapter/the-quantum-harmonic-oscillator/.

[48] A. R. Group, "Allolib." Allosphere Research Group, Mar. 2021, Accessed: Mar. 26, 2021. [Online]. Available: https://github.com/AlloSphere-Research-Group/allolib.

[49] "FFTW." Accessed: Nov. 11, 2021. [Online]. Available: http://www.fftw.org/.

[50] G. Scavone, "RtAudio." Accessed: Mar. 26, 2021. [Online]. Available: https://www.music.mcgill.ca/~gary/rtaudio/.

[51] "GLFW," *GLFW*. Accessed: Mar. 26, 2021. [Online]. Available: https://www.glfw.o

rg/.

[52] O. Cornut, "ImGui." Mar. 2021, Accessed: Mar. 30, 2021. [Online]. Available: https://github.com/ocornut/imgui.

[53] R. DuPlessis, "QHOSYN." 2021, [Online]. Available: https://github.com/rodneydup/QHOSYN.

[54] R. DuPlessis, "Pulsar~." Oct. 2021, Accessed: Nov. 04, 2021. [Online]. Available: https://github.com/rodneydup/pd-pulsar.

[55] D. Harmon, "Story Structure 101: Super Basic Shit," *Channel 101 Wiki*. Accessed: Nov. 09, 2021. [Online]. Available: https://channel101.fandom.com/wiki/Story_Structure_101:_Super_Basic_Shit.

[56] J. Alison, *Meander, Spiral, Explode: Design and Pattern in Narrative*. New York: Catapult, 2019.

[57] J. H. Murray, *Hamlet on the Holodeck: The Future of Narrative in Cyberspace*. Cambridge, MA, USA: MIT Press, 1998.

[58] B. Almén, *A Theory of Musical Narrative*. Indiana University Press, 2008.

[59] C. Roads, J. Kilgore, and R. DuPlessis, "Architecture for Real-Time Granular Synthesis: EmissionControl2," *Manuscript submitted for publication*, p. 27, 2021.

[60] C. Roads, "Sound Composition with Pulsars," *Journal of the Audio Engineering Society*, vol. 49, no. 3, pp. 134–147, Mar. 2001, Accessed: Mar. 26, 2021. [Online]. Available: https://www.aes.org/e-lib/browse.cfm?elib=10198.

[61] D. M. Fradkin, "Three-Dimensional Isotropic Harmonic Oscillator and SU3," *American Journal of Physics*, vol. 33, no. 3, pp. 207–211, Mar. 1965, doi: 10.1119/1.1971373.

[62] C. Roads, *Composing Electronic Music: A New Aesthetic*, 1st edition. Oxford ; New York: Oxford University Press, 2015.

[63] C. Keysers, "Mirror neurons," *Current Biology*, vol. 19, no. 21, pp. R971–R973, Nov. 2009, doi: 10.1016/j.cub.2009.08.026.

[64] P. F. Ferrari and G. Rizzolatti, "Mirror neuron research: The past and the future," *Philosophical Transactions of the Royal Society B: Biological Sciences*, vol. 369, no. 1644, Jun. 2014, doi: 10.1098/rstb.2013.0169.

[65] H. Chapin, K. Jantzen, J. A. Scott Kelso, F. Steinberg, and E. Large, "Dynamic Emotional and Neural Responses to Music Depend on Performance Expression and Listener Experience," *PLoS ONE*, vol. 5, no. 12, p. e13812, Dec. 2010, doi: 10.1371/journal.pone.0013812.

[66] V. Gazzola, L. Aziz-Zadeh, and C. Keysers, "Empathy and the Somatotopic Auditory Mirror System in Humans," *Current Biology*, vol. 16, no. 18, pp. 1824–1829, Sep. 2006, doi: 10.1016/j.cub.2006.07.072.

[67] M. Johnson, *The body in the mind: The bodily basis of meaning, imagination, and reason*. Chicago, IL, US: University of Chicago Press, 1987.

[68] G. Lakoff and R. Nuñez, *Where Mathematics Come From: How The Embodied Mind Brings Mathematics Into Being*, 1st Printing edition. New York, NY: Basic Books, 2001.

[69] S. Roddy and D. Furlong, "Embodied Aesthetics in Auditory Display," *Organised Sound*, vol. 19, no. 1, pp. 70–77, Apr. 2014, doi: 10.1017/S1355771813000423.

Appendices

A QHOSYN code

This appendix contains all of the code for QHOSYN. It is divided into 5 files. Note that QHOSYN depends on several libraries including Allolib, Native File Dialog, FFTW, and the GSL (GNU Scientific Library). These must be linked in order to compile. CMAKE is the easiest method to manage linking these libraries and compiling an executable. For the complete project with build scripts and realease files for various platforms, you can visit the git repository hosted at (as of December 2021): https://github.com/rodneydup/QHOSYN

A.1 QHOSYN.hpp

```
/*
Quantum Harmonic Oscillator Synthesizer (QHOSYN)
By Rodney DuPlessis (2021)
*/

#define __STDCPP_WANT_MATH_SPEC_FUNCS__ 1

// C libraries
#include <complex.h>
#include <time.h>

#include <fstream>
#include <functional>
#include <iomanip>
#include <iostream>
#include <random>
#include <vector>

// Native File Dialog
#include "../external/nativefiledialog/src/include/nfd.h"

// FFTW
#include <fftw3.h>

// Allolib
#include "Gamma/Filter.h"
#include "Gamma/tbl.h"
#include "al/app/al_App.hpp"
#include "al/graphics/al_Shapes.hpp"
#include "al/io/al_MIDI.hpp"
#include "al/ui/al_ControlGUI.hpp"
#include "al/ui/al_Parameter.hpp"
#include "al_ext/soundfile/al_OutputRecorder.hpp"

// QHOSYN
#include "shadercode.hpp"
#include "wavefunction.hpp"

using namespace al;

class QHOSYN : public App, public MIDIMessageHandler {
 public:
  /**
   * @brief Initilialize the synth interface.
   */
  virtual void onInit() override;

  /**
   * @brief Run once on starup.
   */
  virtual void onCreate() override;
```

```
52
53  /**
54   * @brief Audio rate processing of synth.
55   */
56  virtual void onSound(al::AudioIOData& io) override;
57
58  virtual void onAnimate(double dt) override;
59
60  /**
61   * @brief Draw rate processing of synth interface.
62   */
63  virtual void onDraw(al::Graphics& g) override;
64
65  virtual void onResize(int w, int h) override;
66
67  /**
68   * @brief Do something when a key is pressed
69   */
70  virtual bool onKeyDown(al::Keyboard const& k) override;
71
72  void onMessage(osc::Message& m) override;
73
74  static const int MAX_AUDIO_OUTS = 2;
75
76  // Meshes for visual display
77  Mesh waveFunctionPlot;  // complex valued wave function
78  Mesh probabilityPlot;   // probability distribution
79  Mesh axes;
80  Mesh grid;
81  Mesh samples;
82
83  std::array<Vec3d, 20> measurementPoints;
84  std::array<int, 20> sampleDisplayTimer;
85  int measurementPointsCounter = 0;
86  float automeasureTimer = 0;
87
88  ShaderProgram lineShader;
89  ShaderProgram pointShader;
90
91  Texture lineTexture;
92  Texture pointTexture;
93
94  FBO renderTarget;
95  Texture rendered;
96
97  void updateFBO(int w, int h) {
98    rendered.create2D(w, h);
99    renderTarget.bind();
100   renderTarget.attachTexture2D(rendered);
101   renderTarget.unbind();
102 }
```

```
103
104 // some variables to keep track of states
105 double simTime = 0;
106 float tableReader = 0;
107 bool drawGUI = 1;
108
109 // GUI stuff
110
111 ImGuiWindowFlags flags = ImGuiWindowFlags_NoCollapse | ImGuiWindowFlags_NoMove |
112                          ImGuiWindowFlags_NoResize | ImGuiWindowFlags_NoSavedSettings |
113                          ImGuiWindowFlags_AlwaysAutoResize | ImGuiCond_Once;
114
115 // simulation parameters
116 ParameterInt dims{"Eigenstates", "", 2, "", 1, 15};
117 ParameterBool coeffList{"Manual Coefficient Entry", "", 0};
118 ParameterMenu presetFuncs{"Function Presets"};
119 ParameterBool project{"Project in Orthogonal Basis", "", 1};
120 Parameter simSpeed{"Simulation Speed", 1.0, -5.0, 5.0};
121
122 // audio parameters
123 ParameterBool audioOn{"Audio On", "", 0};
124 ParameterBool panner{"Panner", "", 0};
125 bool pannerTrigger[2] = {0, 0};
126 Parameter wavetableFreq{"Frequency", 200, 10, 1000};
127 ParameterInt ifftBin{"Start Bin", "", 1, "Bin ", 1, fftSize / 4};
128 ParameterInt ifftBandwidth{"Bandwidth", "", 1, "", 1, 16};
129 Parameter volume{"Volume", 0.0, 0, 1};
130 ParameterMenu sourceOneMenu{"Channel 1 source"};
131 ParameterMenu sourceTwoMenu{"Channel 2 source"};
132 SmoothValue<float> pan[2];
133
134 // drawing parameters
135 ParameterBool drawGrid{"Grid", "", 1};
136 ParameterBool drawAxes{"Axes", "", 1};
137 ParameterBool drawWavefunction{"Wave function", "", 1};
138 ParameterBool drawProbability{"Probability", "", 1};
139 ParameterBool drawIfft{"Inverse Fourier Waveform", "", 1};
140 ParameterBool drawNoise{"Noise Waveform", "", 1};
141 ParameterBool drawMeasurements{"Measurements", "", 1};
142
143 // OSC parameters
144 ParameterBool oscSenderOn{"OSC Sender On", "OSC", 0};
145 ParameterBool oscReceiverOn{"OSC Receiver On", "OSC", 0};
146 ParameterBool oscWaveform{"Send Waveform", "OSC", 0};
147 ParameterBool oscMeasurement{"Send Measurements", "OSC", 0};
148
149 // Measurement parameters
150 ParameterBool measurementTrigger{"Measure", "", 0};
151 ParameterBool automeasure{"Auto-measure", "", 0};
152 Parameter automeasureInterval{"Interval (s)", 0.1, 0.016, 2};
153
```

```
  gam::Biquad<> filter[2];
  gam::Biquad<> antiAlias[2];
  ParameterBool filterOn{"Filter", "", 1};

  // custom wrapper for double precision imgui sliders
  bool SliderDouble(const char* label, double* v, double v_min, double v_max,
,                   const char* format = NULL, float power = 1.0f) {
    return ImGui::SliderScalar(label, ImGuiDataType_Double, v, &v_min, &v_max, format,
                               power);
  }

  // wave function for generating coefficients
  // std::function has some undesirable overhead, I'd rather use a function pointer or
  // lambda, but if I ever want to capture some member variable to factor into the
  // function, (for example to do something like "x = 1 for highest eigenstate only" I
  // need std::function.
  std::function<double(double x)> initWaveFunction = [&](double x) { return sin(x); };

  // Create Hilbert basis with 15 dimensions
  HilbertSpace basis{15};

  // initialize the wave function
  WaveFunction psi{&basis, 2, initWaveFunction};

  // copy the coefficients into our variable for manual coefficient control
  std::vector<double> manualCoefficients = psi.coefficients;

  // size of all our arrays
  static const int resolution = 256;

  // range of position values to report for visualization and sonification {-5,5}
  std::vector<double> posValues = linspace<double>(-5, 5, resolution);

  // array to store results from looking up wavefunction values
  std::array<std::complex<double>, resolution> psiValues;

  // These arrays store values for copying to the audio wavetable
  std::array<float, resolution> reValues;
  std::array<float, resolution> imValues;
  std::array<float, resolution> probValues;
  std::array<float, resolution> emptyTable;

  // // ifft
  int sourceSelect[2] = {0, 0};
  static const int fftSize = resolution * 32;
  float binSize = 48000 / fftSize;

  // // fftw inverse fft
  fftw_complex* c2rIn;
  double* c2rOut;
  fftw_plan c2r;
```

```
205  static const int M = resolution * 8;
206  std::array<float, M> filterKernelWindow;
207
208  double* r2cIn;
209  fftw_complex* r2cOut;
210  fftw_plan r2c;
211
212  // // fftw forward fft
213  fftw_complex* c2cForwardIn;
214  fftw_complex* c2cForwardOut;
215  fftw_plan c2cForward;
216  Mesh ifftPlot;
217  Mesh noisePlot;
218
219  static const int bufferLen = fftSize / 4;
220  std::array<std::array<std::array<float, bufferLen>, 4>, 2> ifftBuffers;
221  std::array<float, bufferLen> overlapAddWindow;
222  int fftBufferLenDiff = fftSize - bufferLen;
223
224  int currentIfftBuffer[2] = {0, 0};
225  int currentIfftSample[2] = {0, 0};
226
227  // audio wavetable
228  // output samples
229  float sample[2] = {0, 0};
230  std::array<std::array<float, resolution>, 2> wavetable;
231  std::array<float, resolution>* source[2] = {&reValues, &imValues};
232
233  // // mutex lock to avoid audio thread grabbing a half-written wavetable
234  // std::mutex wavetableLock;
235
236  // RNG to use in weighted distribution (to take a measurement of wavefunction)
237  // I checked that this worked by taking many results and plotting it (July 20, 2021)
238  std::random_device randomDevice;
239  std::mt19937 gen{randomDevice()};
240  std::normal_distribution<> norm{0, 1};
241  std::uniform_real_distribution<> uniform{0, 1000};
242
243  // OSC stuff
244  std::unique_ptr<osc::Send> oscSender;     // create an osc Sender
245  int oscSenderPort = 16447;                // osc port
246  std::string oscSenderAddr = "127.0.0.1";  // ip address
247
248  std::unique_ptr<al::osc::Recv> oscReceiver;             // create an osc Receiver
249  int oscReceiverPort = 16448;                            // osc port
250  std::string oscReceiverAddr = "127.0.0.1";              // ip address
251  int previousoscReceiverPort = oscReceiverPort;          // osc port
252  std::string previousoscReceiverAddr = oscReceiverAddr;  // ip address
253  float oscReceiverTimeout = 0.02;
254
255  std::string measurementArg = "/measurement";
```

```
ImGuiInputTextCallback inputTextCallback;
void* CallbackUserData;

bool isOscWarningWindow = false;

void resetOSC() {
  if (oscReceiver != nullptr) oscReceiver->stop();
  oscReceiver.reset();
  oscReceiver = std::make_unique<al::osc::Recv>(
    oscReceiverPort, oscReceiverAddr.c_str(), oscReceiverTimeout);
  if (oscReceiver->isOpen()) {
    oscReceiver->handler(oscDomain()->handler());
    oscReceiver->start();
    std::cout << "OSC Receiver Settings:" << std::endl;
    std::cout << "IP Address: " << oscReceiverAddr << std::endl;
    std::cout << "Port: " << oscReceiverPort << std::endl;
    std::cout << "Timeout: " << oscReceiverTimeout << std::endl;
    previousoscReceiverAddr = oscReceiverAddr;
    previousoscReceiverPort = oscReceiverPort;
  } else {
    std::cerr
      << "Could not bind to UDP socket. Is there a server already bound to that port?"
      << std::endl;
    oscReceiverAddr = previousoscReceiverAddr;
    oscReceiverPort = previousoscReceiverPort;
    oscReceiver.reset();
    oscReceiver =
      std::make_unique<al::osc::Recv>(oscReceiverPort, oscReceiverAddr.c_str(), 0.02);
    isOscWarningWindow = true;
  }
  oscSender->open(oscSenderPort, oscSenderAddr.c_str());
  std::cout << "OSC Sender Settings:" << std::endl;
  std::cout << "IP Address: " << oscSenderAddr << std::endl;
  std::cout << "Port: " << oscSenderPort << std::endl;
}
// MIDI stuff

RtMidiIn midiIn;

void onMIDIMessage(const MIDIMessage& m) {
  printf("%s: ", MIDIByte::messageTypeString(m.status()));

  switch (m.type()) {
    case MIDIByte::NOTE_ON:
      if (!m.velocity() == 0) wavetableFreq = bpScale[m.noteNumber() - 36] * 10;
      printf("Note %u, Vel %f", m.noteNumber(), m.velocity());
      break;

    case MIDIByte::NOTE_OFF:
      printf("Note %u, Vel %f", m.noteNumber(), m.velocity());
      break;
```

```
307
308       case MIDIByte::PITCH_BEND:
309         printf("Value %f", m.pitchBend());
310         break;
311
312       // Control messages need to be parsed again...
313       case MIDIByte::CONTROL_CHANGE:
314         printf("%s ", MIDIByte::controlNumberString(m.controlNumber()));
315         switch (m.controlNumber()) {
316           case MIDIByte::MODULATION:
317             printf("%f", m.controlValue());
318             break;
319
320           case MIDIByte::EXPRESSION:
321             printf("%f", m.controlValue());
322             break;
323         }
324         break;
325       default:;
326     }
327
328     // If it's a channel message, print out channel number
329     if (m.isChannelMessage()) {
330       printf(" (MIDI chan %u)", m.channel() + 1);
331     }
332
333     printf("\n");
334
335     // Print the raw byte values and time stamp
336     printf("\tBytes = ");
337     for (unsigned i = 0; i < 3; ++i) {
338       printf("%3u ", (int)m.bytes[i]);
339     }
340     printf(", time = %g\n", m.timeStamp());
341   }
342
343   // for ImGUI variable length string input fields
344   struct InputTextCallback_UserData {
345     std::string* Str;
346     ImGuiInputTextCallback ChainCallback;
347     void* ChainCallbackUserData;
348   };
349
350   static int InputTextCallback(ImGuiInputTextCallbackData* data) {
351     InputTextCallback_UserData* user_data = (InputTextCallback_UserData*)data->UserData;
352     if (data->EventFlag == ImGuiInputTextFlags_CallbackResize) {
353       // Resize string callback
354       // If for some reason we refuse the new length (BufTextLen) and/or capacity
355       // (BufSize) we need to set them back to what we want.
356       std::string* str = user_data->Str;
357       IM_ASSERT(data->Buf == str->c_str());
```

```
      str->resize(data->BufTextLen);
      data->Buf = (char*)str->c_str();
    } else if (user_data->ChainCallback) {
      // Forward to user callback, if any
      data->UserData = user_data->ChainCallbackUserData;
      return user_data->ChainCallback(data);
    }
    return 0;
  };
  bool InputText(const char* label, std::string* str, ImGuiInputTextFlags flags,
,                ImGuiInputTextCallback callback, void* user_data) {
    IM_ASSERT((flags & ImGuiInputTextFlags_CallbackResize) == 0);
    flags |= ImGuiInputTextFlags_CallbackResize;

    InputTextCallback_UserData cb_user_data;
    cb_user_data.Str = str;
    cb_user_data.ChainCallback = callback;
    cb_user_data.ChainCallbackUserData = user_data;
    return ImGui::InputText(label, (char*)str->c_str(), str->capacity() + 1, flags,
                            InputTextCallback, &cb_user_data);
  }

  std::string currentAudioDeviceOut;
  std::array<unsigned int, MAX_AUDIO_OUTS> AudioChanIndexOut;
  int getLeadChannelOut() const { return AudioChanIndexOut[0]; }
  bool isPaused = false;
  double globalSamplingRate = 48000;
  const int BLOCK_SIZE = 1024;
  std::string soundOutput;
  al::OutputRecorder mRecorder;
  nfdresult_t result;
  nfdchar_t* outPath = NULL;

  int getSampleRateIndex() {
    unsigned s_r = (unsigned)globalSamplingRate;
    switch (s_r) {
      case 44100:
        return 0;
      case 48000:
        return 1;
      case 88200:
        return 2;
      case 96000:
        return 3;
      default:
        return 0;
    }
  }
  void setSoundOutputPath(std::string sound_output_path) {
    soundOutput = al::File::conformPathToOS(sound_output_path);
  }
```

```
void setAudioSettings(float sample_rate) {
  globalSamplingRate = sample_rate;
  configureAudio(sample_rate, BLOCK_SIZE, MAX_AUDIO_OUTS, 0);
}
void setOutChannels(int lead_channel, int max_possible_channels) {
  AudioChanIndexOut[0] = lead_channel;
  if (max_possible_channels == 1) {
    for (int i = 1; i < MAX_AUDIO_OUTS; i++) {
      AudioChanIndexOut[i] = lead_channel;
    }
  } else {
    // assert(lead_channel + (consts::MAX_AUDIO_OUTS) < max_possible_channels);
    for (int i = 1; i < MAX_AUDIO_OUTS; i++) {
      AudioChanIndexOut[i] = lead_channel + i;
    }
  }
}

void drawAudioIO(AudioIO* io) {
  struct AudioIOState {
    int currentSr = 1;
    int currentBufSize = 3;
    int currentDeviceOut = 0;
    int currentDeviceIn = 0;
    int currentOut = 1;
    int currentIn = 1;
    int currentMaxOut;
    int currentMaxIn;
    std::vector<std::string> devices;
  };

  auto updateOutDevices = [&](AudioIOState& state) {
    state.devices.clear();
    int numDevices = AudioDevice::numDevices();
    int dev_out_index = 0;
    for (int i = 0; i < numDevices; i++) {
      if (!AudioDevice(i).hasOutput()) continue;

      state.devices.push_back(AudioDevice(i).name());
      if (currentAudioDeviceOut == AudioDevice(i).name()) {
        state.currentDeviceOut = dev_out_index;
        state.currentOut = getLeadChannelOut() + 1;
        state.currentMaxOut = AudioDevice(i).channelsOutMax();
      }
      dev_out_index++;
    }
  };

  static std::map<AudioIO*, AudioIOState> stateMap;
  if (stateMap.find(io) == stateMap.end()) {
    stateMap[io] = AudioIOState();
```

```
460       updateOutDevices(stateMap[io]);
461     }
462     AudioIOState& state = stateMap[io];
463     ImGui::PushID(std::to_string((unsigned long)io).c_str());
464
465     if (io->isOpen()) {
466       std::string text;
467       text += "Output Device: " + state.devices.at(state.currentDeviceOut);
468       text += "\nInput Device: " + state.devices.at(state.currentDeviceIn);
469       text += "\nSampling Rate: " + std::to_string(int(io->fps()));
470       text += "\nBuffer Size: " + std::to_string(io->framesPerBuffer());
471       text += "\nOutput Channels: " + std::to_string(state.currentOut) + ", " +
472               std::to_string(state.currentOut + 1);
473       text += "\nInput Channels: " + std::to_string(state.currentIn) + ", " +
474               std::to_string(state.currentIn + 1);
475       ImGui::Text("%s", text.c_str());
476       if (ImGui::Button("Stop")) {
477         isPaused = true;
478         io->stop();
479         io->close();
480         state.currentSr = getSampleRateIndex();
481       }
482     } else {
483       if (ImGui::Button("Update Devices")) {
484         updateOutDevices(state);
485       }
486
487       ImGui::PushItemWidth(ImGui::GetContentRegionAvail().x);
488       if (ImGui::Combo("Output Device", &state.currentDeviceOut,
489                        ParameterGUI::vector_getter, static_cast<void*>(&state.devices),
490                        state.devices.size())) {
491         state.currentMaxOut =
492           AudioDevice(state.devices.at(state.currentDeviceOut), AudioDevice::OUTPUT)
493             .channelsOutMax();
494       }
495       std::string chan_label_out =
496         "Select Outs: (Up to " + std::to_string(state.currentMaxOut) + " )";
497       ImGui::Text(chan_label_out.c_str(), "%s");
498       // ImGui::SameLine();
499       // ImGui::Checkbox("Mono/Stereo", &isStereo);
500       // ImGui::Indent(25 * fontScale);
501       // ImGui::PushItemWidth(50 * fontScale);
502       ImGui::DragInt("Chan 1 out", &state.currentOut, 1.0f, 0, state.currentMaxOut - 1,
503                      "%d", 1 << 4);
504
505       if (state.currentOut > state.currentMaxOut - 1)
506         state.currentOut = state.currentMaxOut - 1;
507       if (state.currentOut < 1) state.currentOut = 1;
508
509       ImGui::PushStyleVar(ImGuiStyleVar_Alpha, ImGui::GetStyle().Alpha * 0.5f);
510       for (int i = 1; i < MAX_AUDIO_OUTS; i++) {
```

```
511         ImGui::SameLine();
512         int temp = state.currentOut + i;
513         std::string channel = "Chan " + std::to_string(i + 1);
514         ImGui::DragInt(channel.c_str(), &temp, 1.0f, 0, state.currentMaxOut, "%d",
515                        1 << 4);
516       }
517       ImGui::PopStyleVar();
518
519       // ImGui::Unindent(25 * fontScale);
520       ImGui::PopItemWidth();
521
522       std::vector<std::string> samplingRates{"44100", "48000", "88200", "96000"};
523       ImGui::Combo("Sampling Rate", &state.currentSr, ParameterGUI::vector_getter,
524                    static_cast<void*>(&samplingRates), samplingRates.size());
525       if (ImGui::Button("Start")) {
526         globalSamplingRate = std::stof(samplingRates[state.currentSr]);
527         io->framesPerSecond(globalSamplingRate);
528         io->framesPerBuffer(BLOCK_SIZE);
529         io->deviceOut(
530           AudioDevice(state.devices.at(state.currentDeviceOut), AudioDevice::OUTPUT));
531         currentAudioDeviceOut = state.devices.at(state.currentDeviceOut);
532         setOutChannels(state.currentOut - 1, state.currentMaxOut);
533         io->open();
534         io->start();
535         isPaused = false;
536       }
537       ImGui::SameLine();
538     }
539     ImGui::PopID();
540   }
541
542   void drawRecorderWidget(al::OutputRecorder* recorder, double frameRate,
543 ,                         uint32_t numChannels, std::string directory,
544 ,                         uint32_t bufferSize) {
545     struct SoundfileRecorderState {
546       bool recordButton;
547       bool overwriteButton;
548     };
549     static std::map<SoundFileBufferedRecord*, SoundfileRecorderState> stateMap;
550     if (stateMap.find(recorder) == stateMap.end()) {
551       stateMap[recorder] = SoundfileRecorderState{0, false};
552     }
553     SoundfileRecorderState& state = stateMap[recorder];
554     ImGui::PushID(std::to_string((unsigned long)recorder).c_str());
555     ImGui::Text("Output File Name:");
556     static char buf1[64] = "test.wav";
557     ImGui::PushItemWidth(ImGui::GetContentRegionAvail().x - 10.0f);
558     ImGui::InputText("##Record Name", buf1, 63);
559     ImGui::PopItemWidth();
560
561     if (state.recordButton) {
```

```
562       ImGui::PushStyleColor(ImGuiCol_Button, ImVec4(0.9, 0.3, 0.3, 1.0));
563       ImGui::PushStyleColor(ImGuiCol_ButtonHovered, ImVec4(0.8, 0.5, 0.5, 1.0));
564       ImGui::PushStyleColor(ImGuiCol_Text, ImVec4(1.0, 1.0, 1.0, 1.0));
565     }
566     std::string buttonText = state.recordButton ? "Stop" : "Record";
567     bool recordButtonClicked = ImGui::Button(buttonText.c_str());
568     if (state.recordButton) {
569       ImGui::PopStyleColor();
570       ImGui::PopStyleColor();
571       ImGui::PopStyleColor();
572     }
573     if (recordButtonClicked) {
574       state.recordButton = !state.recordButton;
575       if (state.recordButton) {
576         uint32_t ringBufferSize;
577         if (bufferSize == 0) {
578           ringBufferSize = 8192;
579         } else {
580           ringBufferSize = bufferSize * numChannels * 4;
581         }
582         std::string filename = buf1;
583         if (!state.overwriteButton) {
584           int counter = 1;
585           while (File::exists(directory + filename) && counter < 9999) {
586             filename = buf1;
587             int lastDot = filename.find_last_of(".");
588             filename = filename.substr(0, lastDot) + "_" + std::to_string(counter++) +
589                        filename.substr(lastDot);
590           }
591         }
592         if (!recorder->start(directory + filename, frameRate, numChannels, ringBufferSize,
593                              gam::SoundFile::WAV, gam::SoundFile::FLOAT)) {
594           std::cerr << "Error opening file for record" << std::endl;
595         }
596       } else {
597         recorder->close();
598       }
599     }
600     ImGui::SameLine();
601     ImGui::Checkbox("Overwrite", &state.overwriteButton);
602     ImGui::Text("Writing to:");
603     ImGui::TextWrapped("%s", directory.c_str());
604 
605     ImGui::PopID();
606   }
607 
608   virtual void onExit() {
609     // destroy inverse fftw things
610     fftw_destroy_plan(c2r);
611     fftw_free(c2rIn);
612     fftw_free(c2rOut);
```

```
613
614     // destroy forward fftw things
615     fftw_destroy_plan(r2c);
616     fftw_free(r2cIn);
617     fftw_free(r2cOut);
618
619     // destroy forward fftw things
620     fftw_destroy_plan(c2cForward);
621     fftw_free(c2cForwardIn);
622     fftw_free(c2cForwardOut);
623   };
624 };
```

A.2 QHOSYN.cpp

```cpp

/*
Quantum Harmonic Oscillator Synthesizer (QHOSYN)
By Rodney DuPlessis (2021)
*/

#define __STDCPP_WANT_MATH_SPEC_FUNCS__ 1

#include "QHOSYN.hpp"

void QHOSYN::onInit() {
  title("QHOSYN");
  audioIO().setStreamName("QHOSYN");

  audioIO().deviceOut(AudioDevice::defaultOutput());
  setAudioSettings(1);

  // audioIO().append(mRecorder);
  srand(std::time(0));
  emptyTable.fill(0);

  imguiInit();
  ImGui::GetIO().IniFilename = NULL;

  // Set up parameters
  dims.registerChangeCallback([&](int x) {
    manualCoefficients.resize(x);
    if (coeffList)
      psi.newHilbertSpace(&basis, x, NULL, manualCoefficients);
    else {
      psi.newHilbertSpace(&basis, x, initWaveFunction, {}, project);
    }
    manualCoefficients = psi.coefficients;

    for (int i = 0; i < manualCoefficients.size(); i++) {
      std::string coeffName = "C_n " + std::to_string(i);
      std::cout << coeffName << " = " << psi.coefficients[i] << std::endl;
    }
  });

  coeffList.registerChangeCallback([&](bool x) {
    if (x) {
      psi.newHilbertSpace(&basis, dims, NULL, manualCoefficients);
    } else {
      bool project = 1;
      if (presetFuncs.get() == 1) project = 0;
      psi.newHilbertSpace(&basis, dims, initWaveFunction, {}, project);
    }
    for (int i = 0; i < manualCoefficients.size(); i++) {
      std::string coeffName = "C_n " + std::to_string(i);
      std::cout << coeffName << " = " << psi.coefficients[i] << std::endl;
```

```
  }
});

presetFuncs.setElements({"psi = sin(x)", "psi = 1 if x=dimensions; 0 otherwise",
                         "psi = random(0 to 1)", "psi = x - 2", " psi = 1/(x+1)"});
presetFuncs.registerChangeCallback([&](int choice) {
  switch (choice) {
    case 0:
      initWaveFunction = [](double x) { return sin(x); };
      break;
    case 1:
      initWaveFunction = [&](double x) {
        if (abs(x) == 10)
          return 1.0;
        else
          return 0.0;
      };
      break;
    case 2:
      initWaveFunction = [](double x) { return double(rand()) / (RAND_MAX / 2) - 1; };
      break;
    case 3:
      initWaveFunction = [](double x) { return x - 2; };
      break;
    case 4:
      initWaveFunction = [](double x) { return 1.0 / (x + 1); };
      break;

    default:
      break;
  }
  psi.newHilbertSpace(&basis, dims, initWaveFunction, {}, project);
});

project.registerChangeCallback([&](bool on) {
  if (on)
    psi.newHilbertSpace(&basis, dims, initWaveFunction, {}, 1);
  else
    psi.newHilbertSpace(&basis, dims, initWaveFunction, {}, 0);
});

sourceOneMenu.setElements({"none", "Real Wavetable", "Imaginary Wavetable",
                           "Probability Wavetable", "Probability Noise Band",
                           "Inverse Fourier Transform"});
sourceOneMenu.registerChangeCallback([&](int x) {
  sourceSelect[0] = 0;
  switch (x) {
    case 0:
      source[0] = &emptyTable;
      break;
    case 1:
```

```
      source[0] = &reValues;
      break;
    case 2:
      source[0] = &imValues;
      break;
    case 3:
      source[0] = &probValues;
      break;
    case 4:
      sourceSelect[0] = 4;
      break;
    case 5:
      sourceSelect[0] = 5;
      break;
    default:
      break;
  }
});
sourceOneMenu.set(1);

sourceTwoMenu.setElements({"none", "Real Wavetable", "Imaginary Wavetable",
                           "Probability Wavetable", "Probability Noise Band",
                           "Inverse Fourier Transform"});
sourceTwoMenu.registerChangeCallback([&](int x) {
  sourceSelect[1] = 0;
  switch (x) {
    case 0:
      source[1] = &emptyTable;
      break;
    case 1:
      source[1] = &reValues;
      break;
    case 2:
      source[1] = &imValues;
      break;
    case 3:
      source[1] = &probValues;
      break;
    case 4:
      sourceSelect[1] = 4;
      break;
    case 5:
      sourceSelect[1] = 5;
      break;
    default:
      break;
  }
});
sourceTwoMenu.set(2);

// Check for connected MIDI devices
```

```
154   if (midiIn.getPortCount() > 0) {
155     // Bind ourself to the RtMidiIn object, to have the onMidiMessage()
156     // callback called whenever a MIDI message is received
157     MIDIMessageHandler::bindTo(midiIn);
158 
159     // Open the last device found
160     unsigned int port = midiIn.getPortCount() - 1;
161     midiIn.openPort(port);
162     printf("Opened port to %s\n", midiIn.getPortName(port).c_str());
163   } else {
164     printf("Error: No MIDI devices found.\n");
165   }
166 
167   std::cout << "onInit() - All domains have been initialized " << std::endl;
168 }
169 
170 void QHOSYN::onCreate() {
171   // Camera setup
172   navControl().useMouse(false);
173   nav().pos(Vec3f(1, 1, 12));
174   nav().faceToward(Vec3f(0.0, 0.0, 0.0));
175   nav().setHome();
176 
177   // OSC setup
178   oscSender = std::make_unique<osc::Send>();
179   resetOSC();
180 
181   // set up drawing meshes
182   waveFunctionPlot.primitive(Mesh::LINE_STRIP);
183   probabilityPlot.primitive(Mesh::LINE_STRIP);
184   noisePlot.primitive(Mesh::LINE_STRIP);
185   ifftPlot.primitive(Mesh::LINE_STRIP);
186   axes.primitive(Mesh::LINES);
187   axes.vertex(-5, 0, 0);
188   axes.vertex(5, 0, 0);
189   axes.vertex(0, 5, 0);
190   axes.vertex(0, -5, 0);
191   axes.vertex(0, 0, 5);
192   axes.vertex(0, 0, -5);
193   axes.generateNormals();
194 
195   grid.primitive(Mesh::LINES);
196   for (int i = -5; i <= 5; i++) {
197     // back grid
198     grid.vertex(i, -5, -5);
199     grid.vertex(i, 5, -5);
200     grid.vertex(-5, i, -5);
201     grid.vertex(5, i, -5);
202     // left grid
203     grid.vertex(-5, -5, i);
204     grid.vertex(-5, 5, i);
```

```
    grid.vertex(-5, i, -5);
    grid.vertex(-5, i, 5);
    // bottom grid
    grid.vertex(-5, -5, i);
    grid.vertex(5, -5, i);
    grid.vertex(i, -5, -5);
    grid.vertex(i, -5, 5);
    for (int i = 0; i < 12; i++) {
      grid.texCoord(0.8, 0.0);
      grid.color(1.0, 1.0, 1.0, 0.2);
    }
  }
  for (int i = 0; i < 6; i++) {
    axes.texCoord(0.8, 0.0);
    axes.color(1.0, 1.0, 1.0, 1.0);  // add color for each vertex
  }
  for (int i = 0; i < resolution; i++) {
    waveFunctionPlot.texCoord(1.0, 0.0);
    probabilityPlot.texCoord(1.0, 0.0);
    waveFunctionPlot.color(1.0, 0.0, 0.0, 0.9);  // add color for each vertex
    probabilityPlot.color(0.0, 0.0, 1.0, 0.9);   // add color for each vertex
  }
  for (int i = 0; i < bufferLen; i++) {
    ifftPlot.texCoord(1.0, 0.0);
    noisePlot.texCoord(1.0, 0.0);
    ifftPlot.color(0.0, 1.0, 1.0, 0.9);
    noisePlot.color(1.0, 1.0, 0.0, 0.9);
  }
  for (int i = 0; i < 20; i++) {
    samples.texCoord(1.0, 0.0);
    samples.color(1.0, 1.0, 1.0, 1.0);
  }

  samples.primitive(Mesh::POINTS);

  // compile and link shaders
  pointShader.compile(shader::pointVertex, shader::pointFragment, shader::pointGeometry);
  lineShader.compile(shader::lineVertex, shader::lineFragment, shader::lineGeometry);

  // Textures for fuzzy lines and points (thanks Karl Yerkes)
  pointTexture.create2D(512, 512, Texture::R8, Texture::RED, Texture::SHORT);
  int Nx = pointTexture.width();
  int Ny = pointTexture.height();
  std::vector<short> alpha;
  alpha.resize(Nx * Ny);
  for (int j = 0; j < Ny; ++j) {
    float y = float(j) / (Ny - 1) * 2 - 1;
    for (int i = 0; i < Nx; ++i) {
      float x = float(i) / (Nx - 1) * 2 - 1;
      float m = exp(-13 * (x * x + y * y));
      m *= pow(2, 15) - 1;  // scale by the largest positive short int
```

```
      alpha[j * Nx + i] = m;
    }
  }
  pointTexture.submit(&alpha[0]);

  lineTexture.create1D(512, Texture::R8, Texture::RED, Texture::SHORT);
  std::vector<short> beta;
  beta.resize(lineTexture.width());
  for (int i = 0; i < beta.size(); ++i) {
    beta[i] = alpha[256 * beta.size() + i];
  }
  lineTexture.submit(&beta[0]);

  // fill windowing arrays
  gam::tbl::hann(&overlapAddWindow[0], overlapAddWindow.size());
  gam::tbl::hamming(&filterKernelWindow[0], filterKernelWindow.size());

  // inverse fftw
  c2rIn = (fftw_complex*)fftw_malloc(sizeof(fftw_complex) * ((fftSize / 2) + 1));
  c2rOut = (double*)fftw_malloc(sizeof(double) * fftSize);
  c2r = fftw_plan_dft_c2r_1d(fftSize, c2rIn, c2rOut, FFTW_ESTIMATE);

  // forward fftw
  r2cIn = (double*)fftw_malloc(sizeof(double) * fftSize);
  r2cOut = (fftw_complex*)fftw_malloc(sizeof(fftw_complex) * ((fftSize / 2) + 1));
  r2c = fftw_plan_dft_r2c_1d(fftSize, r2cIn, r2cOut, FFTW_ESTIMATE);

  // zero the buffers
  for (int i = 0; i < ifftBuffers.size(); i++)
    for (int j = 0; j < ifftBuffers[i].size(); j++)
      for (int k = 0; k < ifftBuffers[i][j].size(); k++) ifftBuffers[i][j][k] = 0;

  // FFT bin size
  binSize = audioIO().framesPerSecond() / fftSize;

  // Set filters and smoothvalues
  for (int chan = 0; chan < 2; chan++) {
    pan[chan].changeType("lin");
    pan[chan].setTime(16.0f);
    filter[chan].type(gam::RESONANT);
    filter[chan].res(2);
    antiAlias[chan].type(gam::LOW_PASS);
    antiAlias[chan].res(0.5);
    antiAlias[chan].freq(15000);
  }

  std::cout << "onCreate() - Graphics context now available" << std::endl;
}

void QHOSYN::onAnimate(double dt) {
  simTime += dt * simSpeed;
```

```
  if (automeasure) automeasureTimer += dt;
  nav().faceToward(Vec3f(0.0, 0.0, 0.0));
  waveFunctionPlot.vertices().clear();
  probabilityPlot.vertices().clear();
  samples.vertices().clear();
  ifftPlot.vertices().clear();
  noisePlot.vertices().clear();

  // Update wavefunction
  for (int i = 0; i < resolution; i++) {
    psiValues[i] = psi.evaluate(posValues[i], simTime);
    reValues[i] = psiValues[i].real();
    imValues[i] = psiValues[i].imag();
    probValues[i] = pow(abs(psiValues[i]), 2);
    waveFunctionPlot.vertex(posValues[i], reValues[i], imValues[i]);
    probabilityPlot.vertex(posValues[i], probValues[i], 0);
  }

  // Fourier methods, skipped if not active for efficiency
  for (int chan = 0; chan < 2; chan++)
    if (sourceSelect[chan] >= 4) {
      for (int i = 0; i < fftSize / 2 + 1; i++) {
        c2rIn[i][0] = 0;
        c2rIn[i][1] = 0;
      }
      // Noise band synthesis
      if (sourceSelect[chan] == 4) {
        for (int i = ifftBin; i < ifftBin + (resolution * ifftBandwidth); i++) {
          if (i < fftSize / 2) {
            double phase = uniform(gen);
            float t = (i - ifftBin) / (float)ifftBandwidth;
            float temp;
            float value = ((probValues[floor(t)] * (1.0f - modf(t, &temp))) +
                           (probValues[floor(t) + 1] * modf(t, &temp))) /
                          2.0f;
            c2rIn[i][0] = cos(phase) * value / (resolution / 8) / dims;
            c2rIn[i][1] = sin(phase) * value / (resolution / 8) / dims;
          } else {
            break;
          }
        }
        fftw_execute(c2r);
        for (int i = 0; i < bufferLen; i++) {
          ifftBuffers[chan][(currentIfftBuffer[chan] + 1) % 4][i] = c2rOut[i];
        }
        if (noisePlot.vertices().size() == 0) {
          for (int i = 0; i < bufferLen; i++) {
            noisePlot.vertex(((i / (float)bufferLen) * 10.0f) - 5.0f, c2rOut[i], 0);
          }
        }
        // IFFT Synthesis
```

```
    } else if (sourceSelect[chan] == 5) {
      for (int i = ifftBin; i < ifftBin + (resolution * ifftBandwidth); i++) {
        if (i < fftSize / 2) {
          float t = (i - ifftBin) / (float)ifftBandwidth;
          float temp;
          float reVal = ((reValues[floor(t)] * (1.0f - modf(t, &temp))) +
                         (reValues[floor(t) + 1] * modf(t, &temp))) /
                        2.0f;
          float imVal = ((imValues[floor(t)] * (1.0f - modf(t, &temp))) +
                         (imValues[floor(t) + 1] * modf(t, &temp))) /
                        2.0f;
          c2rIn[i][0] = reVal / (M / 8) / dims;
          c2rIn[i][1] = imVal / (M / 8) / dims;
        }
      }
      fftw_execute(c2r);
      for (int i = 0; i < M; i++) {
        r2cIn[i] = c2rOut[((fftSize - (M / 2)) + i) % fftSize] * filterKernelWindow[i];
      }
      for (int i = M; i < fftSize; i++) {
        r2cIn[i] = 0;
      }
      fftw_execute(r2c);

      c2rIn[0][0] = 0;
      c2rIn[0][1] = 0;
      c2rIn[fftSize / 2][0] = 0;
      c2rIn[fftSize / 2][1] = 0;
      for (int i = 1; i < fftSize / 2; i++) {
        c2rIn[i][0] = r2cOut[i][0] / M;
        c2rIn[i][1] = r2cOut[i][1] / M;
      }
      fftw_execute(c2r);

      for (int i = 0; i < bufferLen; i++) {
        ifftBuffers[chan][(currentIfftBuffer[chan] + 1) % 4][i] = c2rOut[i];
      }
      if (ifftPlot.vertices().size() == 0) {
        for (int i = 0; i < bufferLen; i++) {
          ifftPlot.vertex(((i / (float)bufferLen) * 10.0f) - 5.0f, c2rOut[i], 0);
        }
      }
    }
  }

// Measurements
if (automeasureTimer > automeasureInterval) {
  measurementTrigger = 1;
  automeasureTimer -= automeasureInterval;
}
if (measurementTrigger) {
```

```
409     std::discrete_distribution<> measurement(probValues.begin(), probValues.end());
410     measurementPoints[measurementPointsCounter] =
411       Vec3d(posValues[measurement(gen)], 0, 0);
412     sampleDisplayTimer[measurementPointsCounter] = 20;
413     pannerTrigger[0] = 1;
414     pannerTrigger[1] = 1;
415   }
416   for (int i = 0; i < 20; i++) {
417     if (sampleDisplayTimer[i] > 0) {
418       samples.vertex(measurementPoints[i]);
419       sampleDisplayTimer[i] -= 1;
420     }
421   }
422 
423   // OSC sending
424   if (oscSenderOn) {
425     if (oscWaveform) {
426       osc::Packet p;
427       p.beginMessage("/realValues/firstHalf");
428       for (int i = 0; i < 128; i++) p << (float)reValues[i];
429       p.endMessage();
430       oscSender->send(p);
431       p.clear();
432       p.beginMessage("/realValues/secondHalf");
433       for (int i = 128; i < 256; i++) p << (float)reValues[i];
434       p.endMessage();
435       oscSender->send(p);
436 
437       p.clear();
438       p.beginMessage("/imaginaryValues/firstHalf");
439       for (int i = 0; i < 128; i++) p << (float)imValues[i];
440       p.endMessage();
441       oscSender->send(p);
442       p.clear();
443       p.beginMessage("/imaginaryValues/secondHalf");
444       for (int i = 128; i < 256; i++) p << (float)imValues[i];
445       p.endMessage();
446       oscSender->send(p);
447     }
448     if (oscMeasurement) {
449       if (measurementTrigger) {
450         oscSender->send(measurementArg,
451                         (float)measurementPoints[measurementPointsCounter][0]);
452       }
453     }
454   }
455   measurementPointsCounter = (measurementPointsCounter + 1) % 20;
456   measurementTrigger = 0;
457 }
458 
459 void QHOSYN::onDraw(Graphics& g) {
```

```
460  g.clear();
461  // needed for transparency
462  gl::blending(true);
463  gl::blendMode(GL_SRC_ALPHA, GL_DST_ALPHA, GL_FUNC_ADD);
464  g.lighting(true);
465
466  // Draw lines
467  lineTexture.bind();
468  g.meshColor();
469  g.shader(lineShader);
470  if (drawAxes) g.draw(axes);
471  if (drawGrid) g.draw(grid);
472  if (drawWavefunction) g.draw(waveFunctionPlot);
473  if (drawProbability) g.draw(probabilityPlot);
474  if (drawIfft) g.draw(ifftPlot);
475  if (drawNoise) g.draw(noisePlot);
476  lineTexture.unbind();
477
478  // Draw measurement points
479  pointTexture.bind();
480  g.shader(pointShader);
481  if (drawMeasurements) g.draw(samples);
482  pointTexture.unbind();
483
484  if (drawGUI) {
485    imguiBeginFrame();
486    int yposition = 0;
487
488    // Simulation Control Panel
489    ParameterGUI::beginPanel("Simulation", 0, yposition, flags);
490    ImGui::Text("Simulation Time: %.2fs", simTime);
491    ImGui::PushItemWidth(-120);
492    ParameterGUI::drawParameter(&simSpeed);
493    ParameterGUI::drawParameterInt(&dims, "");
494    ParameterGUI::drawParameterBool(&coeffList, "");
495    // Option to manually change each coefficient via slider
496    if (coeffList) {
497      for (int i = 0; i < manualCoefficients.size(); i++) {
498        std::string coeffName = "C_n " + std::to_string(i);
499        if (SliderDouble((coeffName).c_str(), &manualCoefficients[i], -10, 10)) {
500          psi.newHilbertSpace(&basis, dims, NULL, manualCoefficients);
501        }
502      }
503    } else {
504      ParameterGUI::drawMenu(&presetFuncs);
505      ParameterGUI::drawParameterBool(&project);
506      if (ImGui::CollapsingHeader("Coefficient Values"))
507        for (int i = 0; i < psi.coefficients.size(); i++) {
508          ImGui::Text("coefficient %i: %.10f ", i, psi.coefficients[i]);
509        }
510      ImGui::PopItemWidth();
```

```
511     }
512     yposition += ImGui::GetWindowHeight();
513     ParameterGUI::endPanel();
514
515     // Audio Control Panel
516     ParameterGUI::beginPanel("Audio", 0, yposition, flags);
517     ImGui::PushItemWidth(-120);
518     ParameterGUI::drawParameterBool(&audioOn);
519     ImGui::SameLine();
520     ParameterGUI::drawParameterBool(&panner);
521     ParameterGUI::drawParameter(&volume);
522     ParameterGUI::drawMenu(&sourceOneMenu);
523     ParameterGUI::drawMenu(&sourceTwoMenu);
524     if (!sourceSelect[0] || !sourceSelect[1]) {
525       ImGui::Text("Wavetable Synthesis");
526       ParameterGUI::drawParameter(&wavetableFreq);
527     }
528     if (sourceSelect[0] || sourceSelect[1]) {
529       ImGui::Text("Fourier Synthesis");
530       ParameterGUI::drawParameterInt(&ifftBin, "");
531       ParameterGUI::drawParameterInt(&ifftBandwidth, "");
532     }
533     if (ImGui::CollapsingHeader("Recorder")) {
534       drawRecorderWidget(&mRecorder, audioIO().framesPerSecond(),
535                          audioIO().channelsOut() <= 1 ? 1 : 2, soundOutput, BLOCK_SIZE);
536       if (ImGui::Button("Change Output Path")) {
537         result = NFD_PickFolder(NULL, &outPath);
538
539         if (result == NFD_OKAY) {
540           std::string temp = outPath;
541           setSoundOutputPath(outPath);
542         }
543       }
544     }
545     if (ImGui::CollapsingHeader("Audio IO Settings")) {
546       drawAudioIO(&audioIO());
547     }
548     ImGui::PopItemWidth();
549     yposition += ImGui::GetWindowHeight();
550     ParameterGUI::endPanel();
551
552     // Draw Control Panel
553     ParameterGUI::beginPanel("Draw", 0, yposition, flags);
554     ParameterGUI::drawParameterBool(&drawGrid);
555     ParameterGUI::drawParameterBool(&drawAxes);
556     ParameterGUI::drawParameterBool(&drawWavefunction);
557     ParameterGUI::drawParameterBool(&drawProbability);
558     ParameterGUI::drawParameterBool(&drawIfft);
559     ParameterGUI::drawParameterBool(&drawNoise);
560     ParameterGUI::drawParameterBool(&drawMeasurements);
561     yposition += ImGui::GetWindowHeight();
```

```
562     ParameterGUI::endPanel();
563
564     // OSC Control Panel
565     ParameterGUI::beginPanel("OSC", 0, yposition, flags);
566     ParameterGUI::drawParameterBool(&oscSenderOn);
567     if (oscSenderOn) {
568       ImGui::PushItemWidth(200);
569       if (InputText("Client IP", &oscSenderAddr, ImGuiInputTextFlags_EnterReturnsTrue,
570                     inputTextCallback, CallbackUserData))
571         resetOSC();
572       if (ImGui::InputInt("Client Port", &oscSenderPort, 1, 1,
573                           ImGuiInputTextFlags_EnterReturnsTrue))
574         resetOSC();
575       ParameterGUI::drawParameterBool(&oscWaveform);
576       ParameterGUI::drawParameterBool(&oscMeasurement);
577       InputText("Measure Argument", &measurementArg, ImGuiInputTextFlags_EnterReturnsTrue,
578                 inputTextCallback, CallbackUserData);
579       ImGui::PopItemWidth();
580     }
581     ParameterGUI::drawParameterBool(&oscReceiverOn);
582     if (oscReceiverOn) {
583       ImGui::PushItemWidth(200);
584       if (InputText("Server IP", &oscReceiverAddr, ImGuiInputTextFlags_EnterReturnsTrue,
585                     inputTextCallback, CallbackUserData))
586         resetOSC();
587       if (ImGui::InputInt("Server Port", &oscReceiverPort, 1, 1,
588                           ImGuiInputTextFlags_EnterReturnsTrue))
589         resetOSC();
590       ImGui::PopItemWidth();
591     }
592     yposition += ImGui::GetWindowHeight();
593     ParameterGUI::endPanel();
594
595     // Measurement Control Panel
596     ParameterGUI::beginPanel("Measurement", 0, yposition, flags);
597     measurementTrigger = ImGui::Button("Measure", ImVec2(100, 20));
598     ParameterGUI::drawParameterBool(&automeasure);
599     ImGui::PushItemWidth(-120);
600     ParameterGUI::drawParameter(&automeasureInterval);
601     ImGui::PopItemWidth();
602     yposition += ImGui::GetWindowHeight();
603     ParameterGUI::endPanel();
604
605     // OSC warning window if OSC port fails to connect.
606     if (isOscWarningWindow) {
607       ImGui::OpenPopup("OSC Error");
608     }
609     bool isOSCWarningOpen = true;
610     if (ImGui::BeginPopupModal("OSC Error", &isOSCWarningOpen)) {
611       isOscWarningWindow = false;
612       ImGui::TextColored(ImVec4(0.9, 0.2, 0.2, 0.9), "Warning");
```

```
613       ImGui::Text(
614         "Could not bind to UDP socket. Is there a server already bound to that "
615         "port?");
616       ImGui::EndPopup();
617     }
618
619     imguiEndFrame();
620
621     imguiDraw();
622   }
623 }
624
625 void QHOSYN::onSound(al::AudioIOData& io) {
626   // This is the sample loop
627   while (io()) {
628     if (audioOn) {
629       // Wavetable Synthesis
630       if (sourceSelect[0] <= 3 || sourceSelect[1] <= 3) {
631         filter[0].freq(wavetableFreq * dims);
632         filter[1].freq(wavetableFreq * dims);
633
634         // increment table reader
635         tableReader += wavetableFreq / (io.fps() / resolution);
636         // if table reader gets to the end of the table
637         if (tableReader >= resolution) {
638           // loop
639           tableReader -= resolution;
640           for (int chan = 0; chan < 2; chan++) {
641             // new panning
642             if (panner) {
643               if (pannerTrigger[chan]) {
644                 if (sourceSelect[chan] <= 3) {
645                   pan[chan].setCurrentValue(
646                     (measurementPoints[measurementPointsCounter].x / 10) + 0.5);
647                   pannerTrigger[chan] = false;
648                 }
649               }
650             } else {
651               if (sourceSelect[chan] <= 3) pan[chan].setCurrentValue(chan);
652             }
653
654             // update table
655             wavetable[chan] = *source[chan];
656           }
657         }
658         // linear interpolation for table reads between indices
659         for (int chan = 0; chan < 2; chan++) {
660           int j = floor(tableReader);
661           float x0 = wavetable[chan][j];
662           float x1 = wavetable[chan][(j == (wavetable[chan].size() - 1))
663                                      ? 0
```

```
                                    : j + 1];  // wrap at end of table
        float t = tableReader - j;
        if (sourceSelect[chan] <= 3)
          if (filterOn)
            sample[chan] = filter[chan]((x1 * t) + (x0 * (1 - t))) /
                           2;  // (divided by 2 because it's loud)
          else
            sample[chan] = ((x1 * t) + (x0 * (1 - t))) / 2;
      }
    }
    // Fourier Synthesis
    for (int chan = 0; chan < 2; chan++) {
      if (sourceSelect[chan] >= 4) {
        if (currentIfftSample[chan] >= bufferLen / 2) {
          currentIfftSample[chan] = 0;
          currentIfftBuffer[chan] = (currentIfftBuffer[chan] + 1) % 4;
        }

        float ifftsample =
          ifftBuffers[chan][currentIfftBuffer[chan]][currentIfftSample[chan]] *
            overlapAddWindow[currentIfftSample[chan]] +
          ifftBuffers[chan][(currentIfftBuffer[chan] + 3) % 4]
                     [currentIfftSample[chan] + bufferLen / 2] *
            overlapAddWindow[currentIfftSample[chan] + bufferLen / 2];

        currentIfftSample[chan]++;
        sample[chan] = ifftsample;

        if (panner) {
          if (pannerTrigger[chan])
            if (ifftsample < 0.000001 || ifftsample > -0.000001) {
              pan[chan].setTarget((measurementPoints[measurementPointsCounter].x / 10) +
                                  0.5);
              pannerTrigger[chan] = false;
            }
        } else {
          pan[chan].setTarget(chan);
        }
      }
    }
    for (int chan = 0; chan < 2; chan++) pan[chan].process();
    // output
    io.out(0) = antiAlias[0]((sample[0] * (1.0f - pan[0].getCurrentValue())) +
                             (sample[1] * (1.0f - pan[1].getCurrentValue()))) *
                volume;
    io.out(1) = antiAlias[1]((sample[0] * pan[0].getCurrentValue()) +
                             (sample[1] * pan[1].getCurrentValue())) *
                volume;
  }
 }
}
```

```
715
716 bool QHOSYN::onKeyDown(Keyboard const& k) {
717   switch (k.key()) {
718     case 'g':
719       drawGUI = 1 - drawGUI;
720       break;
721     case 'r':
722       nav().home();
723       break;
724     default:
725       break;
726   }
727   return true;
728 }
729
730 void QHOSYN::onResize(int w, int h) {
731   updateFBO(width(), height());
732   //
733 }
734
735 void QHOSYN::onMessage(osc::Message& m) {
736   // Check that the address and tags match what we expect
737   if (m.addressPattern() == "/measure" && m.typeTags() == "si") {
738     // Extract the data out of the packet
739     std::string str;
740     int val;
741     m >> str >> val;
742
743     if (val == 1) measurementTrigger = 1;
744   }
745 }
746
747 int main() {
748   AudioDevice dev = AudioDevice::defaultOutput();
749   dev.print();
750
751   QHOSYN app;
752   app.start();
753   return 0;
754 }
```

A.3 wavefunction.hpp

```cpp
#define __STDCPP_WANT_MATH_SPEC_FUNCS__ 1

#include <cassert>
#include <cmath>
#include <complex>
#include <cstring>
#include <functional>
#include <iostream>
#include <memory>
#include <stack>

#include "utility.hpp"

using namespace std::complex_literals;

// The state space of the wave function
class HilbertSpace {
 public:
  HilbertSpace(int dimensions, double hamiltonianPotential(double)) {
    v = hamiltonianPotential;
    dim = dimensions;
    calcEigenBasis();
  }

  HilbertSpace(int dimensions) {
    dim = dimensions;
    calcEigenBasis();
  }

  // pi^1/4 / sqrt(2^n * n!)  * e^(-(x^2/2)) * Hn
  double qhoBasis(int n, double x) {
    return (pow(M_PI, 0.25) / sqrt(pow(2, n) * factorial(n))) * exp(pow(x, 2) / -2.0) *
           std::hermite(n, x);
  };

  void calcEigenBasis() {
    for (int i = 0; i < dim; i++)
      qhoBasisApprox.push_back(std::make_unique<SampleFunction>(
        [=](double x) -> double { return qhoBasis(i, x); }, -10, 10, 1024));
  }

  double eigenbasis(int n, float x) { return qhoBasisApprox[n]->lookup(x); }

  template <typename T>
  T eigenvalues(T x) {
    return 0.5 + x;
  }

  std::vector<std::unique_ptr<SampleFunction>> qhoBasisApprox;
  int dim;  // dimensions of the hilbert space (effectively the number of superposed
            // energy states)
```

```
  double (*v)(double);  // hamiltonian operator
};

// The QHO wavefunction
class WaveFunction {
 public:
  WaveFunction(
    HilbertSpace *hs, int dimensions,
    std::function<double(double)> initWaveFunc = [](double x) { return 1.0; },
    std::vector<double> coeff = {}, bool project = 1) {
    newHilbertSpace(hs, dimensions, initWaveFunc, coeff, project);
  }

  // calculate phase
  std::complex<double> phaseFactor(double n, double t) {
    return exp(-1i * hilbertSpace->eigenvalues(n) * t);
  }

  // Lookup value from wavefuntion for time t at position value x, summing over
  // eigenstates
  std::complex<double> wavefunc(double x, double t) {
    std::complex<double> sum = (0, 0);
    for (int n = 0; n < eigenStates; n++)
      sum += coefficients[n] * hilbertSpace->eigenbasis(n, x) * phaseFactor(n, t);
    return sum;
  }

  // calculate normalize value
  void normalize() {
    auto ptr = [=](double x) { return pow(abs(wavefunc(x, 0)), 2); };
    gsl_function_pp<decltype(ptr)> Fp(ptr);
    gsl_function *normFunc = static_cast<gsl_function *>(&Fp);
    gsl_integration_cquad_workspace *normW = gsl_integration_cquad_workspace_alloc(1000);
    gsl_integration_cquad(normFunc, -1, 1, 1.49e-08, 1.49e-08, normW, &integrationResult,
                          NULL, NULL);
    gsl_integration_cquad_workspace_free(normW);
    normF = sqrt(integrationResult);
  }

  // Return wavefunc normalized
  std::complex<double> evaluate(double x, double t) { return wavefunc(x, t) / normF; }

  // get the coefficients from an orthogonal basis projection of the initial function
  std::vector<double> orthogonalBasisProjection(
,    std::function<double(double x)> waveFunc) {
    std::vector<double> tempCoeff;
    tempCoeff.resize(eigenStates);

    gsl_integration_cquad_workspace *orthoW = gsl_integration_cquad_workspace_alloc(1000);
    for (int i = 0; i < tempCoeff.size(); i++) {
      auto ptr = [=](double x) {
```

```
        return (std::conj(hilbertSpace->eigenbasis(i, x)) * waveFunc(x)).real();
      };
      gsl_function_pp<decltype(ptr)> Fp(ptr);
      gsl_function *orthoFunc = static_cast<gsl_function *>(&Fp);
      gsl_integration_cquad(orthoFunc, -10, 10, 1.49e-08, 1.49e-08, orthoW,
                            &integrationResult, NULL, NULL);
      tempCoeff[i] = integrationResult;
    }
    gsl_integration_cquad_workspace_free(orthoW);
    return tempCoeff;
  }

  // instantiate a new hilbert space
  void newHilbertSpace(
,    HilbertSpace *hs, int dimensions,
,    std::function<double(double x)> initWaveFunc = [](double x) { return 1.0; },
,    std::vector<double> coeff = {}, bool project = 1) {
    eigenStates = dimensions;
    hilbertSpace = hs;
    if (!coeff.empty()) {
      coefficients = coeff;
    } else {
      if (project) {
        coefficients = orthogonalBasisProjection(initWaveFunc);
      } else {
        for (int i = 0; i < eigenStates; i++) coefficients[i] = initWaveFunc(i);
      }
    }
    normalize();
  }

  // collapse the wavefunction, would need to find a numerical solution to make this work
  // in real-time and continue evolving.
  void collapse() {}

  int eigenStates;
  double integrationResult;
  double normF;
  HilbertSpace *hilbertSpace;
  std::vector<double> coefficients;
};
```

A.4 utility.hpp

```cpp
#define __STDCPP_WANT_MATH_SPEC_FUNCS__ 1

#include <gsl/gsl_integration.h>
#include <unistd.h>

#include <vector>

// helper functions

int factorial(int x) {
  int result = 1;
  for (int i = 1; i <= x; i++) result = result * i;
  return result;
}

// create a discretized linear space in range {start_in, end_in} with num_in points
template <typename T>
std::vector<T> linspace(T start_in, T end_in, int num_in) {
  std::vector<T> linspaced;

  T start = static_cast<T>(start_in);
  T end = static_cast<T>(end_in);
  T num = static_cast<T>(num_in);

  if (num == 0) {
    return linspaced;
  }
  if (num == 1) {
    linspaced.push_back(start);
    return linspaced;
  }

  T delta = (end - start) / (num - 1);

  for (int i = 0; i < num - 1; ++i) {
    linspaced.push_back(start + delta * i);
  }
  linspaced.push_back(end);  // I want to ensure that start and end
                             // are exactly the same as the input
  return linspaced;
}

// class that discretizes (samples) a function and stores results for lookup later.
class SampleFunction {
 public:
  template <typename T>
  SampleFunction(T f, float min, float max, int numSamples) {
    minX = min;
    maxX = max;
    rangeX = max - min;
    numSamplesX = numSamples;
```

```
52
53      std::vector<double> domain = linspace<double>(minX, maxX, numSamplesX);
54      functionSamples.resize(domain.size());
55      for (int x = 0; x < domain.size(); x++) functionSamples[x] = f(domain[x]);
56    }
57
58    double lookup(float x) {
59      if (isnan(x)) return 0;
60      x = std::min(std::max(x, minX), maxX - 1);
61      x -= minX;
62      x = std::round(x * (numSamplesX / rangeX));
63      return functionSamples[int(x)];
64    }
65
66    int size() { return functionSamples.size(); }
67
68   private:
69    int numSamplesX;
70    float minX, maxX, rangeX;
71    std::vector<double> functionSamples;
72  };
73
74  // some black magic to cast a function pointer in a way that gsl likes
75  template <typename F>
76  class gsl_function_pp : public gsl_function {
77   public:
78    gsl_function_pp(const F &func) : _func(func) {
79      function = &gsl_function_pp::invoke;
80      params = this;
81    }
82
83   private:
84    const F &_func;
85    static double invoke(double x, void *params) {
86      return static_cast<gsl_function_pp *>(params)->_func(x);
87    }
88  };
89
90  // Bohlen-pierce ratios for tuning midi input
91  std::vector<float> bpScale{
92    1.000000,  1.080000,  1.190476,  1.285714,  1.400000,  1.530612,  1.666667,  1.800000,
93    1.960000,  2.142857,  2.333333,  2.520000,  2.777778,  3.000000,  3.240000,  3.571429,
94    3.857143,  4.200000,  4.591837,  5.000000,  5.400000,  5.880000,  6.428571,  7.000000,
95    7.560000,  8.333333,  9.000000,  9.720000,  10.714286, 11.571429, 12.600000, 13.775510,
96    15.000000, 16.200000, 17.640000, 19.285714, 21.000000, 22.680000, 25.000000, 27.000000,
97    29.160000, 32.130000, 34.830000, 37.800000, 41.310000, 45.090000, 48.600000, 52.920000,
98    57.780000, 62.910000, 68.040000, 75.060000, 81.000000};
99
100 // circular buffer with some useful methods
101 template <typename T>
102 class RingBuffer {
```

```
public:
 RingBuffer(unsigned maxSize) : mBufferSize(maxSize) {
   mBuffer.resize(mBufferSize);
   mWriteHead = -1;
   mReadHead = -1;
   mPrevSample = 0;
 }

 // return the size of the RingBuffer
 unsigned size() const { return mBufferSize; }

 // resize the RingBuffer
 void resize(unsigned maxSize) {
   mBufferSize = maxSize;
   mBuffer.resize(mBufferSize);
 }

 // increment the write head and write value at write head
 void push_back(T value) {
   mMutex.lock();
   mWriteHead = (mWriteHead + 1) % mBufferSize;
   mBuffer[mWriteHead] = value;
   mMutex.unlock();
 }

 // increment the write head and write value at write head
 void overwrite(int index, T value) {
   if (index >= mBufferSize) {
     std::cerr << "RingBuffer index out of range." << std::endl;
     index = index % mBufferSize;
   }
   mMutex.lock();
   mBuffer[index] = value;
   mMutex.unlock();
 }

 // return the index of the write head
 unsigned getWritehead() const { return mWriteHead; }

 // return the index of the read head
 unsigned getReadhead() const { return mReadHead; }

 // Read value at index
 T at(unsigned index) {
   if (index >= mBufferSize) {
     std::cerr << "RingBuffer index out of range." << std::endl;
     index = index % mBufferSize;
   }
   if (mMutex.try_lock()) {
     mPrevSample = mBuffer[index];
     mMutex.unlock();
```

```
154     }
155     return mPrevSample;
156   }
157
158   // read value at index and set value to 0.
159   T consume(unsigned index) {
160     if (index >= mBufferSize) {
161       std::cerr << "RingBuffer index out of range." << std::endl;
162       index = index % mBufferSize;
163     }
164     if (mMutex.try_lock()) {
165       mPrevSample = mBuffer[index];
166       mBuffer[index] = 0;
167       mMutex.unlock();
168     }
169     return mPrevSample;
170   }
171
172   // increment read head, read value, and set value to 0.
173   T consume() {
174     mReadHead = (mReadHead + 1) % mBufferSize;
175     if (mMutex.try_lock()) {
176       mPrevSample = mBuffer[mReadHead];
177       mBuffer[mReadHead] = 0;
178       mMutex.unlock();
179     }
180     return mPrevSample;
181   }
182
183   // overridden [] operator to call at()
184   T operator[](unsigned index) { return this->at(index); }
185
186   // overridden () operator to call at() and increment read head
187   T operator()() {
188     mReadHead = (mReadHead + 1) % mBufferSize;
189     return this->at(mReadHead);
190   }
191
192   // return a pointer to the buffer data
193   const T *data() { return mBuffer.data(); }
194
195   // get RMS value over a number of elements before mWriteHead
196   T getRMS(unsigned lookBackLength) {
197     if (lookBackLength > mBufferSize) {
198       std::cerr << "Lookback length must be less than ringbuffer size. Setting lookback "
199                    "to length "
200                    "of ringbuffer"
201                 << std::endl;
202       lookBackLength = mBufferSize;
203     }
204     int start = mWriteHead - lookBackLength;
```

```
    if (start < 0) start = mBufferSize + start;

    T val = 0.0;
    for (unsigned i = 0; i < lookBackLength; i++) {
      val += pow(mBuffer[(start + i) % mBufferSize], 2);
    }
    return sqrt(val / lookBackLength);
  }

  // send buffer values to standard output
  void print() const {
    for (auto i = mBuffer.begin(); i != mBuffer.end(); ++i) std::cout << *i << " ";
    std::cout << "\n";
  }

 private:
  std::vector<T> mBuffer;
  unsigned mBufferSize;
  int mWriteHead;
  int mReadHead;
  T mPrevSample;

  std::mutex mMutex;
};

template <class T>  // The class is templated to allow a variety of data types
class SmoothValue {
 public:
  SmoothValue(
    float initialTime = 50.0f,
    std::string interpolationType =
      "log") {  // how much time, in ms, does it take to arrive (or approach target value)
    arrivalTime = initialTime;  // store the input value (default to 50ms)
    interpolation = interpolationType;
    calculateCoefficients();  // calculate a and b (it is a lowpass filter)
    z = 0.0f;
  }

  inline T process() {  // this function will be called per-sample on the data
    if (stepsTaken <= numSteps) {
      if (interpolation == "log") z = (targetValue * b) + (z * a);
      if (interpolation == "lin") z += linStep;
      stepsTaken += 1;
    }
    return z;  // return the new z value (the output sample)
  }

  void setTime(float newTime) {  // set time (in ms)
    arrivalTime = newTime;       // store the input value
    numSteps = arrivalTime * 0.001f * gam::sampleRate();
    if (interpolation == "log") calculateCoefficients();  // calculate a and b
```

```
  }

  void setTarget(T newTarget) {
    targetValue = newTarget;
    linStep = (targetValue - z) / numSteps;
    stepsTaken = 0;
  }

  // Jump to a value without ramping
  void setCurrentValue(T newValue) {
    targetValue = newValue;
    z = newValue;
    stepsTaken = numSteps;
  }

  // Change between log and lin ramp types
  void changeType(std::string newType) { interpolation = newType; }

  // Get current value without processing a step
  T getCurrentValue() { return z; }

  // Get the current target value
  T getTargetValue() { return targetValue; }

  // Get the current ramp time in milliseconds
  float getTime() { return arrivalTime; }

 private:
  void
  calculateCoefficients() {  // called only when 'setTime' is called (and in constructor)
    a = std::exp(-(M_PI * 2) / numSteps);  // rearranged lpf coeff calculations
    b = 1.0f - a;
  }

  T targetValue;      // what is the destination (of type T, determined by implementation)
  T currentValue;     // how close to the destination? (output value)
  float arrivalTime;  // how long to take
  float a, b;         // coefficients
  T linStep;          // size of step in linear interpolation
  T numSteps;         // number of steps in interpolation
  int stepsTaken;     // variable to keep track of steps to avoid overshooting
  T z;                // storage for previous value
  std::string interpolation;
};
```

A.5 shadercode.hpp

```
// This shader code allows for the fuzzy look
// Thanks Karl Yerkes for the foundation to this.
namespace shader {
const char *lineVertex = R"(
#version 400

layout(location = 0) in vec3 vertexPosition;
layout(location = 1) in vec4 vertexColor;
layout(location = 2) in vec2
    vertexSize;  // as a 2D texture cordinate, but we ignore the y

out Vertex {
  vec4 color;
  float size;
}
vertex;

uniform mat4 al_ModelViewMatrix;
uniform mat4 al_ProjectionMatrix;

void main() {
  gl_Position = al_ModelViewMatrix * vec4(vertexPosition, 1.0);
  vertex.color = vertexColor;
  vertex.size = vertexSize.x;
}
)";

const char *lineGeometry = R"(
#version 400

layout(lines) in;
layout(triangle_strip, max_vertices = 4) out;

uniform mat4 al_ProjectionMatrix;

in Vertex {
  vec4 color;
  float size;
}
vertex[];

out Fragment {
  vec4 color;
  float textureCoordinate;
}
fragment;

void main() {
  mat4 m = al_ProjectionMatrix;  // rename to make lines shorter
  vec4 a = gl_in[0].gl_Position;
  vec4 b = gl_in[1].gl_Position;
```

```
52
53    const float r = 0.03;
54
55    vec4 d = vec4(normalize(cross(b.xyz - a.xyz, vec3(0.0, 0.0, 1.0))), 0.0) * r;
56
57    gl_Position = m * (a + d * vertex[0].size);
58    fragment.color = vertex[0].color;
59    fragment.textureCoordinate = 0.0;
60    EmitVertex();
61
62    gl_Position = m * (a - d * vertex[0].size);
63    fragment.color = vertex[0].color;
64    fragment.textureCoordinate = 1.0;
65    EmitVertex();
66
67    gl_Position = m * (b + d * vertex[1].size);
68    fragment.color = vertex[1].color;
69    fragment.textureCoordinate = 0.0;
70    EmitVertex();
71
72    gl_Position = m * (b - d * vertex[1].size);
73    fragment.color = vertex[1].color;
74    fragment.textureCoordinate = 1.0;
75    EmitVertex();
76
77    EndPrimitive();
78  }
79  )";
80
81  const char *lineFragment = R"(
82  #version 400
83
84  in Fragment {
85    vec4 color;
86    float textureCoordinate;
87  }
88  fragment;
89
90  uniform sampler1D alphaTexture;
91
92  layout(location = 0) out vec4 fragmentColor;
93
94  void main() {
95    float a = texture(alphaTexture, fragment.textureCoordinate).r;
96    if (a < 0.05) discard;
97    fragmentColor = vec4(fragment.color.xyz, fragment.color.a * a);
98  }
99  )";
100
101 const char *pointVertex = R"(
102 #version 400
```

```
103
104 layout(location = 0) in vec3 vertexPosition;
105 layout(location = 1) in vec4 vertexColor;
106 layout(location = 2) in vec2
107     vertexSize;  // as a 2D texture cordinate, but we ignore the y
108
109 uniform mat4 al_ModelViewMatrix;
110 uniform mat4 al_ProjectionMatrix;
111
112 out Vertex {
113   vec4 color;
114   float size;
115 }
116 vertex;
117
118 void main() {
119   gl_Position = al_ModelViewMatrix * vec4(vertexPosition, 1.0);
120   vertex.color = vertexColor;
121   vertex.size = vertexSize.x;
122 }
123 )";
124
125 const char *pointGeometry = R"(
126 #version 400
127
128 // take in a point and output a triangle strip with 4 vertices (aka a "quad")
129 //
130 layout(points) in;
131 layout(triangle_strip, max_vertices = 4) out;
132
133 uniform mat4 al_ProjectionMatrix;
134
135 in Vertex {
136   vec4 color;
137   float size;
138 }
139 vertex[];
140
141 out Fragment {
142   vec4 color;
143   vec2 textureCoordinate;
144 }
145 fragment;
146
147 void main() {
148   mat4 m = al_ProjectionMatrix;   // rename to make lines shorter
149   vec4 v = gl_in[0].gl_Position;  // al_ModelViewMatrix * gl_Position
150
151   float r = 0.15;
152   r *= vertex[0].size;
153
```

```
  gl_Position = m * (v + vec4(-r, -r, 0.0, 0.0));
  fragment.textureCoordinate = vec2(0.0, 0.0);
  fragment.color = vertex[0].color;
  EmitVertex();

  gl_Position = m * (v + vec4(r, -r, 0.0, 0.0));
  fragment.textureCoordinate = vec2(1.0, 0.0);
  fragment.color = vertex[0].color;
  EmitVertex();

  gl_Position = m * (v + vec4(-r, r, 0.0, 0.0));
  fragment.textureCoordinate = vec2(0.0, 1.0);
  fragment.color = vertex[0].color;
  EmitVertex();

  gl_Position = m * (v + vec4(r, r, 0.0, 0.0));
  fragment.textureCoordinate = vec2(1.0, 1.0);
  fragment.color = vertex[0].color;
  EmitVertex();

  EndPrimitive();
}
)";

const char *pointFragment = R"(
#version 400

in Fragment {
  vec4 color;
  vec2 textureCoordinate;
}
fragment;

uniform sampler2D alphaTexture;

layout(location = 0) out vec4 fragmentColor;

void main() {
  float a = texture(alphaTexture, fragment.textureCoordinate).r;
  // if (a < 0.05) discard;
  fragmentColor = vec4(fragment.color.xyz, a);
}
)";
}  // namespace shader
```