

**Final report: Improving the Realism of a Physically
Simulated Musical Instrument**

**Author: Dave Humphreys
Supervisor: Dr. Kirill Sidorov
Moderator: Prof. Ralph Martin**

CM3203: One-semester individual project (40 credits)

c1322278, 12/04/16

Contents

1. Introduction	4
2. Background	5
Early work	
Physical models	6
Sample-based virtual instruments	8
Additive synthesis	9
3. Relevant theory	10
String simulation	
The bowed string	11
Model solutions	12
4. Specification and Design	13
System specification	
Functional design	14
5. Implementation	15
Preparation	
Cello simulation engine	16
<i>String simulation</i>	
<i>Cello body simulation</i>	18
<i>String damping</i>	23
<i>Sympathetic resonance</i>	
<i>Natural variations</i>	24
Spectral comparison	26
<i>Comparison function</i>	
<i>Validity checks</i>	27
Optimiser	29
<i>Construction</i>	
<i>Optimisation methods</i>	31
6. Results and evaluation	32
Test overview	
Objective testing	34
<i>Recordings I</i>	
<i>Recordings II</i>	38
Conclusion (metric comparison)	42
Subjective testing	43
<i>Identification of real sounds</i>	
<i>Simulation preference</i>	44
<i>Conclusion (subjective evaluation)</i>	45
Demonstration audio	
7. Future work	46
Comparison and analysis	
Frequency domain correction	
Realistic transitions	

Optimiser extension	47
Performance	
Realistic input controls	
Finalisation as a virtual instrument	
8. Conclusions	48
9. Reflection	49
Skills and knowledge	
Choices and contributions	
Design challenges	50
<i>Preparation</i>	
<i>Initial trials</i>	51
<i>Attempted solutions</i>	
<i>Optimisation</i>	52
Progress monitoring	53
Goals and achievements	54
Personal strengths and weaknesses	55
Professional development	
Future skills and learning	56
Outcome	
Appendices	57
Appendix A: Source code	
<i>Cello Simulator</i>	
<i>Convolve (construct)</i>	88
<i>FFTv (construct)</i>	89
<i>Impulse Response</i>	91
<i>String Simulation</i>	99
<i>Generate inputSim</i>	128
<i>Compare CQT</i>	131
<i>Optimiser</i>	133
<i>Curve Generator</i>	149
Appendix B: Subjective evaluation software	152
Appendix C: Optimiser output	157
<i>Optimised string parameter settings</i>	
<i>Optimised string damping coefficients</i>	158
<i>Example of search space traversal</i>	162
Appendix D: Development diary	168
Bibliography	173

1. Introduction

One approach to the synthesis of highly realistic musical sounds in virtual instruments is to mathematically describe the physical properties of a real musical instrument and run the simulation of the relevant physical laws on a computer. This approach appears to be far more promising than using playback of pre-recorded sounds of actual instruments (sample libraries) by sampling software, where every sound from each instrument must be recorded for every technique and at every level of loudness required. Many effects, including arbitrary changes in pitch (legato or glissando) and amplitude (crescendo and diminuendo), are not supported by the sampling approach unless they are captured within this finite set of recordings. However a physical model may, in principle, reproduce every possible effect.

A pre-recorded sample set is limited by storage capacity, a requirement which becomes more significant with increased sampling rate and bit depth. One professional sample library, *Berlin Strings* from Schwarzer and Mantik GmbH [25] consumes 268Gb of uncompressed disk space. This is due to the large number of combinations needed to reproduce the many articulations.

The recordings themselves are costly to produce. Hire of an experienced cellist can cost £120 per day [13] and a professional studio £200 per day [1], a formidable outlay considering the volume of samples required. A physical model, on the other hand, requires only the storage capacity needed for the application itself, and does not necessitate the hire of specialist musicians and recording engineers.

However, a physical simulation requires extensive computation, and current physical models do not yet possess the fidelity needed for entirely accurate instrument simulation [26]. Research continues in an effort to discover and model the fine details necessary to replicate the quality of tone a real instrument provides.

This project aims to improve the perceived realism of such a model, with the following contributions:

- A physical model of a cello string was implemented in Matlab.
- Bounded variations in bow force, velocity and finger position were added to represent changes occurring during play from human and physical inaccuracies.
- Convolution vectors were calculated which would transform the string's output to closely match that of real instruments, and replicate the effect of the instrument body.
- Optimisation was employed to tune the model's string diameter, linear density, Young's modulus and damping characteristic to further match that of real instruments.

In experiments, output was found to improve when compared objectively to professional instrument recordings, with this work producing a greater spectral match to real sounds than the original model for every sample tested. In human trials, it was subjectively chosen as more realistic than the original model's output in 62% of cases, and its output classified as real alongside actual recordings 47% of the time.

2. Background

Early work

Analysis by Helmholtz in 1895 identified the periodic components of waves producing the perception of tone [9]. This work applies to the vibrations produced by musical instruments. He called this “analysis of compound into simple vibrations” (*figure 2.1*), and showed that partials, the different components that make up the vibrations conducted to the ear via particles in the air, were instrumental in the perception of tone.

Helmholtz also described the basis of the motion arising from the action of bowing, and using what he termed a “vibration microscope” was able to record and analyse this motion on a violin string, showing that it principally represented a sawtooth pattern (*figure 2.2*).

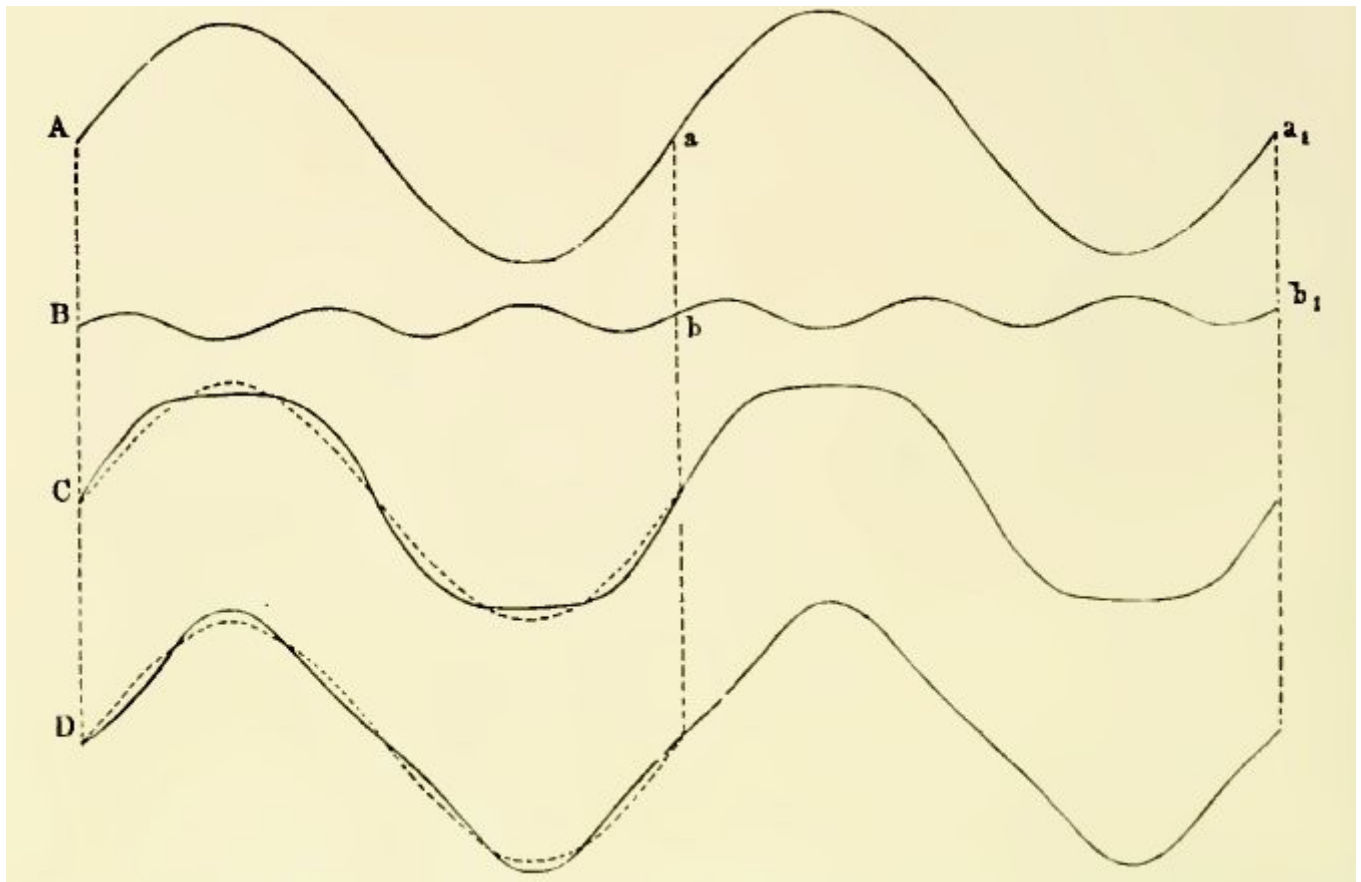


Figure 2.1: Periodic wave motion D is composed of partials A, B and C. [9]

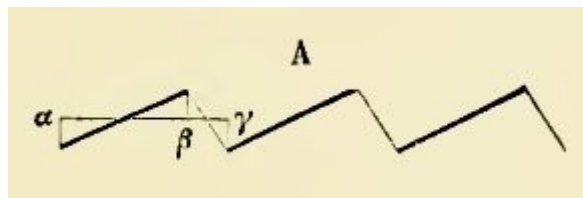


Figure 2.2: Helmholtz identified the “principal motion” of string displacement when bowed. [9]

Physical models

In 1979, research by McIntyre and Woodhouse proposed the development of a model for bowed string simulation which included bow velocity and normal force, and yielded a fast algorithm which would allow the varying of parameters during computation [16]. Within their model, interaction between bow and string is governed by the bow's normal force F_b and velocity v_b relative to the string's surface. From these a relative velocity v is produced, and the resulting friction f calculated from an idealised friction curve (figure 2.3). They concluded that, whilst capable of plausible behaviour, further work was required to include bow width, correct string damping and reflection properties within the model's calculations.

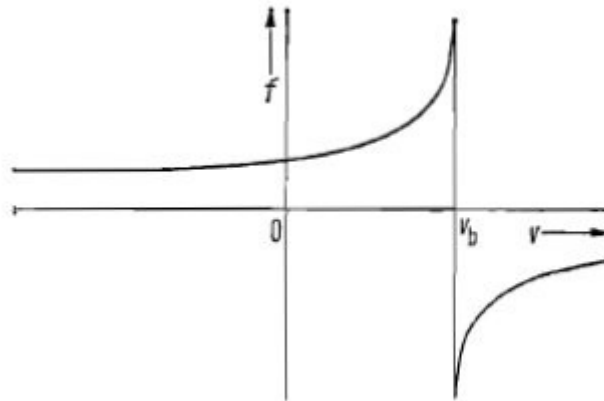


Figure 2.3: The idealised curve used by McIntyre and Woodhouse to calculate bow friction from the relative velocity of bow and string. [16]

Work on such models continues to this day. The *Next Generation Sound Synthesis* group at the University of Edinburgh is a European Research Council funded project exploring instrument simulation. In 2014, their researchers Bilbao and Desvages [2] developed a stringed and bowed instrument implementation whose output, to the casual listener, contains time domain effects reminiscent of a bowed cello. Their model, unlike previous attempts, computes the string displacement in two planes (figure 2.4) and includes string interaction with bow hair, the player's finger and the instrument's neck. The position of the bow relative to its distance from the instrument's bridge is also included in the calculations governing excitation of the string.

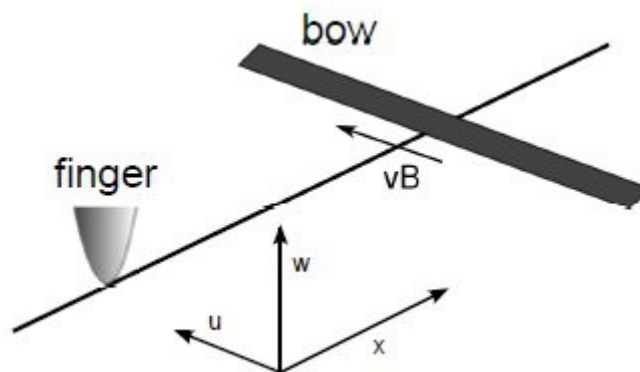


Figure 2.4: Bilbao and Desvages model computes string displacement in two planes, u and w . [2]

Research conducted by Demoucron in 2008 [6] proposed a mode-based model of the bowed string. This model also included bow to bridge distance, bow force, velocity and finger position as input parameters, in a similar manner to Bilbao and Desvages' design. With minor changes, it was used as

a basis for Percival's development of *Artifastring*, a shortened form of “artificial fast string” [20]. It is also capable of producing output containing effects reminiscent of a bowed cello, allows customisation of string parameters, bow force and velocity, and supports neck finger positioning (figure 2.5).

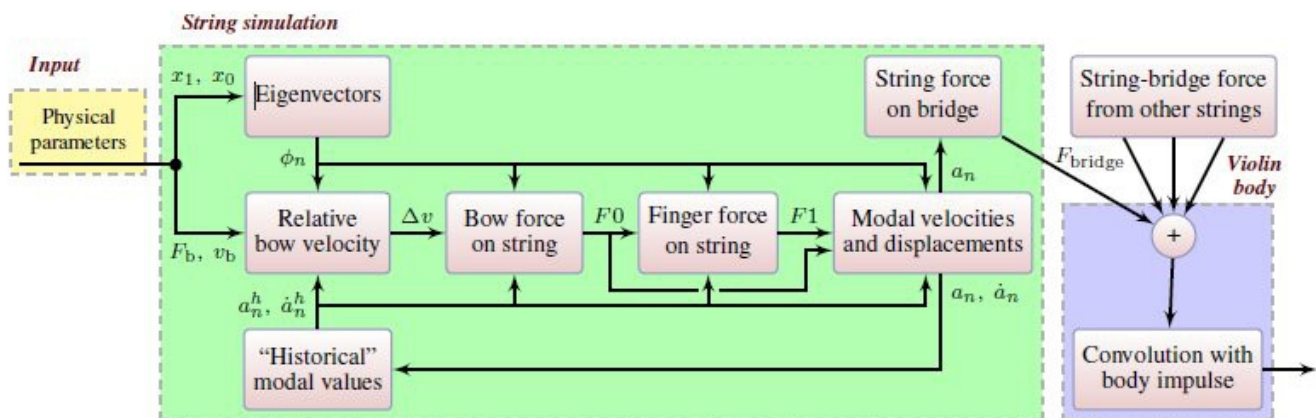


Figure 2.5: The various components present in “Artifastring”, which is based on Demoucron's model. [20]

Some commercial physically modelled instruments currently exist. *Pianoteq* produce the *Pianoteq 5* [22], an entirely physical simulation supporting a choice of several different instrument configurations. It consumes 40Mb of disk space, and retails for \$99 to \$399, depending on the package chosen.

Similarly, *Image Line's Drumaxx* [10] is a physical modelling drum simulator, representing the membrane of each drum as a mesh of individual points, each with its own properties. It requires 30Mb of disk space, and retails for €89.

A physical simulation possesses many advantages, specifically:

- All articulations and effects possible on a real instrument may theoretically be created.
- Model parameters may be altered to match individual instruments, or create new ones.
- No storage of recordings is required, only the software itself.
- Since no recordings are required, hire of musicians and recording personnel is also not required.

However, such a model also has disadvantages:

- Calculation of string displacement is necessary at each output step, which is computationally expensive.
- Fidelity is only as high as the model allows; no entirely accurate real-time stringed instrument models currently exist [26].

Sample-based virtual instruments

Native Instruments Kontakt 5, a “flexible and powerful sampler” [17] is a professional software sampler featuring high quality filters and effects, and support for comprehensive third-party sound libraries. It is used by individuals and music production companies worldwide, and maintaining a high level of realism is a key aspect of its instruments' design.

Various organisations produce sound libraries and virtual instrument patches for use with commercial sampling software such as *Kontakt 5* and the *Vienna Instruments Sample Player* [29], to name a few examples. *Berlin Strings* from Schwarzer and Mantik GmbH is one instance, and supports four legato types, twenty-four round robin spiccato, a collection of short notes, playable runs, five microphone positions, soft, immediate and accented bow strokes and vibrato expressions [25].

The *Vienna Symphonic Library* includes the *Solo Strings Bundle*, a collection of over 149,000 samples from violin, viola, cello and double bass. Supported effects include spiccato, ricochet, col legno, snap pizzicato, tremolo, glissando, trills and harmonics [28]. The *Vienna Instruments Sample Player* supports the library as a playback and performance tool, and the *Solo Strings Bundle* itself consumes 86.7Gb of storage space.

The extensive storage requirements of sample-based virtual instruments may be easily understood by example. Given an average sample duration of 7 seconds at a resolution of 24-bit at 96kHz for 6 simultaneous channels, a single sample consumes >11.5Mb. For a range of 2 octaves per string, supporting 6 effects at 4 amplitude levels and a mere 3 transitions per recording, >6.4Gb of storage is required for a single instrument and microphone position. The *Berlin Strings* and *Solo Strings Bundle* libraries contain a far greater combination of recordings, as may be understood by their considerable storage requirements.

Both of these sample libraries remain expensive to purchase: *Berlin Strings* retails at €840 [25], and *Solo Strings Bundle* at €790 [28].

A sample-based virtual instrument possesses many advantages:

- Static fidelity [26]; quality of tone is only limited to that of the original recordings.
- Low processing overhead, since audio is replayed but not computed.
- Effects such as pitch changes, mixing of desirable instrument noise and blending between samples may be used to produce greater sonic combinations.

However, such an instrument also has disadvantages:

- Extensive storage is required for the library of recordings used.
- Only the actual instruments recorded may be reproduced.
- Only articulations and effects pre-recorded, or generated by manipulation, may be produced.
- Hire of musicians and recording personnel is required for capture of the audio library.
- To maintain low latency playback, buffer memory large enough to hold all possible proceeding sample segments may be necessary.
- Phasing may result from sample blending needed to produce velocity or articulation transitions.

Additive synthesis

An interesting alternative to physical modelling and sample-based instruments is the reconstruction of sonic character using additive synthesis. This technique involves generating the short-term Fourier transform of a given audio segment as a mixture of its Fourier components, thus reconstructing it in the frequency domain [26]. Recordings of real sounds are analysed and their Fourier transforms extracted and stored. When a given segment is required by the simulation, it is reconstructed and blended with the preceding segment.

Wallander Instruments' WIVI Band combines this concept with instrument behavioural modelling in order to achieve emulation of brass instruments [30]. It requires 350Mb of disk space, and retails at \$129.00. *Sample Modelling's The Saxophones* [24] also uses this technique, and retails at €259.00, requiring 500Mb of disk space. As such they are more compact and less costly than their sample-based competitors.

Additive synthesis virtual instruments possess many advantages:

- Quality of tone is limited only to that recovered by Fourier analysis of the original recordings.
- Physical simulation of the instrument is not necessary.
- Effects such as pitch changes, mixing of desirable instrument noise and blending between transforms may be used to produce greater sonic combinations.

However, such an instrument also has disadvantages:

- The library of transforms must be stored in a similar manner to sample-based instruments.
- Only the actual instruments recorded may be reproduced.
- Only the articulations and effects originally recorded and transformed, or those generated by transform manipulation, may be produced.
- Hire of musicians and recording personnel is required for capture of the transform library.

3. Relevant theory

String simulation

Simulation of physical motion within a string under tension may be described by a specialisation of the wave equation [4]. Key elements of the simple model include the tension, mass and length of the string, which all affect the period of the motion produced, as specified by the classical physics equation shown in *eq. 3.1*.

$$\rho_L \frac{\delta^2 y(x,t)}{\delta t^2} = T_0 \frac{\delta^2 y(x,t)}{\delta x^2}$$

Eq. 3.1

Where ρ_L is its linear density, T_0 is the tension in Newtons, y is displacement orthogonal to the string, and t represents units of elapsed time. The string is described as shown in *figure 3.1*.

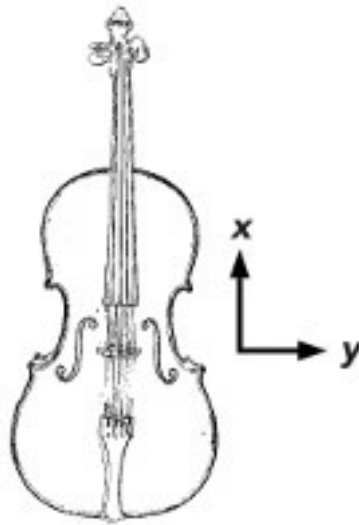


Figure 3.1: Axes describing string motion

Demoucron proposed a more complete model [6], still based on the classical equation, which introduces terms for stiffness and damping (*eq. 3.2*).

$$\rho_L \frac{\delta^2 y(x,t)}{\delta t^2} = T_0 \frac{\delta^2 y(x,t)}{\delta x^2} - EI \frac{\delta^4 y(x,t)}{\delta x^4} + R(w, y, t) + F_0$$

Eq. 3.2

Where $I = \pi d^4 / 64$ with string diameter d , E is the Young's modulus property of the string, and the term $R(w, y, t)$ describes physical damping of the string's motion due to heat and sound dissipation (*eq. 3.3*).

$$R(w, y, t) = -2b_1 \frac{\delta y}{\delta t} + 2b_3 \frac{\delta^3 y}{\delta t^3}$$

Eq. 3.3

Bilbao and Desvages proposed a similar model [2], but based on a “two-polarisation” scheme (motion in two planes, *figure 2.4*) which considered displacement in both horizontal and vertical coordinates orthogonal to the string. Given that these axes may be represented by y and z , they put forward the following equations (*eq. 3.4a and 3.4b*).

$$\rho_L \frac{\delta^2 y(x,t)}{\delta t^2} = T_0 \frac{\delta^2 y(x,t)}{\delta x^2} - EI \frac{\delta^4 y(x,t)}{\delta x^4} + -2b_1 \frac{\delta y}{\delta t} + 2b_3 \frac{\delta^3 y}{\delta t \delta x^2} + F_N - J_F f_F - J_B f_B$$

Eq. 3.4a

$$\rho_L \frac{\delta^2 z(x,t)}{\delta t^2} = T_0 \frac{\delta^2 z(x,t)}{\delta x^2} - EI \frac{\delta^4 z(x,t)}{\delta x^4} + -2b_1 \frac{\delta z}{\delta t} + 2b_3 \frac{\delta^3 z}{\delta t \delta x^2} + F_N + \phi_N - J_F f_F + \phi_F - J_B f_B + \phi_B$$

Eq. 3.4b

In any of these systems, the tension term serves to return the string to its origin (introducing forces which often cause overshoot of the string's origin, and so oscillation), and the stiffness and damping terms reduce the potential motion of the string. External forces are represented by the remaining terms.

The bowed string

A bowed string exhibits a y -axis displacement similar to a sawtooth wave due to the alternate slipping and sticking of the bow fibres to the surface of the string (*figure 3.2*). The increase in tension during the sticking state due to displacement causes an increase in applied force between bow and string, which eventually overcomes the friction between them, releasing the string which rapidly returns towards its origin (the slipping state) until the force reduces to a point where bow to string friction again produces a sticking state, and the cycle repeats. It is this “Helmholtz” motion [9], applied to the string at the bowing point, which produces the vibration characteristic of bowed instruments.

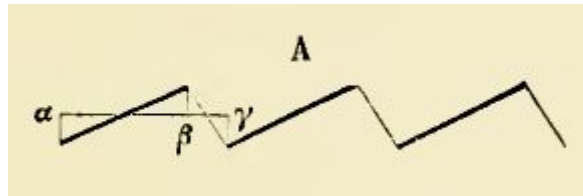


Figure 3.2: Helmholtz's “principal motion” of bowed string displacement. [9]

Extending a vibrating string model to include bowing involves allowing external energy generated by the bowing action to enter the model. Demoucron adds term F_0 to represent these forces, which he suggests may be considered as follows for the forces applicable to the bow (*eq. 3.5*).

$$F_0 = \mu(\Delta v) F_b$$

Eq. 3.5

Here, $\mu(\Delta v)$ represents the friction curve existing between bow and string, and F_b represents the force applied by the bow to the string.

Bilbao and Desvages' model (*eq. 3.4a and 3.4b*) defines terms representing bow, neck and the finger of the player. F_N is the contact force exerted by the neck of the instrument, f_F and f_B are finger and bow contact forces (with spatial distributions J_F and J_B respectively), and ϕ_N , ϕ_F and ϕ_B are friction curves applying to the neck, finger and bow.

Model solutions

To implement such a model, a numerical solution must be found to its differential equations. This may be done using the finite difference method to approximate the changes occurring in the equations' derivatives, so performing discrete integration of the continuous problem. Using this method, the displacement $y(x, t)$ of the string at point x and time t may be calculated discretely.

It is possible to produce this solution in the time domain, as with Bilbao and Desvages' implementation [2]. It is also possible to solve for the frequency domain, as chosen by Demoucron [6]. Doing so avoids discretization of the model's physical space, instead dividing it into harmonics and only computing the bounded frequency range necessary for reproduction of the desired partials. This is possible because the physical motion present on the string is a combination of the Fourier components producing the sound generated by it.

Demoucron's solution involves decomposing the displacement terms into orthonormal functions (eq. 3.6a), and integrating in space to obtain modal equations (eq. 3.6b).

$$\Phi_n(x) = \sqrt{\frac{2}{L}} \sin \frac{n\pi}{L} x$$

Eq. 3.6a

$$\rho_L \ddot{\alpha}_n(t) + T_0 \left(\frac{n\pi}{L} \right)^2 \alpha_n(t) = f_n(t)$$

Eq. 3.6b

Where again ρ_L is the string's linear density, T_0 is the tension in Newtons, L is the length of the string in metres, and α is the modal displacement. The result is a fast, mode-based algorithm for bowed string simulation, as implemented by Percival for *Artifastring* [20].

4. Specification and Design

System specification

From examination of the project brief and supervisor discussions, the following specifications were chosen:

- The simulation should be able to produce a given note of a cello in audio data format upon its execution, and support real-world effects such as glissando and versatile bowing.
- The engine's design should maintain a focus on audio timbre, and include sufficient fidelity and parameters to allow its output to be directly comparable to similar audio recordings of a real cello.
- The engine should model all four instrument strings, and include physical interaction between strings and instrument body for potentially greater perceived realism (sympathetic resonance of strings and body caused by the physical conduction of energy between them).
- Audio recordings of one or more real cellos should be obtained, and simulation parameters set and tuned by mathematical optimisation to match output to these target sounds as closely as possible. Target parameters should be string diameter, linear density, Young's modulus and damping coefficients.
- Objective comparison between audio outputs should be done in terms of their spectra, allowing a mathematical match to be approximate to a match in perceived timbre [9]. Steady-state excerpts should be used to maintain specific focus on the timbre of the notes produced (as opposed to bowing attack and decay, which are akin to noise in spectral terms).
- Subjective comparison between audio outputs should be done through human perception testing.
- Typical simulator outputs should be close to ideal (target sound(s)), with the possibility of expanding this scope to include all simulator output for future work.
- The simulation should be designed with the aim of real-time operation in mind, but revert to a non-real-time approach should this prove impractical with the given resources.

Given this description, the following structure was chosen as a design template:

- The sound generation engine, encapsulating four individual string simulations, should exist as a stand-alone library.
- Engine control will be by parameter assignment and function calls (e.g. *bow(position, pressure, velocity)*), with the addition of real-world control such as MIDI triggering retained as a possibility for later or future work.
- The optimiser should be developed as a separate structure which automatically operates the simulation engine to obtain output audio at each iteration.
- The system will implement PCM format files for ease of storage and analysis.

Functional design

The following design was constructed from these specifications:

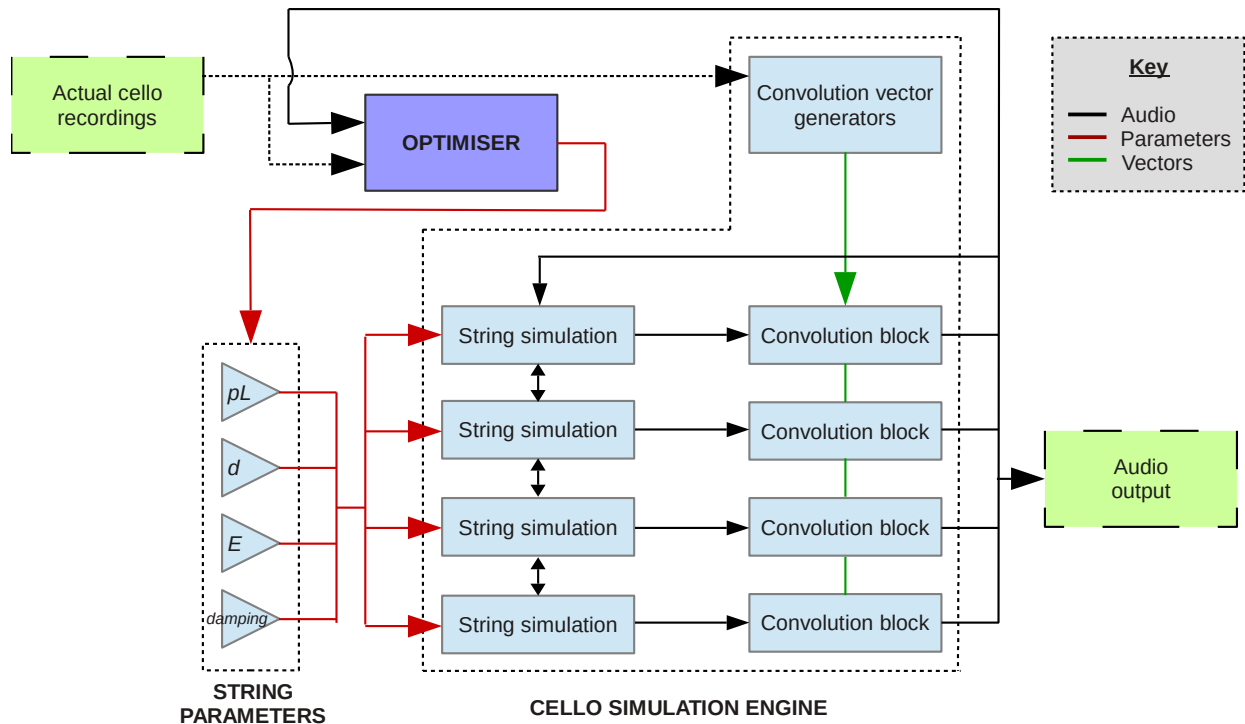


Figure 4.1: Basic functional design, composed of simulation engine and black-box optimiser

The simulation engine is composed of four individual string simulations, each with its own set of parameters and a convolution block. These parameters comprise string diameter (m), linear density (kg/m), Young's modulus and damping coefficients. Output from each convolution block is mixed together to form the engine's audio output, which is returned to the strings themselves via the simulation's bridge transform. Energy is also shared between strings, again via the bridge transform. These feedback mechanisms model the conduction of energy between the strings and body of a real cello [7].

Actual cello recording are used for calculation of convolution vectors suitable to transform simulation output into that which approximates them. They are also used as comparison inputs for the optimiser, which is able to control the four string parameters for each string simulation component, generating new output at each iteration.

5. Implementation

Preparation

Since it is known no entirely effective implementation of a physical model exists [26], the best choice of implementation may be a compromise. An entirely physical model, and a sample player which physically models only the instrument body were selected as possible development paths. Simple construction of both approaches was first attempted, and, following deliberations, it was decided that the resulting physical model's output showed construction of a purely physical simulation from scratch was possible. This route was selected for the project.

A set of recordings of a real cello were made, comprising individual bowed notes from each string. Audio was captured at 48kHz sampling frequency and 24-bit resolution using the open-source *Ardour 2* digital audio workstation software. An Audio Technica AT-4033 condenser microphone was placed approximately 0.75m from the cello body, level with the neck join, and connected via a TL-Audio 5051 pre-amplifier to an M-Audio Delta 1010 sound card. This audio, and professional recordings from the *Vienna Symphonic Library's Solo Cello* instrument [28], were used for comparison against simulator output.

Code created during implementation may be seen in full in *Appendix A*, or downloaded at the following URL:

https://www.dropbox.com/s/05126z8c3b7m7xj/c1322278_src.tar.bz2?dl=0

Cello simulation engine

String simulation

The C source code for *Artifastring* [20] was downloaded, and its string simulation functions implemented in Matlab, initially as a class, and later as an in-line code block to optimise engine performance (*pseudo-code 5.1*). This reduced the output to run-time ratio from 1:300 to 1:60. The *compute_bow()* functions were also re-factored and simplified, since they contained several repeated sequences. Four separate string simulations were run via *for* loop, allowing Matlab's optimiser to parallelise their execution.

```
% Original class-based code
self.a = ah + self.X3 .* fn;
self.ad = adh + self.Y3 .* fn;

...

% New in-line, vectorised code
s_str( currentString + a : currentString + a + aLen ) = ...
s_str( currentString + ah : currentString + ah + ahLen ) + ...
s_str( currentString + X3 : currentString + X3 + X3Len ) .* fn;
s_str( currentString + ad : currentString + ad + adLen ) = ...
s_str( currentString + adh : currentString + adh + adhLen ) + ...
s_str( currentString + Y3 : currentString + Y3 + Y3Len ) .* fn;
```

Pseudo-code 5.1: Use of indices and offsets replaced individual class properties.

Equation 3.6a is represented by four operations. During initialisation, *sqrt_two_div_L* (eq. 5.1) is computed and stored. A multiplier *div_pc_L* representing a division by *L* is also created (eq. 5.2), then a vector *inside_phi* with one element for each mode *n* produced using *div_pc_L* (eq. 5.3). The vectors *phix₀*, *phix₁* and *phix₂* are then generated during execution, each taking the sine function of a scaling *m* of *inside_phi* to produce the modal oscillations (eq. 5.4).

$$\text{sqrt_two_div_L} = \sqrt{\frac{2}{L}}$$

Eq. 5.1

$$\text{div_pc_L} = \frac{1}{L}$$

Eq. 5.2

$$\text{inside_phi} = n \pi \text{div_pc_L}$$

Eq. 5.3

$$\text{phix}_n = (\text{sqrt_two_div_L}) \sin(m(\text{inside_phi}))$$

Eq. 5.4

Equation 3.6b is modelled in a similar fashion. During initialisation, a vector *w0* with one element for each mode *n* is created (eq. 5.5), and used to produce vector sets *X* and *Y* representing step-wise modal displacements and velocities, including first and second derivatives (eq. 5.6a, 5.6b and 5.6c).

$$w0 = \sqrt{(T/\rho_L)(n\pi/L)^2 + (EI/\rho_L)(n\pi/L)^4}$$

Eq. 5.5

$$X1 = \left(\cos(w dt) + \left(\frac{rn}{w} \right) \sin(w dt) \right) \exp(-rn dt)$$

$$Y1 = -(w + rn^2) \sin(w dt) \exp(-rn dt)$$

Eq. 5.6a

$$X2 = \left(\left(\frac{1}{w} \right) \sin(w dt) \right) \exp(-rn dt)$$

$$Y2 = \left(\cos(w dt) - \left(\frac{rn}{w} \right) \sin(w dt) \right) * \exp(-rn dt)$$

Eq. 5.6b

$$X3 = \frac{(1 - X1)}{\rho_L w \theta^2}$$

$$Y3 = \frac{-Y1}{\rho_L w \theta^2}$$

Eq. 5.6c

α and $\ddot{\alpha}$ store the current displacements and velocities, and are initialised to zero at the start of the simulation. At each time step, the free oscillation modes α^h and $\ddot{\alpha}^h$ are computed (eq. 5.7a and 5.7b) and applied to create the final state of α and $\ddot{\alpha}$ (eq. 5.8a and 5.8b), with external forces exerted by multiplication with $phix_0$, $phix_1$ and $phix_2$ proportional to the forces applied. Final output is obtained through multiplication of α against the bridge transform G for each iteration.

$$\alpha^h = X(\alpha) + X'(\ddot{\alpha})$$

Eq. 5.7a

$$\ddot{\alpha}^h = Y(\alpha) + Y'(\ddot{\alpha})$$

Eq. 5.7b

$$\alpha = \alpha^h + X''$$

Eq. 5.8a

$$\ddot{\alpha} = \ddot{\alpha}^h + Y''$$

Eq. 5.8b

Several dedicated variables control the player's interactions with the string, as shown below:

Variable	Function
Fb	Bow normal force (N)
va	Bow acceleration (m/s/s)
vb	Bow velocity (m/s)
Kf_get	Normal force applied to string at neck by finger (N)
$finger_position$	Distance between finger and instrument nut (m)

Table 5.1: Engine variables representing external forces.

Cello body simulation

A cello body resonates differently given different frequencies at different amplitudes [7], for example when a different pitch is played, and when bowed gently or vigorously, producing a soft or strong excitation of the string (*figure 5.1*). A method was required which could simply simulate this behaviour, and time domain convolution was chosen for this purpose.

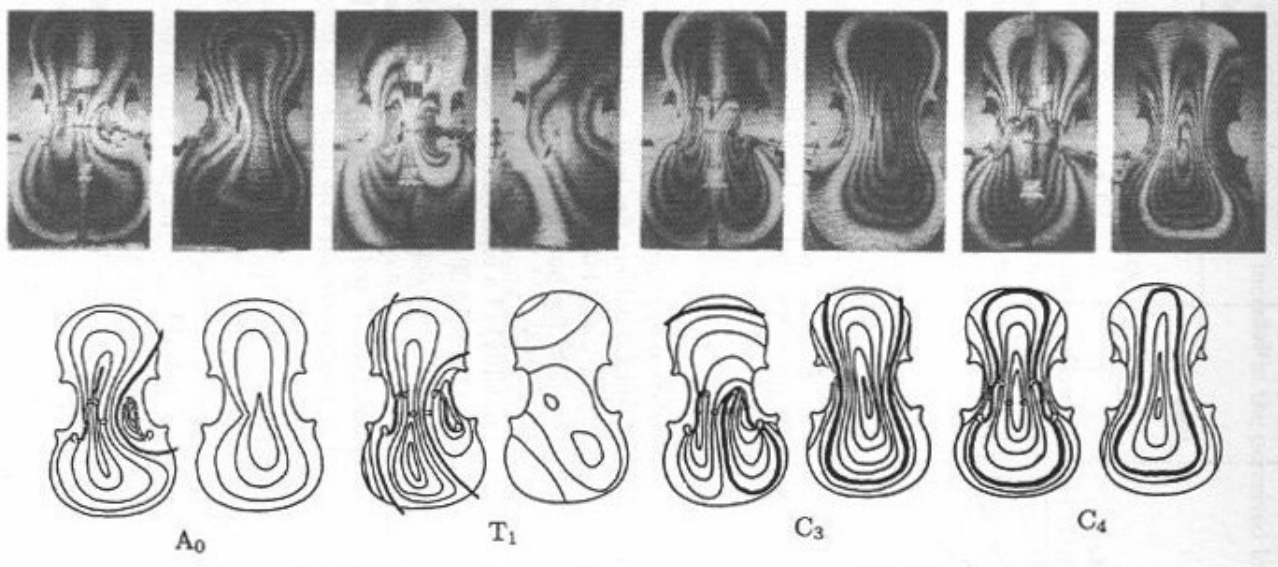


Figure 5.1: Shapes and frequencies of the A_0 , T_1 , C_3 and C_4 modes in a cello (body). [7]

A transform from one sound to another may be found in the frequency domain by dividing the FFT of the desired audio by the FFT of the source audio. Multiplying the source FFT by this transform vector, and performing an inverse FFT operation, results in exact reproduction of the desired audio (*figures 5.2a and 5.2b*).

Unfortunately, using the same vector on a different source sample produces a very different output, due to the altered amplitude and phase information contained within it. Also, the effect of the transform is spread across the duration of the input, so that if the new source audio changes in character in the time domain, relative to the original source audio, that change is lost following the transform operation. Attempting to transform a continuously bowed sound to a sound containing bow attack and decay results in attack and decay being added to the original sound. The opposite is also true, and these features are lost.

```
body_decay_time = 0.1;                % decay time in seconds

for n in num_strings:
    for note in notes:
        edited_input_sim = fade_in_and_out( input_sim( n,note) );
        edited_input_real = fade_in_and_out( input_real(n,note) );

        frequency_transform = fft( edited_input_real ) / fft( edited_input_sim );

        convolution_vector(n,note) = ...
            inverse_fft( fade_out( frequency_transform, body_decay_time ) );
```

Pseudo-code 5.2: Calculation of approximate impulse response vectors

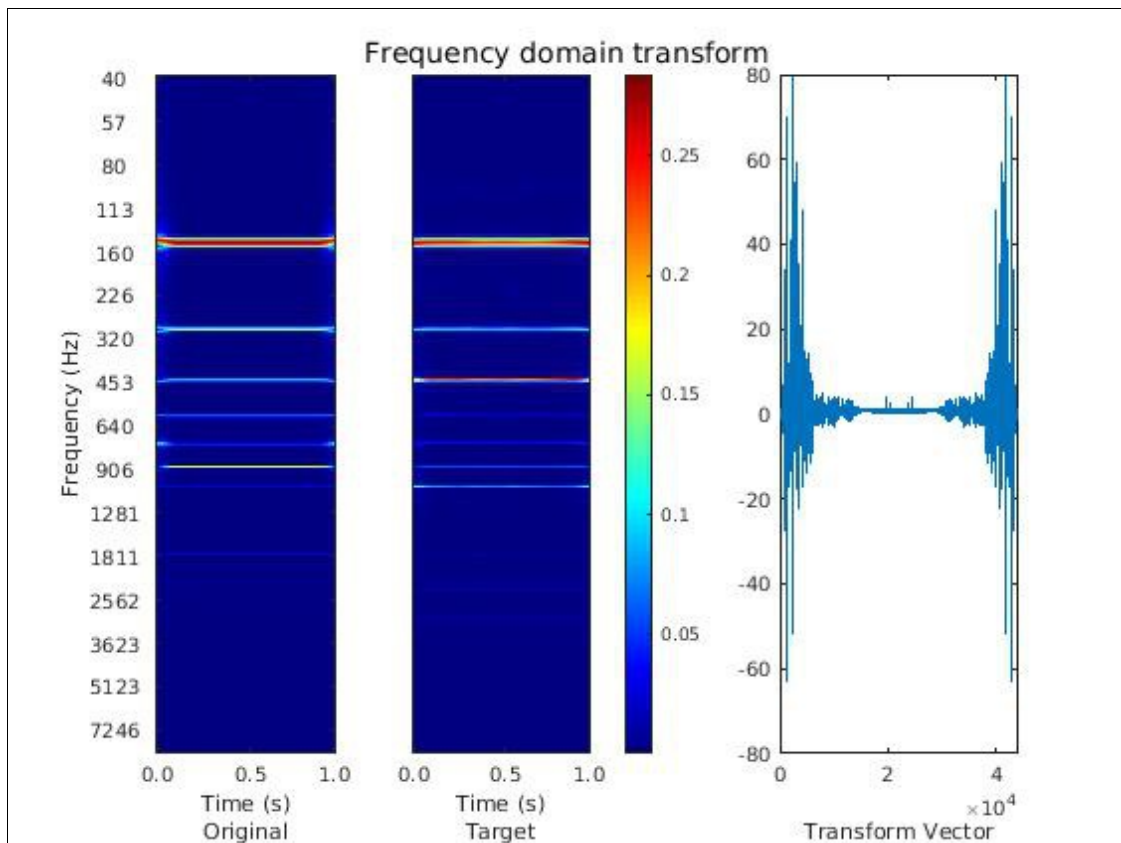


Figure 5.2a: Spectra of original and target audio, and the resulting frequency domain transform vector.

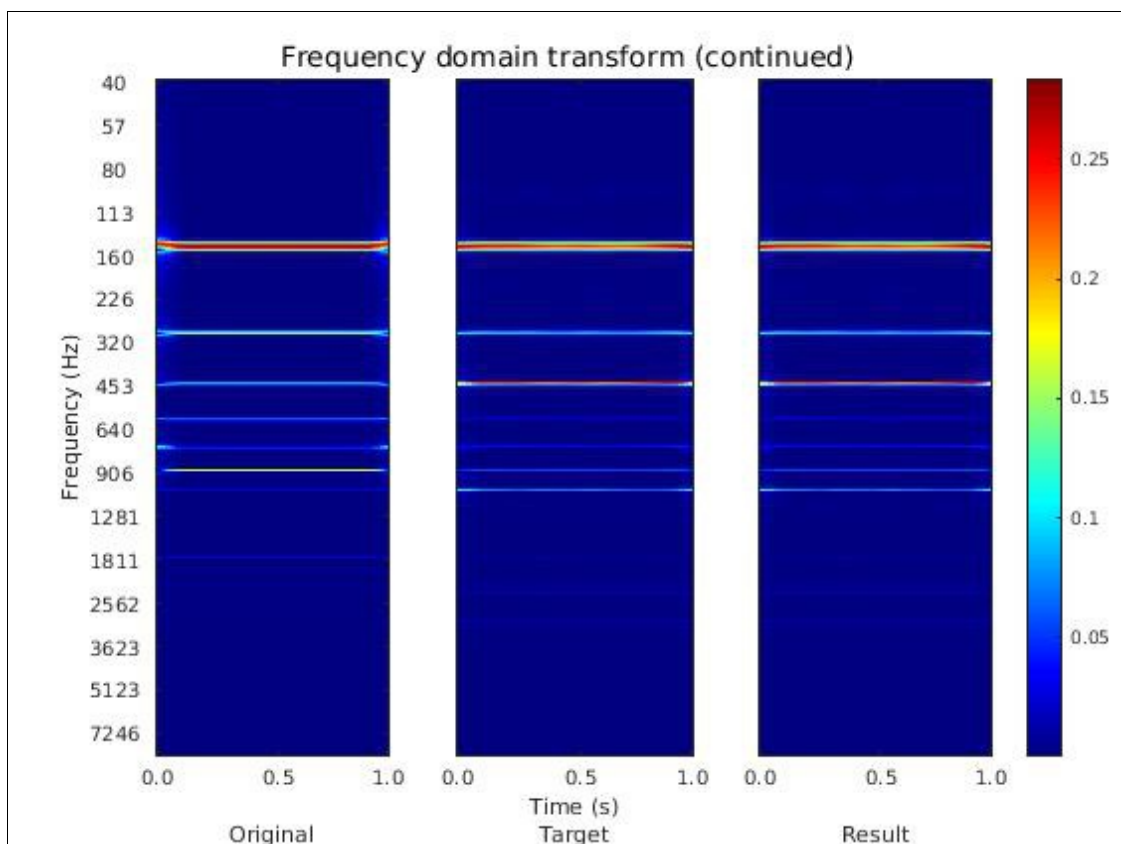


Figure 5.2b: Spectra of original audio, target audio and original audio multiplied by transform vector.

However, transformation in the time domain using a convolution vector does not exhibit this problem. A set of raw string outputs from the simulator at different pitches and amplitudes, and the corresponding real cello samples, were assembled. When the simulation is initialised, a set of convolution vectors is calculated from the difference between these sounds in the frequency domain (eq. 5.9, pseudo-code 5.2).

$$v(s, n) = i \left(\frac{f(a_t t)}{f(a_s t)} \right) d$$

Eq. 5.9

Where s = string number, n = note identifier, a_t = target audio, a_s = source audio, t = short fade in and fade out transform, and d = decay time transform. $f()$ is the FFT function, and $i()$ is the inverse FFT function.

This results in a three-dimensional vector store with dimensions *string number*, *frequency* and *amplitude*.

The relevant pitch-based vector for a string at any given time is calculated as a distance-proportional mix between the two stored vectors closest to the string's current pitch (eq. 5.10), which is computed during simulation (pseudo-code 5.3).

$$f = f_s \left(\frac{1}{d/L} \right)$$

Eq. 5.10

Where f_s = fundamental frequency of the string (Hz), d = distance of finger from nut (m), and L = string length (m).

```
frequency = fundamental_frequency( string_num );

if( finger_position > 0 ):           % if this needs to be taken into account
    frequency = frequency * ( 1 / ( finger_position / string_length ) );
```

Pseudo-code 5.3: Calculation of requested playing frequency for a given string

This results in a set of amplitude-based vectors, and the final active convolution vector for the string is calculated as a distance-proportional mix between the closest of these vectors to the string's current output amplitude (eq. 5.11), also computed during simulation (pseudo-code 5.4).

$$v = \sqrt{o^2} \left(\frac{1}{m} \right)$$

Eq. 5.11

Where o = string output (dB), and m = maximum string output (dB).

```
ratio = 1 / peak_string_amplitude;

amplitude_RMS = sqrt( mean( string_output^2 ) ) * ratio;
```

Pseudo-code 5.4: Calculation of the relative output amplitude of a given string

The method used to arrive at the pitch-based vectors (eq. 5.12a, 5.12b and pseudo-code 5.5) and final vector (eq. 5.13, pseudo-code 5.6) allows smooth dynamic transitions between vectors based on changes in pitch and amplitude, and so a step-wise solution to the vector matrix with responses for individual strings, similar to that of a real cello body.

$$p_u = \frac{1}{(f_u - f_l) / (f_c - f_l)}$$

Eq. 5.12a

$$p_l = 1 - p_u$$

Eq. 5.12b

Where p_u = proportion of upper vector, p_l = proportion of lower vector, f_c = current fundamental frequency (Hz), f_l = lower vector's fundamental frequency (Hz), and f_u = upper vector's fundamental frequency (Hz).

$$v = \left(v_l \left(1 - \frac{1}{(a_u - a_l) / (a_c - a_l)} \right) \right) + \left(v_u \frac{1}{(a_u - a_l) / (a_c - a_l)} \right)$$

Eq. 5.13

Where v_l = vector for lower amplitude, v_u = vector for upper amplitude, a_c = current amplitude (dB), a_l = lower vector's amplitude (dB), and a_u = upper vector's amplitude (dB).

An example of this transition may be seen in figure 5.3. An advantage of this scheme is that any ambience captured by the source recordings, such as tonal colouration from the room, is also included in the convolution, allowing simulation of not only the instrument body but also its environment, if desired.

```
for n in num_strings:
    [ lower_vector_index, upper_vector_index ] = ...
        locate_nearest_vector_indices_by_frequency( current_freq(n) );

    range      = pitch_upper      - pitch_lower;
    distance    = current_freq(n) - pitch_lower;

    upper_proportion = 1 / ( range / distance );
    lower_proportion = 1 - upper_proportion;
```

Pseudo-code 5.5: Calculation of a mixed convolution vector proportional to frequency distance

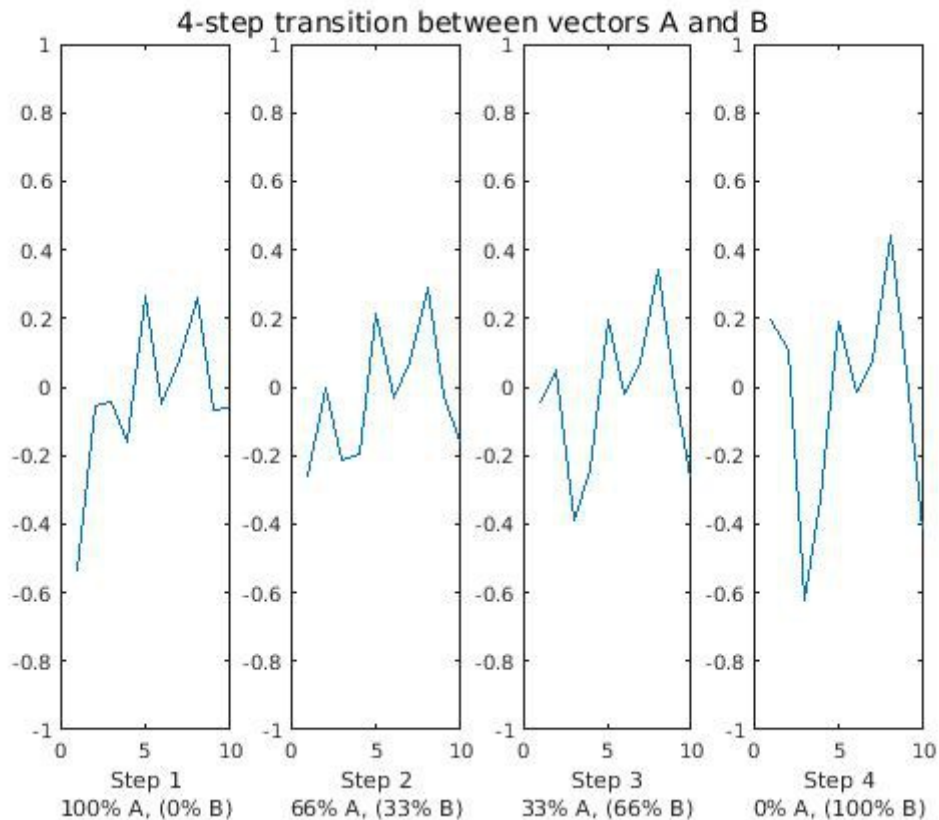


Figure 5.3: Smooth transition between convolution vectors (section), achieved using proportional mixing.

To allow step-wise convolution, a *Convolve* class was developed which accepted a vector and single sample as an input, and output a single sample value representing the result of the convolution process at each time step. The output of an instance of this class was compared numerically to the output of Matlab's *conv()* function, and its correct operation confirmed.

```
for n in num_strings:
    % lower and upper proportions and vector indices calculated based on
    % frequency, as in pseudo-code 5.5
    [ convolution_vector_for_lower_amplitude, ...
      convolution_vector_for_upper_amplitude ] = ...
        locate_nearest_vectors_by_amplitude( amplitude_RMS, ...
                                              lower_proportion, lower_vector_index, ...
                                              upper_proportion, upper_vector_index );

    range      = amp_upper      - amp_lower;
    distance   = current_amplitude(n) - amp_lower;

    upper_proportion = 1 / ( range / distance );
    lower_proportion = 1 - upper_proportion;

    convolution_vector = ...
        ( lower_proportion * convolution_vector_for_lower_amplitude ) + ...
        ( upper_proportion * convolution_vector_for_upper_amplitude );
```

Pseudo-code 5.6: Calculation of convolution vector proportional to frequency and amplitude distance

String damping

A vector of damping coefficients was added to each string, and applied to their modal velocity vectors by multiplication at each time step (eq. 5.14, pseudo-code 5.7). This facilitates selective suppression of each mode individually for each string, allowing fine control of the sonic character of its output by alteration of its coefficients.

$$\begin{aligned}a_d &= a_d d \\ o &= aG \\ \text{Eq. 5.14}\end{aligned}$$

Where a = modal displacements, a_d = modal velocities, d = damping coefficients, and G = bridge transform.

```
for n in num_strings:
    ...
    % apply to first derivative, to affect next cycle...
    ad(n)      = ad(n) .* damping_coefficients(n);
    % ...and then generate output as normal
    output(n)  = a(n) .* G;
```

Pseudo-code 5.7: Damping coefficients applied before each string's y-displacement (output) is calculated

Sympathetic resonance

In order to model the interaction between vibrating and otherwise non-vibrating strings, code was added to allow energy present in each string to be introduced to every other string (eq. 5.15), where that energy occurred at frequencies each individual string exhibited a response to (pseudo-code 5.8).

$$\begin{aligned}v_n &= v_n + p(v_o G) \\ \text{Eq. 5.15}\end{aligned}$$

Where v_n = velocities of string n , v_o = velocities of other strings, p = feedback proportion, and G = bridge transform.

```
% initialisation
for n in num_strings:
    matching_frequencies( n ) = get_matching( string_modes( n ) );
shared_strings = { 1: (2,3,4), 2: (1,3,4), 3: (1,2,4), 4: (1,2,3) };
...
% main loop
for n in num_strings:
    velocities_of_string_n += ...
    velocity( strings_out( shared_strings(n), matching_frequencies(n) ) *G );
```

Pseudo-code 5.8: Energy fed back between strings

Audio frequency energy from the body of a real cello is also able to excite the strings through vibration via the bridge [7]. To model this interaction, terms were added to the energy feedback code (eq. 5.16, pseudo-code 5.9). The string simulation contains a vector G , which represents the frequency domain transform of a cello bridge. This was used to process audio energy passing between the strings and the body.

$$v_n = v_n + p((v_o + v_b)G)$$

Eq. 5.16

Where v_b = velocities of body modes.

```
% main loop
for n in num_strings:
    velocities_of_string_n += ...
    velocity( ( strings_out( shared_strings(n), matching_frequencies(n) ) ...
               + body_out( matching_frequencies(n) ) * G );
```

Pseudo-code 5.9: Body energy fed back to each string

Natural variations

Artifastring applies a random value A_noise to its bow force computations at each time step, in an effort to reduce the mechanical nature of its slip-stick cycles. In spite of this, a certain repetition remains audible in its output. Increasing A_noise does break up the periodicity of the output, but introduces the undesirable artefacts characteristic of step-wise digital noise.

```
% Oscillator setup
delta = ( 2 * PI ) / ( sample_rate / oscillator_base_rate );
step = delta;
state = 0;
```

Pseudo-code 5.10: Initialisation of a randomly varying oscillator

Randomly varying sine oscillators, limited in rate and extent of change, were added to the engine and applied to the bow force, (replacing A_noise , eq. 5.17, pseudo-code 5.10 and 5.11), bow velocity and finger position parameters. A separate oscillator was implemented for each of these parameters, and their outputs applied to all strings. Example oscillator output may be seen in figure 5.4.

$$s_n = s_{n-1} + (a_n(v - 0.5)d)$$

$$r_n = r_{n-1} + s_n$$

$$f_b = f_r + (\sin(r_n)a_o)$$

Eq. 5.17

Where v = random value, s_n = oscillator step n , a_n = oscillator noise amount, d = oscillator delta, r_n = oscillator state (radians), f_b = bow force (N), f_r = requested bow force (N), and a_o = oscillator effect amount.

A fourth oscillator was added to simulate the pitch changes of basic vibrato, caused by modulation of finger position on a real instrument by the performer. Its output was also added to the finger position parameter.

```
% Oscillator calculation at each iteration
step = step + ( ( amount * ( random - 0.5 ) ) * delta );
if( step </> delta -/+ ( max_deviation * delta ) ):
    limit_excursion( step );

state = state + step;
oscillator_driven_noise = sin( state ) * oscillator_amount;

% Oscillator output applied to bow force at each iteration
bow_force = requested_bow_force + oscillator_driven_noise;
```

Pseudo-code 5.11: Step-wise calculation and one application of a randomly varying oscillator

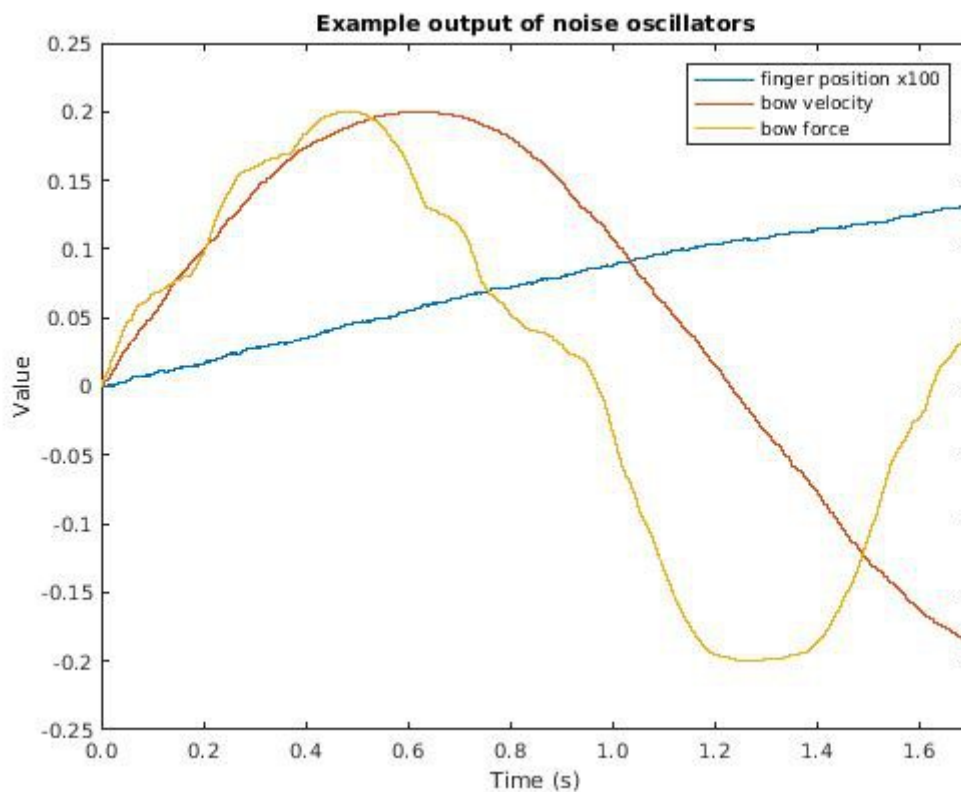


Figure 5.4: Output of oscillators for finger position, bow force and velocity, showing frequency modulation.

Spectral comparison

Comparison function

A function was developed to allow spectral comparison between two audio signals, where the result is the average difference between the absolute values of the spectra's real components in dBW. A result of zero denotes a complete spectral match (0dBW difference).

The constant Q transform [3] was selected as suitable for this process, since it maintains a constant ratio of bandwidth to centre frequency. This logarithmic response matches the logarithmic progression of the musical scale, allowing a linear harmonic spacing between the bins of its output. An equation similar to that of the discrete short-term Fourier transform is used to calculate the k th bin of the time domain vector being analysed (eq. 5.18).

$$x(k) = \frac{1}{N(k)} \sum_{n=0}^{N(k)-1} W(k, n) x(n) \exp\left(\frac{-j2\pi Qn}{N(k)}\right)$$

Eq. 5.18

Where the period in samples is $N(k)/Q$, $W(k, n)$ is the window function whose length is a function of k , and the k th component's frequency may be found by $2\pi Q/N(k)$.

The head and tail of the samples to be analysed are faded to prevent sudden transitions from affecting the result. An overview of the comparison code, which uses the *Constant-Q Transform Toolbox* developed by Klapuri and Schörkhuber [11], may be seen below (pseudo-code 5.12).

```
function compare( a, b, sample_rate ):  
  
    fade_time = 0.01; % fade time, in seconds  
    a = fade_in_and_out( a, fade_time * sample_rate );  
    b = fade_in_and_out( b, fade_time * sample_rate );  
  
    result = maximum_value;  
  
    if( not errors( a, b ) ):  
        ya = abs( real( constant_q_transform( a, sample_rate ) ) );  
        yb = abs( real( constant_q_transform( b, sample_rate ) ) );  
  
        result = mean( log( abs( ya - yb ) ) );  
  
    return result;
```

Pseudo-code 5.12: How the constant-Q transform is used to compare two audio samples

Figure 5.5 shows the transforms and calculated difference in dBW between two audio samples of an open, bowed second string, one from the simulator and one from the *Solo Strings Bundle* [28].

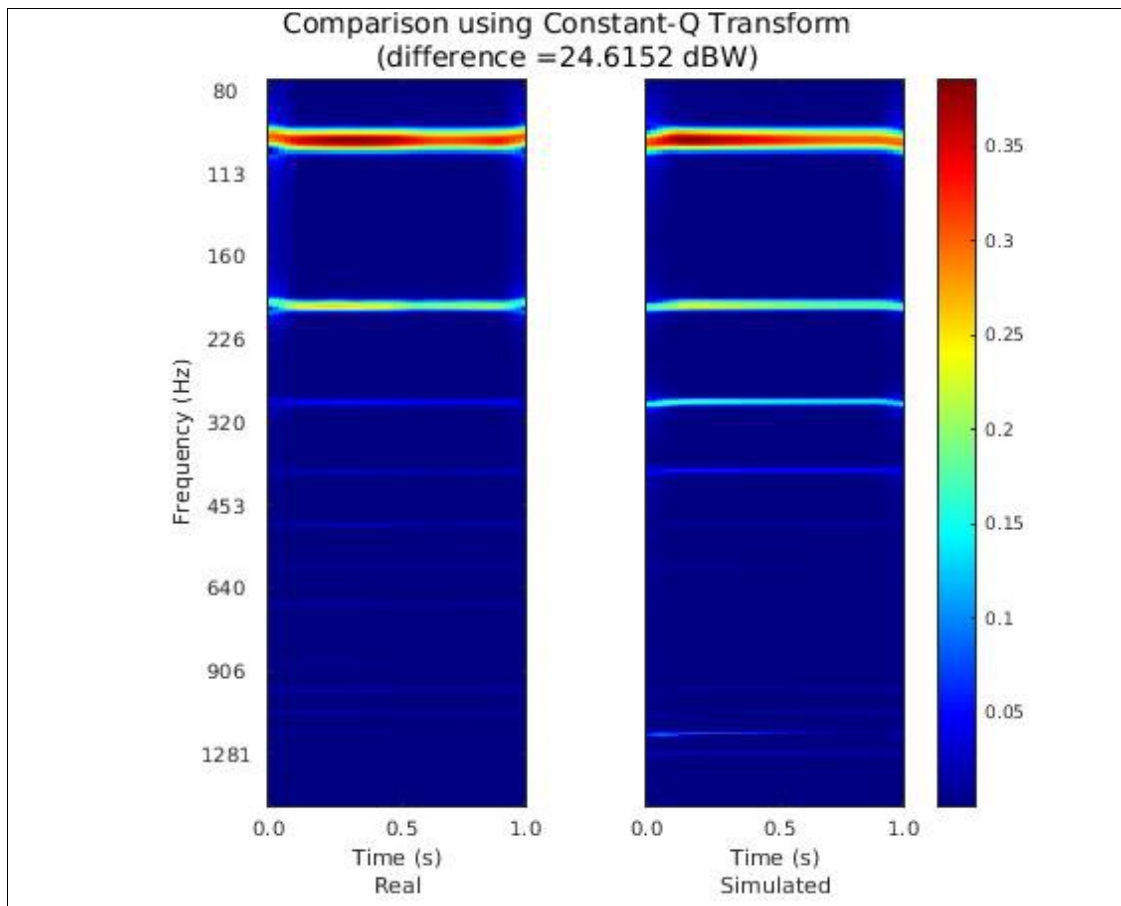


Figure 5.5: Constant-Q transforms of simulated and real cello samples, with calculated difference.

Validity checks

Stringed instruments are difficult to play due to the particular conditions under which correct oscillation of the string occurs [21]. Because of this, and because it is possible to set the simulation's parameters so that its output does not oscillate correctly, the function is occasionally expected to perform a comparison where one of the sounds given is not correctly formed. This causes undefined results which can mislead the optimiser.

To help address this issue, each sample's characteristics are compared to a set of known erroneous states (*pseudo-code 5.13*), and the maximum allowable value (signifying the greatest possible difference) returned if either is discovered to be in error.

```
threshold_level = 0.1;      % samples below this level are considered errors
threshold_length = 0.5;    % maximum allowable length of error in sample

% error flags: if == 1, error has been detected
error_condition_1 = sum( abs( sample ) ) / length > threshold_level;

error_condition_2 = mode( sample ) == 0;

error_condition_3 = abs( sum( sample > 0 ) - sum( sample < 0 ) ) / length ...
                    > threshold_length;
```

Pseudo-code 5.13: Testing for error conditions within the input audio samples

Example waveforms resulting from these three error conditions may be seen in *figure 5.6*:

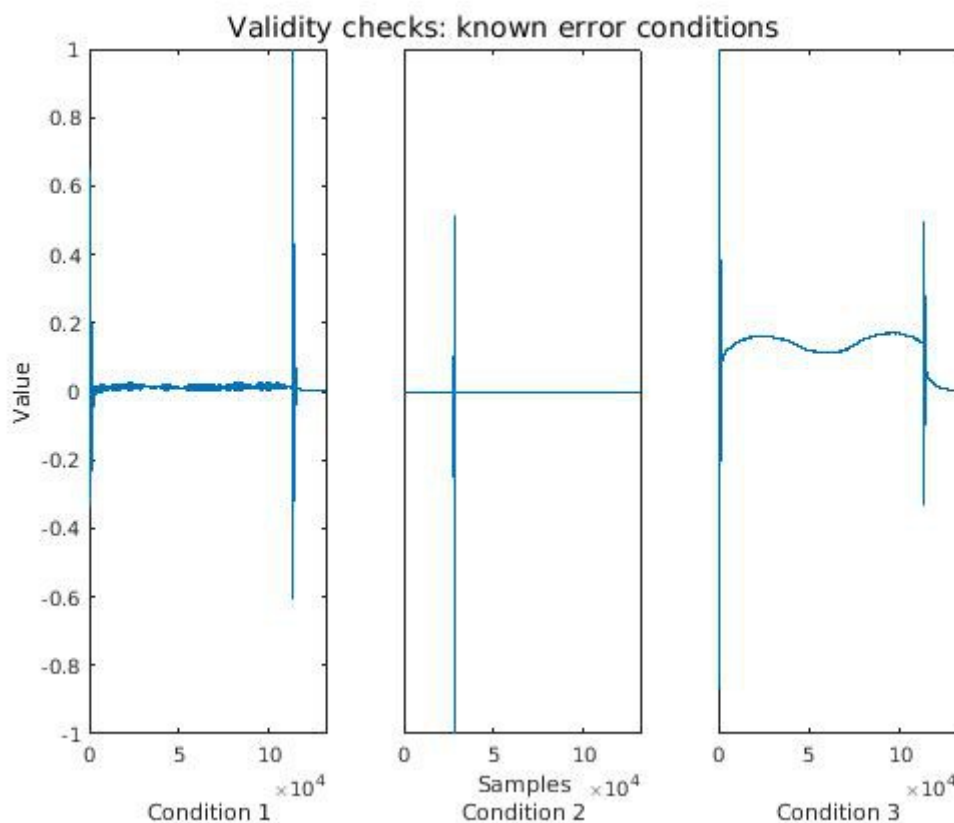


Figure 5.6: Examples of waveforms matching known error conditions, resulting from invalid parameters.

The first error can be detected by examining the proportion of samples which are absolutely lower than a given threshold, and is characterised by a low-amplitude sample. The second can be detected by testing the mode of the sample – when this is zero, a large proportion of the sample is silence, something which should not occur during the steady bowing state. The third error results in a significant offset within the sample, and can be detected by testing whether the proportion of negative to positive values is high. Values for these thresholds were decided upon by experimentation, and may not be optimal.

Optimiser

Construction

An optimiser class was constructed which tunes particular parameters of the cello simulation engine and attempts to minimise the spectral difference between its output and a given sample of a real cello, using the constant-Q comparison function described previously. Steady-state bowing was found to occur in every normal case by 0.45s of simulation output. The simulator was set to continuously bow, and this period removed from its output, and that of the target (real) sample, before both were passed to the comparison function.

The string's properties d , pl and E (representing string diameter (m), linear density (kg/m) and Young's modulus) and its damping vector were selected as suitable parameters for optimisation. An equation giving the required tension T for a combination of d , pl and E values was calculated (eq. 5.19), so that the fundamental pitch produced by the simulation would remain constant and its output could be directly compared to the target sound as described above.

$$T = \frac{\left(\frac{-EI}{pl} \right) \left(\frac{\Pi}{1/L} \right)^4 + (2\Pi f)^2}{pl \left(\frac{\Pi}{1/L} \right)^2}$$

Eq. 5.19: Calculating tension for given d , pl and E values.

Where L = string length (m), and f = desired fundamental (Hz).

In order to reduce the search space of the damping vector, a class was created which allows the generation of a basic n -point curve, controllable via a defined number of bands $< n$ (eq. 5.20a, 5.20b and pseudo-code 5.14).

```
function construct( num_points, num_taps, tap_width ):  
    % set up num_points vector controlled by num_taps values  
    tap_spacing = num_points / ( num_taps - 1 );  
    tap_curve    = hanningWindow( width = tap_spacing );  
  
    current_centre = 0;  
    for tap in num_taps:  
        tap_centres( tap ) = current_centre;  
        current_centre = current_centre + tap_spacing;  
  
function getCurve( tap_values ):  
    % returns a num_points vector from num_taps values  
    for tap in num_taps:  
        window = tap_curve * tap_values( tap );  
        start  = tap_centres( tap ) - floor( tap_width / 2 );  
        stop   = tap_centres( tap ) + ceil( tap_width / 2 );  
  
        curve_vector( start to stop ) = ...  
            max( curve_vector( start to stop ), window );
```

Pseudo-code 5.14: Simple generation of an n -point vector from $< n$ values.

Additionally, to prevent suppression of the string's principal frequencies of vibration, damping

coefficients 1 to 5 and 11 to 15 are fixed at 1. This allows variation of the damping characteristic without production of erroneous output for the open string and +7 semitones chosen for optimisation.

$$c_n = c_{n-1} + \frac{n_p}{(n_t - 1)}$$

Eq. 5.20a

$$i_a = c_n - \left\lfloor \frac{h_w}{2} \right\rfloor$$

$$i_b = c_n + \left\lceil \frac{h_w}{2} \right\rceil$$

$$v_{i_a \leq n \leq i_b} = \max(v_{i_a \leq n \leq i_b}, \text{hann}(h_w) v_n)$$

Eq. 5.20b

Where n_p = number of points in target vector, n_t = number of points in controlling vector, h_w = window width, c = window centres, v = controlling vector, and i = index into target vector.

Examples of string damping curves resulting from the optimiser's use of the n -point curve generator may be seen in figure 5.7 for $n = 8$.

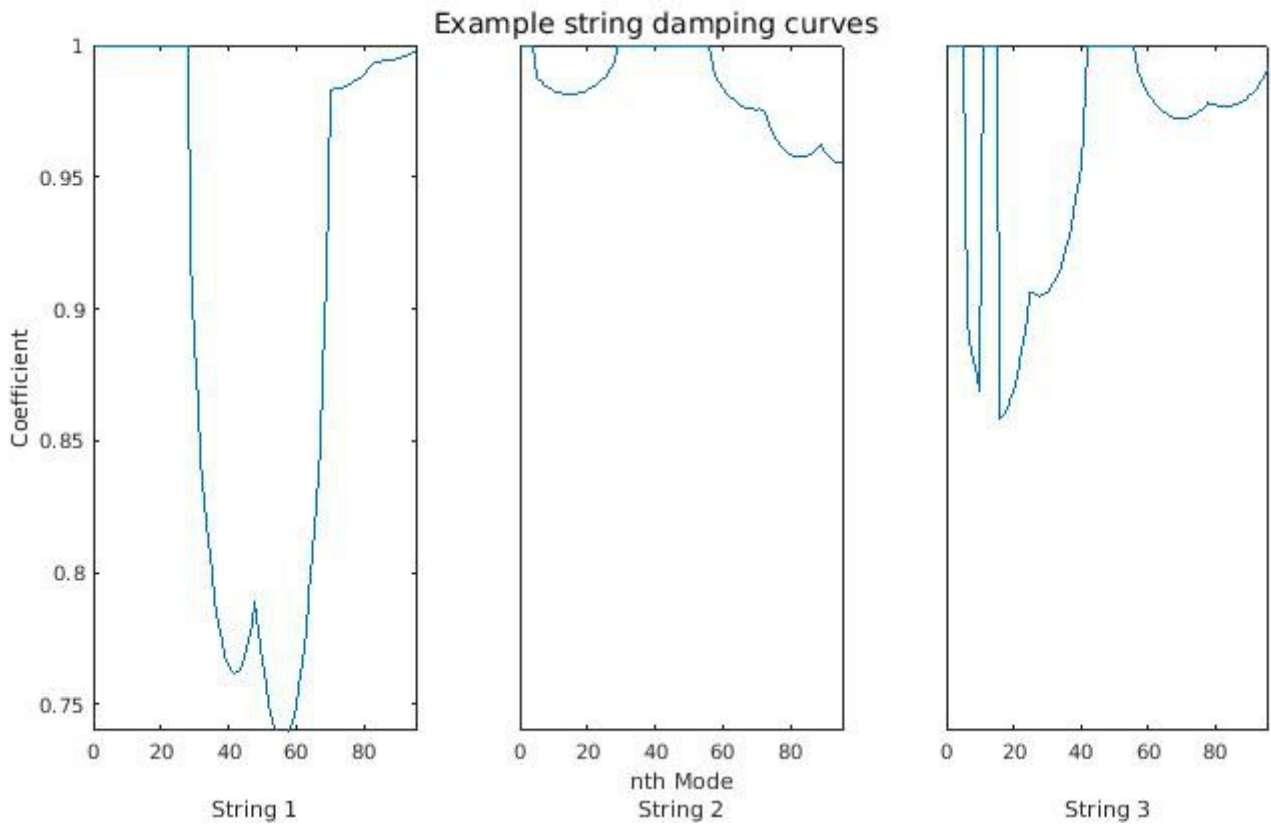


Figure 5.7: Damping curves produced by the optimiser for open strings 1, 2 and 3.

Optimisation methods

Two separate optimisation routines were constructed. The first performs a multi-level coordinate search [19] to seek suitable values for d , pl and E , since this algorithm is able to traverse the search space more aggressively. The second uses a variant of the Nelder-Mead optimisation algorithm with bounds [5] to seek a suitable set of damping coefficients, since the search space can be limited to values between 0 and 1, and an initial state can be set to represent a reasonable guess at the desired damping characteristic.

Since discovering a set of optimum parameters takes a great many iterations, the optimiser was not designed to function as part of a real-time system. Instead, the values returned by it are applied to the simulator prior to the generation of the final optimised output.

To allow a faster repeated traversal of the search space, all simulation output generated during the optimisation process is indexed by its parameter values and stored for later retrieval. Optimal parameter sets generated, and an example of the traversal of the search space, may be seen in *Appendix C*.

6. Results and Evaluation

Test overview

The system's output was examined objectively and subjectively, using mathematical comparison and human perception testing. Real cello recordings from the *Solo Strings Bundle* [28], and those made during preparation (hereafter known as “*Recordings I*” and “*Recordings II*” respectively), were used as examples of real instruments for these tests.

Figure 6.1 shows the spectra of 440Hz (A440) and 220Hz sine waves, recording of a real cello playing an open fourth string (with fundamental of 220Hz), and simulator output of the same note, as an example of spectral composition. Notice how the cello notes' second harmonic (440Hz) is weaker than the fundamental, the third harmonic of the simulated output is overly strong, and that the gradual level reduction of the real sound with increasing frequency is not ideally matched by the simulator. The subtleties of the spectrum produced are responsible for the perception of quality of tone, as discovered by Helmholtz [9].

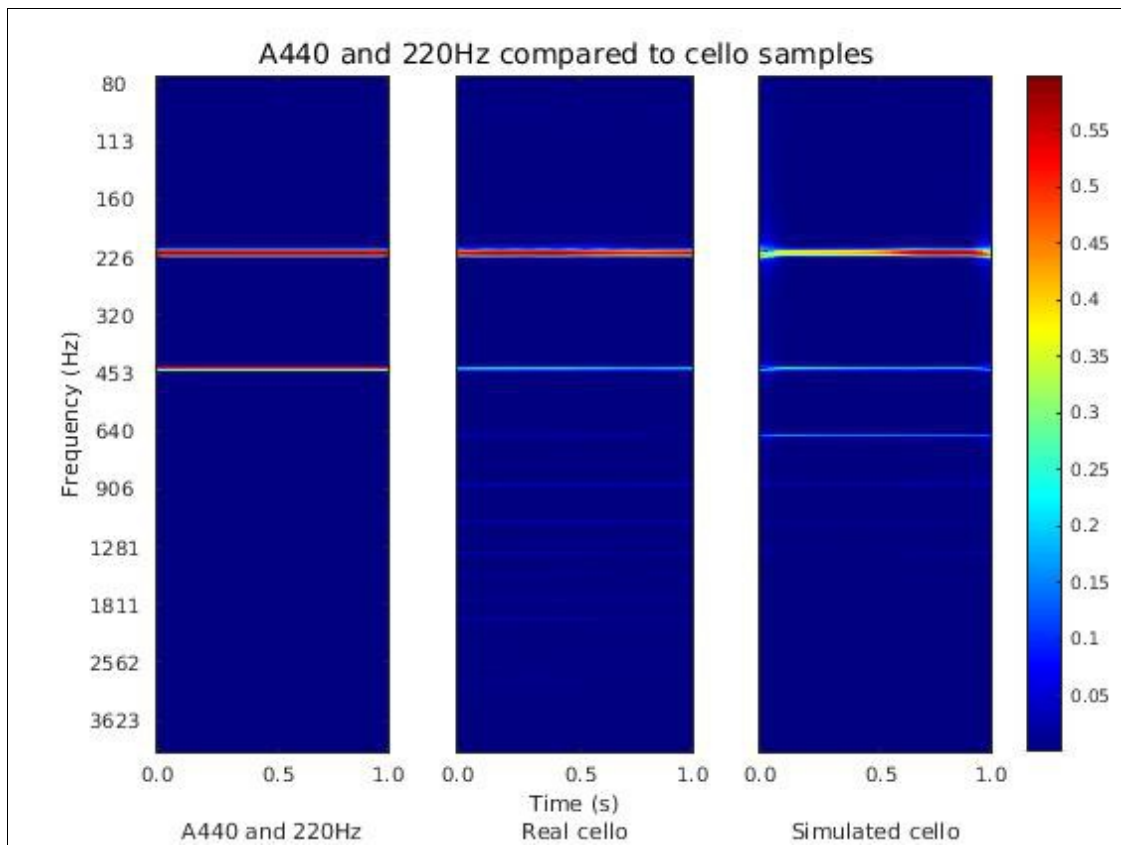


Figure 6.1: Example spectra from a 440Hz sine wave, real cello and simulated output.

Figure 6.2 shows the result of glissando between the open first string and one octave higher on the same string (followed by vibrato). Like legato, this effect is supported particularly by the physical simulation method.

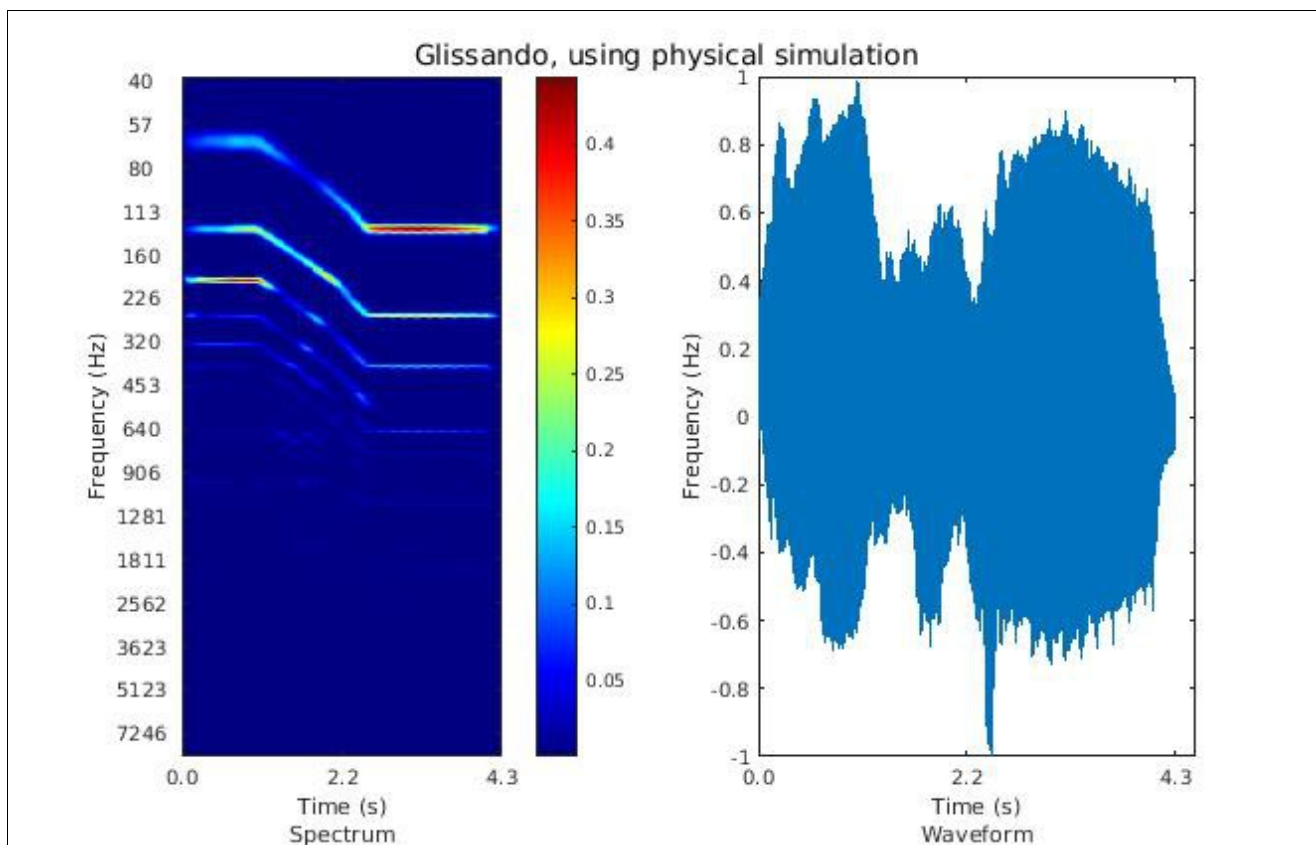


Figure 6.2: Waveform and spectrum resulting from simulating 1 octave glissando.

Objective testing

Overall difference between the constant-Q transforms of real and simulated sounds, as described in the *Spectral comparison* section, were used to compare pre- and post-optimisation audio to equivalent notes from real instruments. All frequencies between 10Hz and the Nyquist frequency (here, 22.05kHz) were analysed using a resolution of 1024 bins per octave, and both steady state (without attack and decay) and full samples (with attack and decay) were compared.

Recordings I

Comparison between the steady-state portions of both *Artifastring* and this work against *Recordings I* yielded the results shown in table 6.1. Figures 6.3 through 6.6 show the different spectra of these samples.

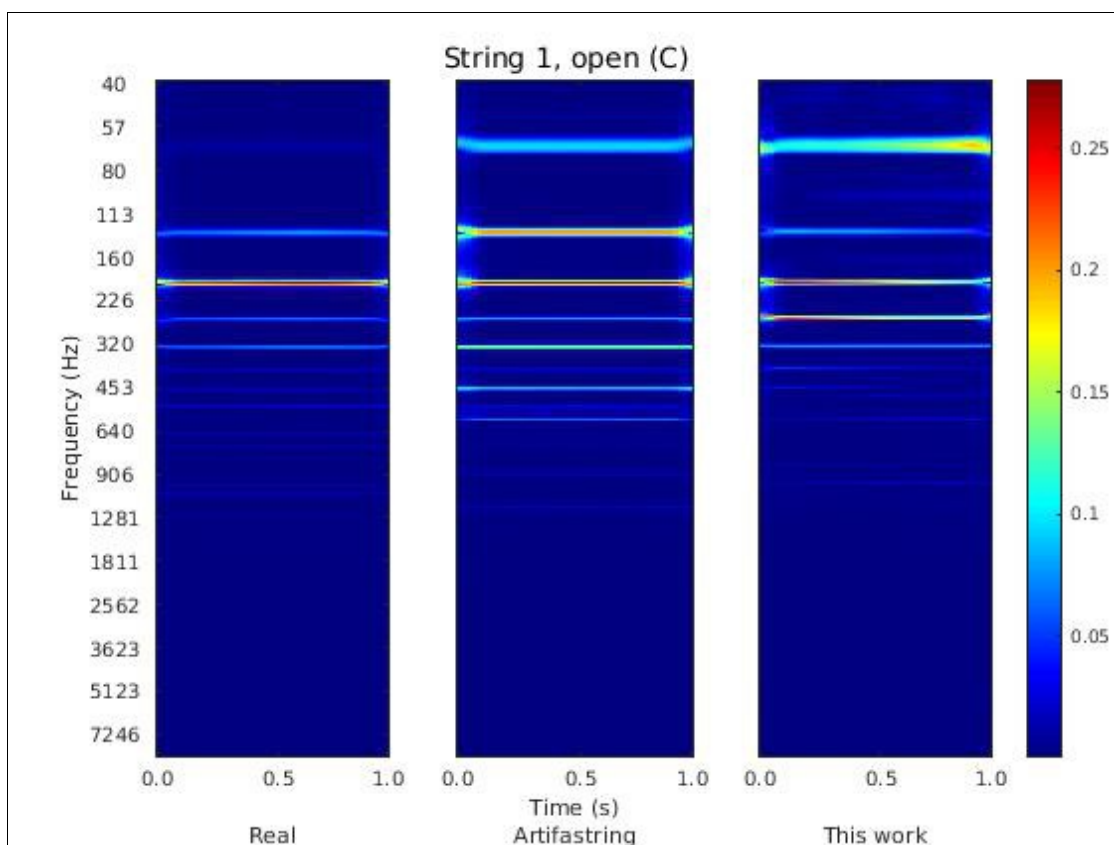


Figure 6.3: Spectra resulting from the open bowing of the first string (real, *Artifastring* and this work).

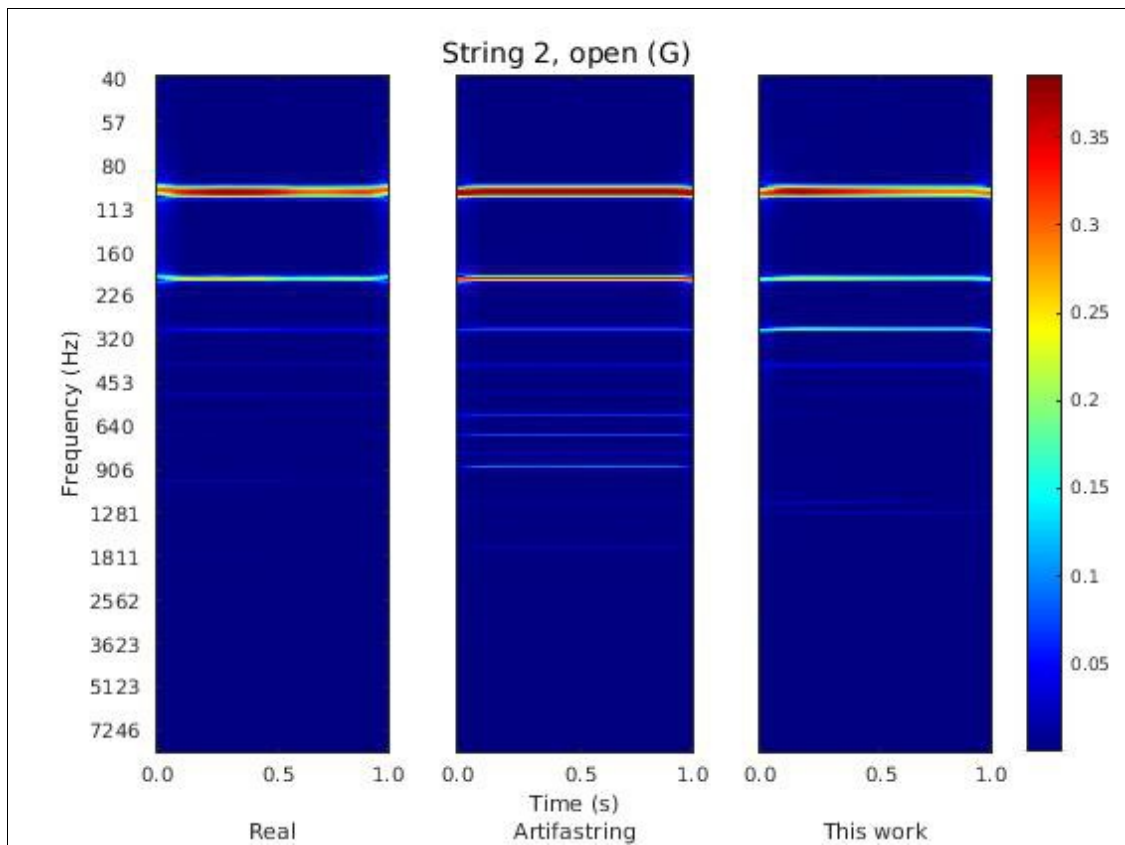


Figure 6.4: Spectra resulting from the open bowing of the second string (real, Artifastring and this work).

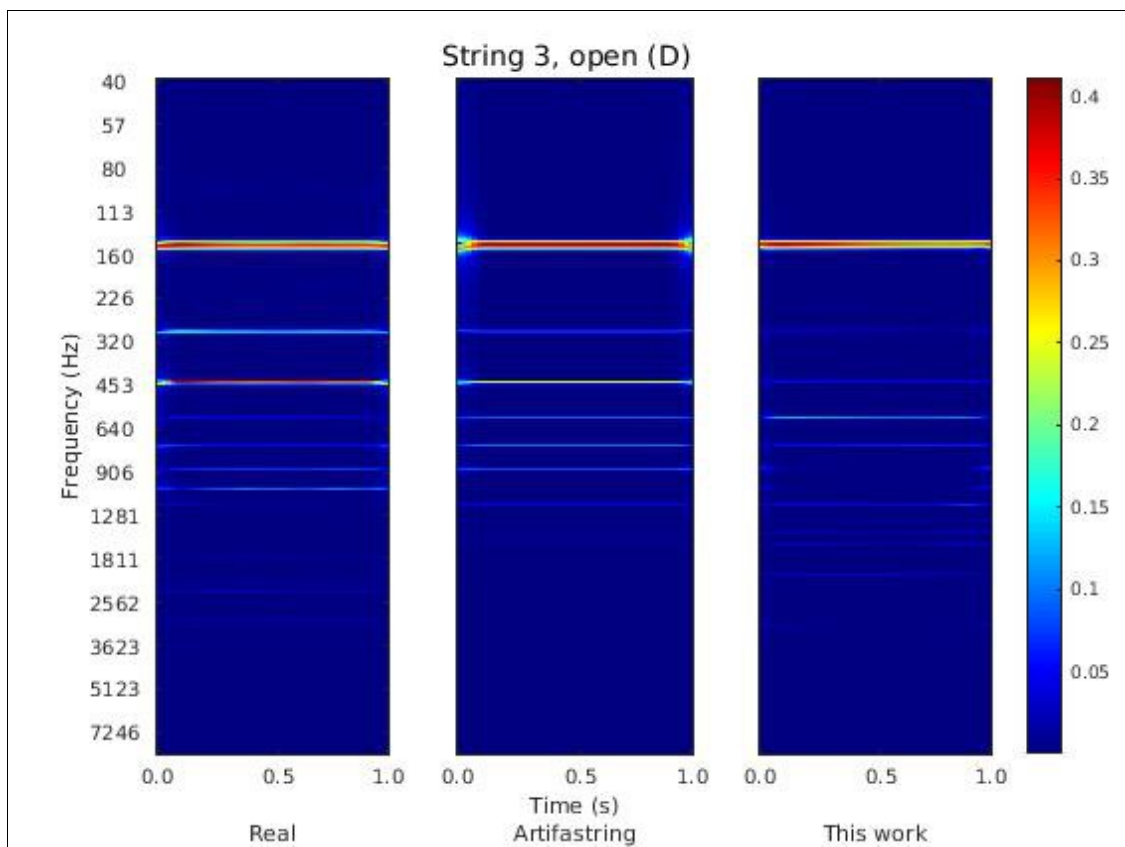


Figure 6.5: Spectra resulting from the open bowing of the third string (real, Artifastring and this work).

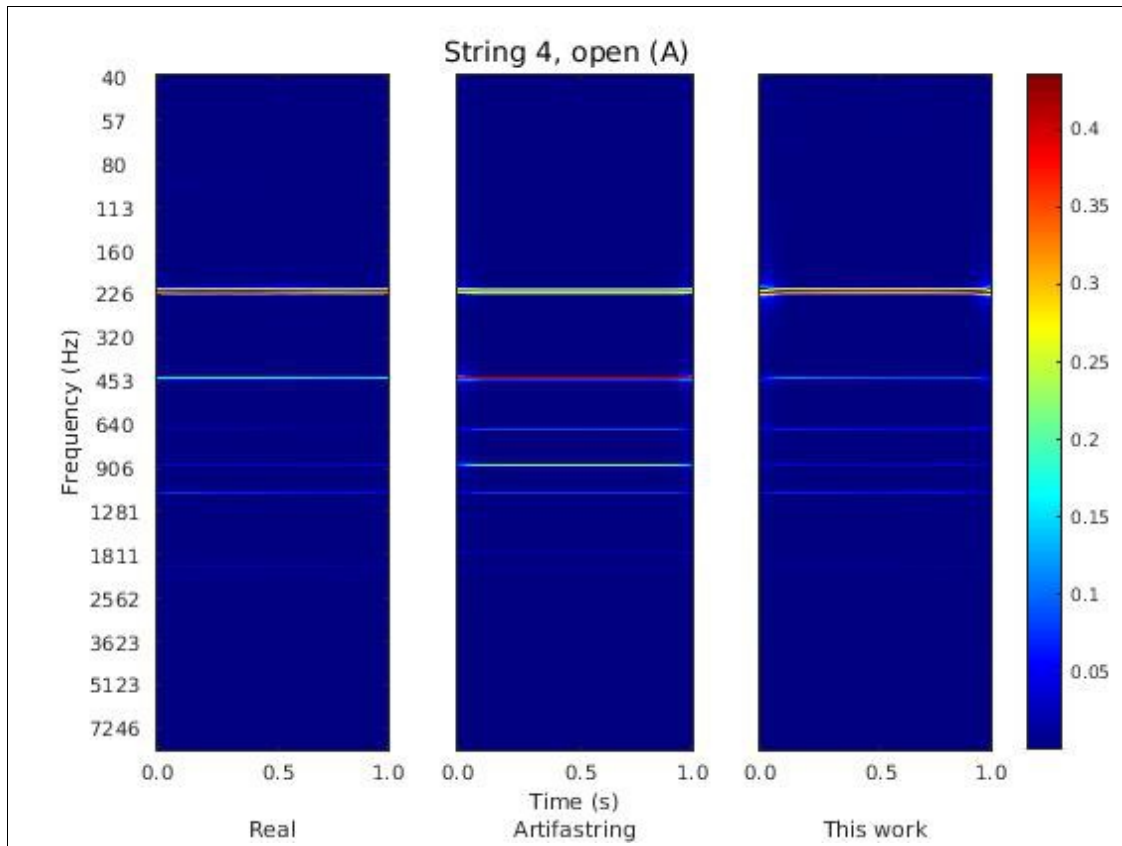


Figure 6.6: Spectra resulting from the open bowing of the fourth string (real, Artifastring and this work).

Comparison between the audio samples resulted in the following figures:

String no.	Note	This work (pre-optimised)	This work (post-optimised)	Artifastring
1	C 2	10.73	10.65	11.32
	G 2	12.32	11.26	12.75
2	G 2	11.90	11.43	12.54
	D 3	34.34	10.72	13.21
3	D 3	14.43	11.13	34.32
	A 3	14.11	10.31	13.89
4	A 3	14.17	11.91	14.20
	E 4	35.22	10.93	13.47
Minimum		10.73	10.31	11.32
Maximum		35.22	11.91	34.32

Table 6.1: The results (dBW) of steady-state comparison against real Recordings I.

Different results were obtained for the full sample comparison (including attack and decay):

String no.	Note	This work (pre-optimised)	This work (post-optimised)	Artifastring
1	C 2	12.10	12.24	12.43
	G 2	12.75	11.70	12.78
2	G 2	12.19	12.24	12.72
	D 3	12.37	11.47	12.32
3	D 3	12.81	12.40	12.40
	A 3	12.50	11.66	12.71
4	A 3	12.69	12.78	12.85
	E 4	12.91	11.27	13.45
Minimum		12.10	11.27	12.32
Maximum		12.91	12.78	13.45

Table 6.2: The results (dBW) of full comparison against real Recordings I.

Recordings II

Comparison between the steady-state portions of both *Artifastring* and this work against *Recordings II* yielded the results shown in *table 6.3*. Figures 6.7 through 6.10 show the different spectra of these samples.

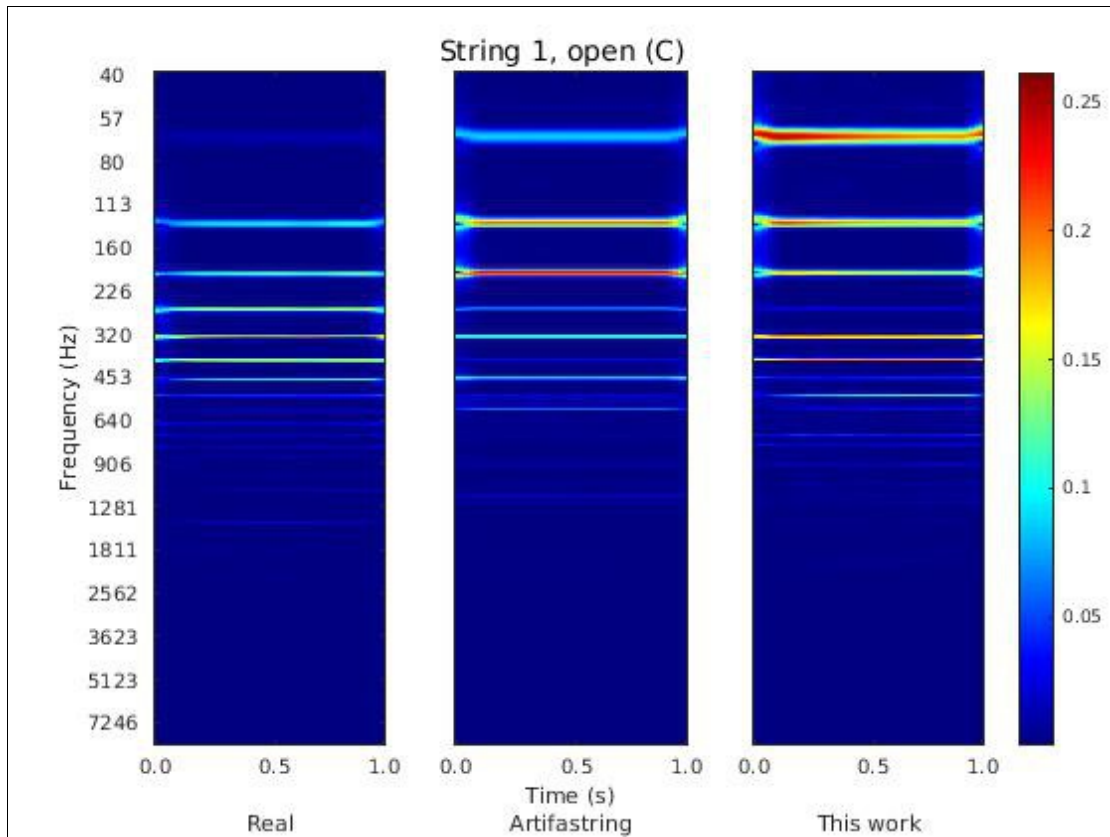


Figure 6.7: Spectra resulting from the open bowing of the first string (real, Artifastring and this work).

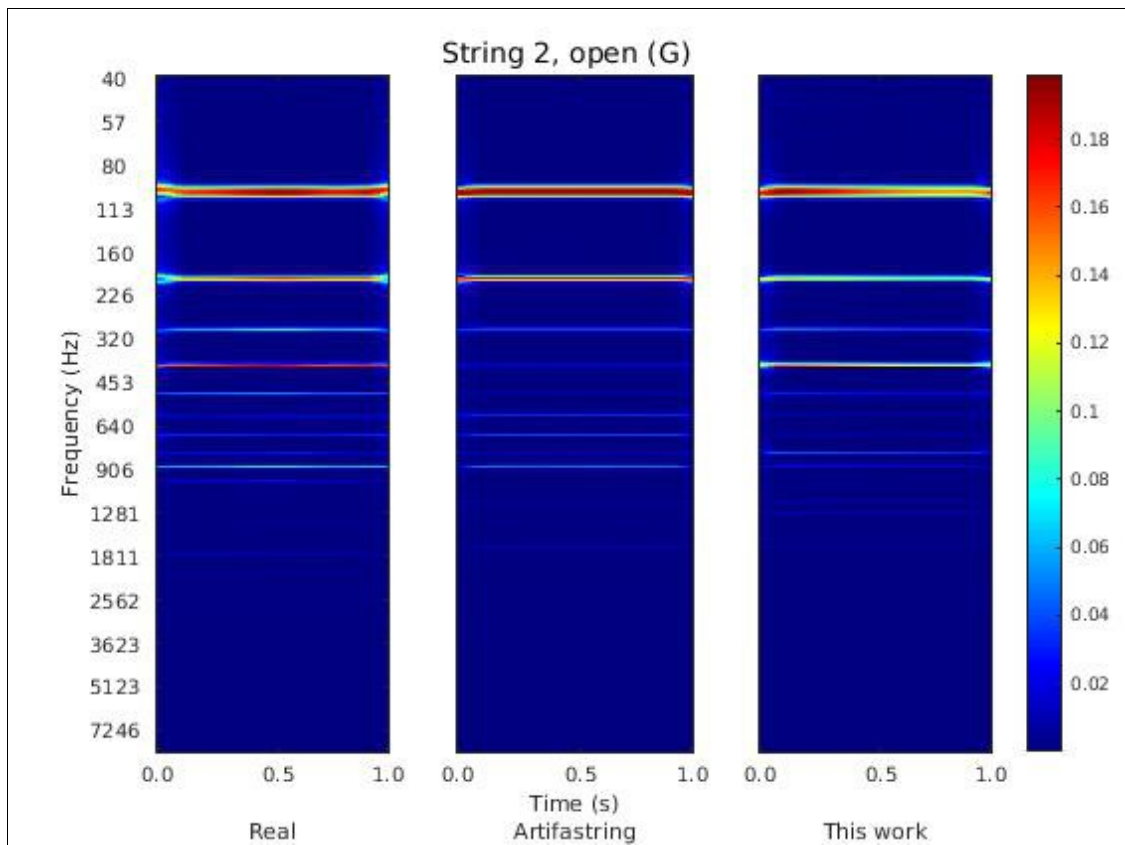


Figure 6.8: Spectra resulting from the open bowing of the second string (real, Artifastring and this work).

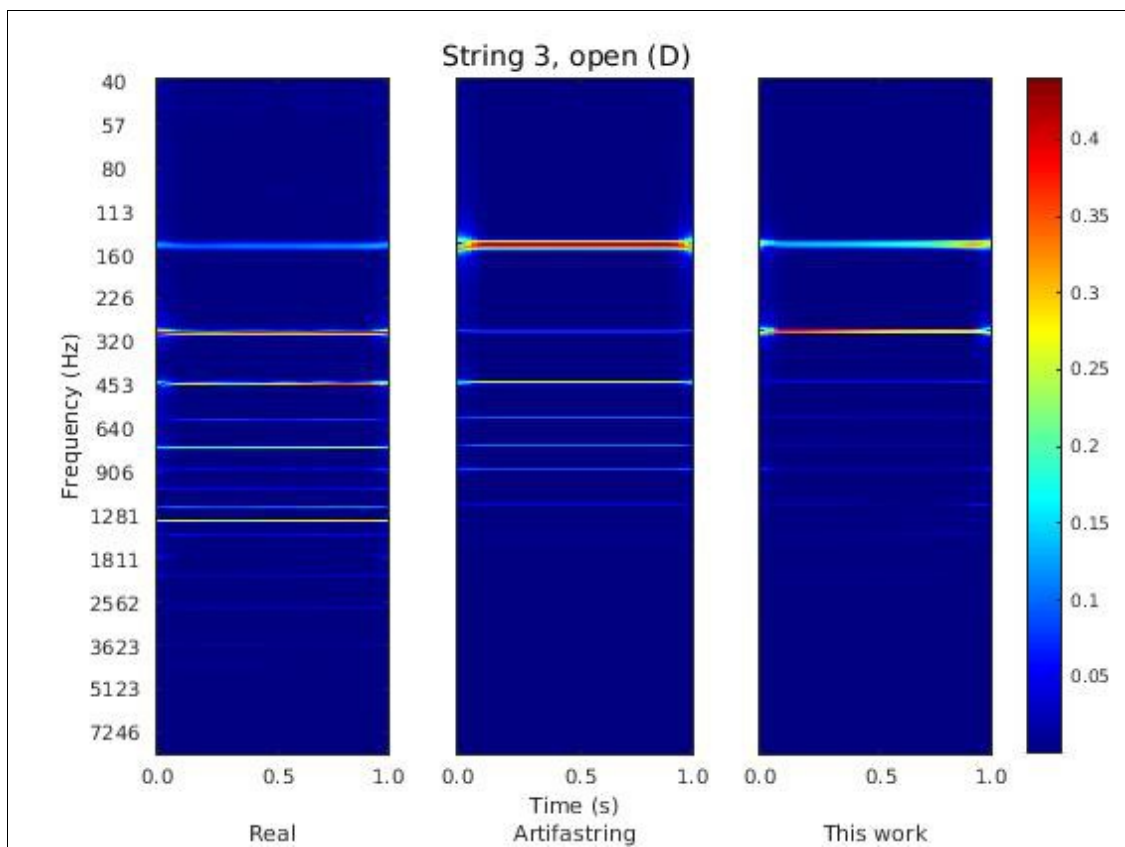


Figure 6.9: Spectra resulting from the open bowing of the third string (real, Artifastring and this work).

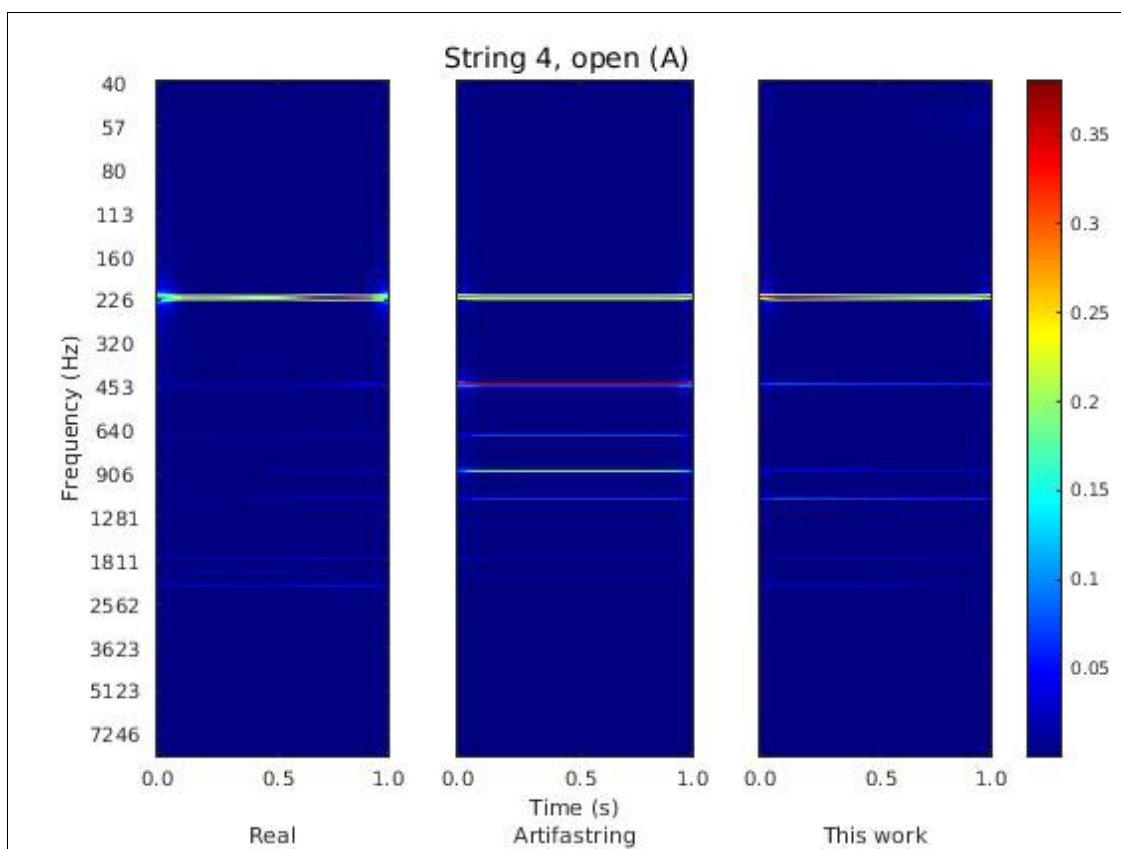


Figure 6.10: Spectra resulting from the open bowing of the fourth string (real, Artifastring and this work).

Comparison between the audio samples resulted in the following figures:

String no.	Note	This work (pre-optimised)	This work (post-optimised)	Artifastring
1	C 2	10.66	10.26	10.78
	G 2	10.84	10.56	11.53
2	G 2	11.22	10.33	11.17
	D 3	11.70	10.04	11.91
3	D 3	10.74	10.44	10.58
	A 3	13.27	11.00	13.39
4	A 3	12.08	10.74	35.80
	E 4	11.90	11.06	11.77
Minimum		10.66	10.04	10.58
Maximum		13.27	11.06	35.80

Table 6.3: The results (dBW) of steady-state comparison against Recordings II.

Similar results were obtained for the full sample comparison:

String no.	Note	This work (pre-optimised)	This work (post-optimised)	Artifastring
1	C 2	11.32	11.25	11.51
	G 2	11.89	11.45	12.19
2	G 2	11.95	11.54	12.04
	D 3	12.03	11.58	12.21
3	D 3	11.72	11.94	12.30
	A 3	13.00	11.85	12.57
4	A 3	12.24	12.06	12.39
	E 4	13.16	12.99	13.30
Minimum		11.32	11.25	11.51
Maximum		13.16	12.99	13.30

Table 6.4: The results (dBW) of full comparison against Recordings II.

Conclusion (metric comparison)

In every case, this work's sounds were closer to the target sounds than *Artifastring*'s outputs in a mathematical comparison of their spectra, with optimised output mainly giving the closest match. The only exceptions were a minority of the full samples where non-optimised sounds held the minimum value, although this is likely due to the inclusion of bowing attack and decay. Although such effects are important for perceived realism, because of their noise-like character and the fact these figures only represent differences in frequency content full sample results should not be taken as indications of matching quality of tone. This considered, the overall result was a greater match of this work to the real target instruments.

The average differences may be seen in *table 6.5*, and the success rate of this work in generating a closer match to the target sounds (against comparison of *Artifastring*'s output to the same targets) in *table 6.6*.

		Target sound			
		Steady-state		Full	
		Recordings I	Recordings II	Recordings I	Recordings II
Simulator	<i>This work</i>	11.04	10.55	11.97	11.83
	<i>Artifastring</i>	15.71	14.62	12.71	12.31
	Improvement	29.73%	27.84%	5.82%	3.9%

Table 6.5: Average differences (dBW) between simulator outputs and target sounds.

Comparison	Target sound	
	Recordings I	Recordings II
Steady-state	100%	100%
Full	100%	100%

Table 6.6: Proportion of notes showing improved match to target audio over *Artifastring*.

On average, this work presented a greater match to the target audio than the original *Artifastring*. However, its frequency content during steady bowing could be more closely matched as the comparison results show. This is likely due to approximation of the convolution vectors used within the cello simulation, and might be solved by development of an additional frequency domain correction or more accurate initial convolution vector calculation.

Subjective testing

A three-part subjective test was used to gauge the perceived realism of the simulated sounds. Thirty-three participants took part, of which twenty considered themselves musicians or in possession of refined musical taste. Participants were played 32 sounds from a mixed group of real and simulated samples, the first 16 containing only the steady-state part of the sounds, the remaining 16 including bowing attack and decay. For each sample, they were asked if they believed they had heard a recording of a real cello.

For the last part, participants were played a set of 8 sample pairs, one generated by the original *Artifastring*, the second by this work. Both samples contained a note of equal pitch and duration. For each pair, participants were asked to select the sound they perceived as most real.

The software used to generate the tests may be seen in *Appendix A*. Individual notes played on all four strings were included in the test, using first, third, fifth and seventh notes of the open string's major scale, as were both *Recordings I* and *II*. The results were grouped into three categories: all, musical and non-musical.

Identification of real sounds

First, participants were asked to identify which of the sounds played to them were from a real instrument. To ensure the results obtained were not due to chance alone, a confusion matrix was constructed and the results were compared to chance using a χ^2 test, with a p-value of 0.05 (*table 6.7*). All groups validated correctly, and the resulting matrices may be seen in *table 6.8*, showing the number of users classifying each recording type as real or synthesised.

Results group	Threshold	Chi ² statistic
All	3.84	77.10
Musical	3.84	50.90
Non-musical	3.84	26.71

Table 6.7: Chi² test results (p-value = 0.05) for the confusion matrices shown in Table 6.8.

Actual sound	Participant-selected classifications per group					
	Musical		Non-musical		All	
	Real	Synthesised	Real	Synthesised	Real	Synthesised
Real	246	74	147	61	393	135
This work	82	85	42	56	124	141
Artifastring	77	76	53	57	130	133

Table 6.8: Confusion matrices for participant-chosen sound classifications.

Participants were able to correctly classify 74% of real sounds correctly. However, the correct rate of classification for simulated sounds was 52%. Classification error rates for both test types are shown in *table 6.9*.

Actual sound	Incorrect classifications per group			
	Musical		Non-musical	
	Steady-state	Full	Steady-state	Full
Real	15.94%	7.19%	21.15%	8.17%
Simulated	27.5%	22.19%	25.00%	20.67%

Table 6.9: Classification errors by test type.

Simulation preference

In the last part of the test, participants were asked to listen to pairs of simulated sounds played in a random order (one generated by *Artifastring*, the other by this work), and select which they perceived as most realistic (*table 6.11*). The results were compared to a binomial distribution with probability $p=0.5$, and it was confirmed that there was insufficient confidence (<95%) and only a small probability of their being the result of chance (*table 6.10*).

Results group	95% Confidence Interval of $p=0.5$	Actual result	Probability of result being due to chance [18]
All	0.44 – 0.56	0.61	2.67×10^{-4}
Musical	0.42 – 0.58	0.59	3.25×10^{-2}
Non-musical	0.40 – 0.60	0.65	2.20×10^{-3}

Table 6.10: Binomial distribution test results for participant preference figures shown in *Table 6.11*.

Simulator	Simulation output preference counts per group		
	Musical	Non-musical	All
This work	94	68	162
Artifastring	66	36	102

Table 6.11: Participants' choice of simulation output, shown by group.

Table 6.12 shows the proportions represented by these results.

Simulator	Simulation output preference per group		
	Musical	Non-musical	All
This work	58.75%	65.38%	61.36%
Artifastring	41.25%	34.62%	38.64%

Table 6.12: Participants preference of simulation output, in percent.

Conclusion (subjective evaluation)

Participants' inability to correctly classify simulated sounds with the same success rate as real sounds shows that a difference exists between the perceived realism of these two types. This difference is due cause to suggest listeners may have difficulty classifying simulated output as synthesised (as would be the case if such output were entirely realistic), although there is clearly room for improvement since the rate of misidentification of simulated output is not as high as the rate of correct identification of real recordings. A good measure of effective simulated realism may be a close correlation between these two figures.

Bow attack and decay is clearly an important factor in perceived realism, its inclusion more than doubling the number of times real sounds are identified correctly. However, the same did not result for simulated sounds, suggesting perhaps the modelling of bow attack and decay requires further attention, although it is already known that *Artifastring's* engine “does not represent the most accurate simulation known to researchers” [20].

The first two test sections did not address differences in realism between the two simulation methods. However, results from the final test showed participants noted a significant difference between them, and mostly preferred this work to that of the original *Artifastring*. Since the figures obtained are shown not to be coincidental, this is reasonable grounds to suggest the features implemented by the project have increased the perceived realism of the simulation.

Demonstration audio

A set of demonstration sounds, comprising a selection of target, pre-optimisation and post-optimisation audio used for evaluation, may be heard by downloading an archive containing a simple web-page at the following URL:

https://www.dropbox.com/s/3pge6ery23y5gl4/c1322278_demo.zip?dl=0

7. Future Work

Future work should improve the realism of the simulation, or enhance the current understanding of the physical instrument and what is required to accurately represent it. The present engine could be altered to include the latest research, or perhaps a new engine developed which models this behaviour as a starting point, and findings from this work applied to enhance it.

Comparison and analysis

The simple mathematical comparison of frequency components between two sounds used for this project cannot allow quantification of quality of tone and time domain effects (such as bowing attack and decay) with clarity. Study and development of a metric set which might be used for this purpose would make analysis of the results more effective, and increase the optimiser's ability to match simulation parameters to target sounds.

The human ability to subjectively evaluate sound could also be leveraged to improve the understanding of perception of quality of tone, and guide development of more realistic simulation techniques. Tests could be devised to analyse how sonic components are related to perceived realism for the specific instrument, and research and development work focussed on the areas with greatest significance. Potential improvements could be trialled and optimised or avoided depending on the results of user testing, and differences between perceived real and non-real sounds used to identify target components for accurate simulation.

Frequency domain correction

As shown during analysis of the results, discrepancies remain between target and simulated sounds constructed using the convolution vector technique devised. Slight differences in pitch between target and raw string sounds resulted in different simulator output, suggesting flaws in the accuracy of the vector produced were likely related to lack of accurate correlation between simulated and target sounds.

Development of a dynamic frequency domain post-convolution correction which does not eliminate time domain effects would provide a significant improvement to the tonal match between target and simulated audio, and may be extended to allow the complex sonic characteristics of effects such as bow attack and decay to be emulated for an instrument instance. Indeed, the use of convolution in the time domain might potentially be replaced by an accurate and transparent frequency domain matching technique.

Realistic transitions

The addition of smoothly but randomly varying bow and finger characteristics may have contributed to participants' preference for output generated from this work. At present, no mechanism exists within this code for creating realistic transitions between finger positions, actively energised strings, and bowing changes such as angle of incidence, direction and inter-string transitions. Actual instances of these transitions could be analysed, and their behaviour added to the model so that realistic output would result even from attempts to unrealistically operate the virtual instrument.

It would also be possible to replace the simple oscillators employed in the creation of force variations with functions which replay and blend control signals based on recorded data from actual

physical interactions. Bow surfaces could be mapped for texture and arrangement of fibres, minor finger position changes measured and the characteristics of rosin coatings examined. These factors could all be modelled to accurately reflect their physical interactions, and the resulting step-wise variations fed into the simulation. More realistic effects might be achieved using this method over the simplistic frequency-varying sine waves currently used.

Optimiser extension

The current optimiser design only reports the discovered optimal settings for a given note and parameter set. However, ideal settings are shown to vary from note to note for any given string. The simulation could be extended to allow transitions between settings (in the same manner as its current interpolation between convolution vectors), and the optimiser altered to provide parameter data in a format which could be used directly by the string engine.

Machine learning could potentially be employed to aid in the comparison process, and weight the results returned to the optimiser by classifying them in terms of desirable qualities. This would allow automatic detection of erroneous output, and promotion of traits sought during comparison, enhancing the ability of the optimiser to select accurate and high-quality matches.

In general, the criterion for accuracy could be changed so that every generated sound should be close to that of a realistic target, whereas currently only typical sounds are focussed upon. Extending the optimiser's function in this manner would form part of this improvement in accuracy.

Performance

Execution speed for the Matlab implementation remains poor. A quality virtual instrument is most useful as a performance tool when operating in real-time, a condition which should form a primary goal of such a simulation. Since execution speed for the C implementation of *Artifastring* is good, implementing the simulation's code in C or a similar high-performance language would likely greatly improve its performance. An effort could be made to produce a real-time implementation of the current system in this way.

Realistic input controls

If such an implementation were realised, a study of physical interactions between human and instrument could be conducted, and an input control system devised (or an existing system used) to provide the performer with the ability to realistically interact with the simulation to produce output, in the same way as a real instrument may be played. Further perception tests may then be designed using human performances, potentially enhancing an understanding of the perception of realism in relation to instrumental recordings, and opening new avenues for development of the simulation where further improvements are possible.

Finalisation as a virtual instrument

Ultimately, the simulation may be developed, refined and tested to the extent that it functions fully as a real-time virtual instrument. Its functions may then be ported to various platforms, and user friendly front-end designs or method interfaces designed to allow its use in real-world situations, either as stand-alone software or as a plug-in or extension for popular music production systems.

8. Conclusions

Accurate modelling of a stringed instrument is a complex task. A great many factors affect the timbre and temporal reactions of the instrument, and average members of the population were able to identify differences between real cello sounds, and those based on the *Artifastring* engine.

However, realism was shown to be improved by changes made during development. The first of these was the introduction of non-linear variations to important simulation parameters. These variations helped to represent naturally arising changes in density of bow fibre or layering of rosin on strings [7], and human interactions causing subtle changes in bowing pressure and force or finger placement. Such steady variations, instead of fine-grained noise (such as was present in the original *Artifastring*), help convert periodic artefacts that the human ear identifies as mechanical into diffuse effects more reminiscent of a physical instrument.

The second change was use of a convolution vector calculated from the difference between raw simulation output and target audio samples. This process helped reduce the distance between real sounds and simulation output by providing a change in timbre designed to approximate that of a real instrument, achieved by tracking changes in pitch and interpolating between members of a set of responses. The timbre of simulated sounds, and subtle variations in the parameters used to create them, affected perceived realism enough to be detectable during user trials. Optimisation further improved the correlation between output and real recordings, although no complete match was identified mathematically.

Differences between the results of the mathematical and subjective tests may be significant. The vast majority of simulation outputs were closer by mathematical comparison to the corresponding target sound, yet approximately half of the time users were able to identify them as computer generated. This suggests that quality of tone is not quantifiable by frequency content alone, and that it may require a more complex metric than the simple difference function used here.

Development of a versatile method of frequency domain transformation which retains time domain effects may be an important area in the physical simulation of musical instruments, since it may be used to apply the tonal qualities of a real instrument to one that is generated computationally. More research in this direction may be highly constructive in the effort to develop highly realistic simulated instruments.

Complete tonal realism is granted by the use of sample-based simulation, within the limits of its design, since actual recordings of the instrument in question are the foundation of its output. However, the range of possibilities and effects available within the physical simulator, when combined with techniques which bring its quality of tone closer to actual instrument instances, render it an important and versatile model in the search for virtual instruments which are wholly indistinguishable from the real thing.

9. Reflection

Skills and knowledge

I learned much during this project. At its inception, I had no knowledge of physical synthesis and only a general idea of creating physical models. Studying textbooks and research papers gave me a great deal of knowledge, and without this background information I would have been unable to generate a design and begin an implementation. Concepts new to me included the use of derivatives, matrix operations and the wave equation, which I was able to gain an understanding of during research conducted at the beginning of the project. Through its implementation, and this report, I have gained knowledge of the composition of simulated physical and sample-based virtual instruments, and the basics of extracting relevant information from scientific documents and constructing a technical report.

I developed many practical and academic skills, which I hope to build upon in the coming years. In particular, I am now able to produce Matlab code, and I have an appreciation of techniques for quick prototyping and optimal design, including Matlab classes, vectorisation and in-line instruction blocks. I am now able to design evaluation functions and have learned how to use these in the application of optimisation routines for common minimisation problems.

Overall, my experience of research material, scientific documentation and mathematical tools has improved significantly, and I believe the skills I have gained in their analysis and application will be extremely important in my future research and development work.

Choices and contributions

Many choices had to be made during this project, especially in response to the various challenges posed by the work and practical limitations. Challenges also came from the development equipment itself. I suffered two technological failures at critical points; the first was the sudden failure of my development machine's system drive, the second dry joints in the power circuit of my office tools laptop. The drive failure did not cause loss of critical data, since I make periodic project backups, a practice I will maintain for all future work.

However, repairing or replacing the drive would have taken valuable time needed to produce audio for the subjective tests. Instead, I decided to re-partition my data drive and perform a minimal install of Gentoo and Matlab to quickly restore the operation of the simulator. This proved to be a successful option, and fortunately the backup plan (purchasing another drive at expense, or waiting for a cheaper unit to arrive before continuing) was not necessary. Similarly, quickly replacing the laptop's power socket with a spare part purchased previously (for this eventuality) enabled its continued operation for long enough to complete this report.

I had initially hoped to produce a functional, real-time virtual instrument by the time of the project deadline. This was not possible due to the volume of work required to create a functional string simulation, develop techniques to improve realism, build a versatile human interface and transform the code to a real-time form. It was necessary to decide which parts of the project to prioritise and which to defer to a future time. I had expected progress to be slow, but not as difficult as it was in parts. This experience has confirmed my understanding that generally development work is more complex than at first appears, and so realistic targets must be set and sufficient time factored in to allow for this. Retaining focus on improving the realism of a physical instrument, as the brief specified, was important in progressing towards the goal.

I believe the choices made during the project enabled it to be completed to a reasonable standard, something which may not have succeeded had I attempted to set extensive targets or not pushed development ahead as far as I believed possible. Without continued daily progress, the project would have failed to complete – I understand this is a vital factor which will apply to my future projects also.

In contributory terms, I hope that I have presented ideas which may be built upon to improve the realism of future virtual instruments. However minor my contribution may be, I recognise that all progress may help towards attaining a common goal. This project has taught me that research should be as extensive as possible, but even small improvements to current knowledge are important for their part. Most importantly, I have learned that research should be validated by good scientific practices, and in future I will definitely be striving to develop comprehensive metrics (and extensive user testing, where relevant) to maximise the possible analysis and understanding of the results, something which I believe could have been improved during this project with hindsight. In particular, evaluation during implementation would aid in the selection of positive development angles.

Design challenges

Preparation

This project presented many design challenges. Professional virtual instruments are often created by a team of seasoned developers, and existing physical models such as those by Bilbao, Desvages [2] and Percival [21] are typically created as part of research during a PhD study three to four years in duration. I decided to research background theory and implementations early on, and seek ways in which to accelerate the development process and arrive quickly at the project goals. Conducting initial trials was vital in ascertaining which approaches were possible within the given time-frame and were likely to produce output of the desired fidelity.

A number of methods were used as preparation for the implementation of the project's design. Regular meetings with the project supervisor were held, where ideas and possible development angles (such as methods of solving the systems outlined under “*Relevant theory*”, utilising existing code and optimisation possibilities) were discussed. Initially, the most promising model seemed to be that developed by Bilbao and Desvages [2], whom I contacted to ask if an implementation existed publicly. Unfortunately, it did not. Other implementations were sought, and *Vibrating String Simulator* [27] and *Artifastring* [20] identified as potentially suitable bases for the initial trials, and possibly for the simulation's final engine.

My personal preparation included exploration of finite difference modelling, Fourier analysis, digital signal processing and Matlab programming from sources such as *Numerical Recipes: the art of scientific computing* [23], *DAFX: digital audio effects* [31], *Mastering MATLAB 7* [8] and lectures on Fourier analysis [12], from which I quickly gained the foundational knowledge needed to begin development.

Given the probable complexity of a physical simulation [2], the work required of the optimiser might have proved extensive, especially since a great many parameters are available. Carefully selecting and grouping these by function helped to minimise the time expended solely on optimisation runs. Simultaneously development of the overall system and execution of the optimiser where practically possible also helped in the meeting of deliverable deadlines.

Initial trials

The purpose of the initial trials was to discover whether use of a physical model was feasible for this project, given the resource limitations. If progress appeared too difficult, or too slow, a basic sample-based model would be constructed, the outputs of both trial versions compared, and the most promising candidate selected for further development. If the physical model proved feasible, it would be given priority for development.

Vedenyov's *Vibrating String Simulator* [27] was selected as a basis for the first trial attempt, since it is written in Matlab and produced a realistic (if metallic) output in response to displacement of its string's position. However, it did not feature a method designed to produce a bowing effect.

Instructions were added to the main loop of *Vibrating String Simulator* to simulate a rudimentary slip-stick action (*psuedo-code* 9.1).

```
for step in simulation_time:
    % added code
    if( step mod bow_cycle_time < bow_cycle_time / 2
        and step < stop_bowing_time ):

        string_y( step, centre ) = string_y( centre ) + 1;
    % end of added code
    y( step, all ) = calculate_new_y_positions( previous_y_positions );
```

Pseudo-code 9.1: Basic bowing action added to *Vibrating String Simulator*

Three tensions (220, 420 and 1220 Newtons) were used to produce three output sounds. Initial results were promising – the audio resembled a bowed string, inasmuch as the note had steady attack and exhibited smooth decay once the time set for the bowing to end had elapsed. Spectrograms were used to compare the output against that of the real cello, playing the same note. Although clearly not entirely accurate – especially to the subjective listener – there was a reasonable correlation between the frequency content of the sounds.

Other methods of engine production were also considered. An attempt made to produce a solution to equations 3.4a and 3.4b in Matlab using the finite difference method. However, after several days the basics of these equations had not been successfully modelled, and due to the level of complexity I estimated I would need to invest several weeks at minimum to complete this task. Following discussions with the project supervisor, I decided that more efficient progress was necessary.

The C source code for *Artifastring* [20] was downloaded and compiled. It was immediately obvious that the output produced using the example cello actions file was far closer to the real cello recording than that of the modified *Vibrating String Simulator* [27]. Due to this, and because it offered an excellent opportunity for further development of Percival and Demoucron's model, it was selected as the core physical engine for the project's simulator following deliberations.

Attempted solutions

Attention was turned to the body simulation. A bank of DSP parametric equalisation filters was prototyped, and tuned to approximately represent an instrument body response curve and decay time. The output of a single string was passed through these filters, and the end result compared to real cello recordings. It was immediately obvious that the audio produced did not resemble that from a real instrument, and that a great many filters would be required to accurately match the

complex responses exhibited, producing slow execution and likely unrealistic results.

An alternative method of body simulation was sought. An impulse response recorded from a real cello [14] was convolved with the string's output (*pseudo-code 9.2*), and the end result compared to a real instrument. This time, the result was clearly a great deal closer to the target audio, and appeared to contain a certain degree of room colouration as well as cello body colouration, as in the target recordings. Use of convolution to achieve colouration generated by both instrument body and room characteristics was selected as worthy of further development.

```
x = audio( 'string_output.wav' );  
c = audio( 'cello_impulse_response.wav' );  
  
y = convolve( x with c );
```

Pseudo-code 9.2: Applying impulse response by convolution

The body response of a real cello varies with frequency of excitation, and so any response applied as part of the simulation must operate similarly. Assuming that a convolution vector exists which may produce the correct impulse response when a given note is played on a particular string of the instrument being modelled, a system of linear equations exists for which a matrix may represent the solutions for all notes under analysis. The proportional mixing of a set of closest convolution vectors was selected as a possible computational solution, a decision which proved fruitful as the project progressed.

In case further ambience processing would be required, the *Freeverb* algorithm built into Matlab was briefly applied to the output of the convolution process to verify it as a possible source of reverberation. As expected, the result was a mixture of dry and reverberant sound, and *Freeverb* was bookmarked as a potentially useful algorithm, although the convolution method finally selected negated the need for it.

An effort was made to construct a frequency domain transform which could correct the deficit in the convolution method's output, and which would not cause the obliteration of time domain effects such as bowing attack and decay. The resulting prototype involved active cross-fading between the transformed FFT and simulator output FFT based on an amplitude threshold, thus retaining the original attack and decay before the threshold was reached. However, this technique did not allow the retention of time domain effects or support alterations in pitch once the threshold had been crossed. Lacking the time needed to further develop it, this prototype was consigned to future work.

Optimisations

Several attempts were made to increase the performance of the Matlab code. The first version of this code implemented a single string simulation as a Matlab class, so that all four instrument strings could be modelled simultaneously. At this point, a for loop over the number of strings handled their computation. In an attempt to improve performance, the *parfor* construct was used to parallelise the isolated string instances. Unfortunately the overhead generated increased execution time, and the change was reversed.

Following recommended Matlab coding practices [15], the simulation's engine was converted from a method-rich class with individual properties to functions operating on values held within a single vector per string (*pseudo-code 5.1*). This reduced the output to run-time ratio from 1:300 to 1:50 for a single string, however for simulation of all four strings this increased from 1:229 to 1:269. Code

was moved from individual functions into a single in-line block with multiple strings indexed by for loop, thus eliminating function calls within the simulation's main loop. This drastically reduced the output to run-time ratio to 1:45, and made proper use of the optimiser feasible (since each output generation by the optimiser now took under 34 seconds).

Experimentally, the existing single vector per string construct was replaced by a set of matrices, each representing an individual variable or array used by the engine. Its code was altered to allow each set of instructions to process all four strings simultaneously. Since certain code sections operated on a string only under particular conditions, masks were used to select or de-select columns of a matrix for computation. This often required a method for applying a mask vector of length 4 (each element representing a string) to a matrix of much greater proportions, such as the mode velocities matrix of dimensions 96x4.

Matlab's *repmat* statement was used first, to extend the dimensions of the vector to match those of the matrix. Unfortunately this greatly increased the execution time, and application of *bsxfun* or *mtcol* instead resulted in a similar calculation overhead. *Mex* was used to create a C executable to perform the task efficiently, and although it executed quickly by comparison to other methods it could not rival the performance of the existing structure (a for loop updating each string vector individually). From examination of Matlab's profiler, it seemed probable that the isolated nature of each string already allowed the parallelisation of their calculations by Matlab at run-time. Since the engine's performance was not improved, the changes were reverted.

Where parallel vector computation were possible by Graphical Processing Unit (in calculations such as FFT and iFFT) the `gpuArray()` method was used to facilitate this. However, any increase in processing performance was overshadowed by the time required to copy the vector to and from GPU memory, and execution speed was again degraded. This change was also reverted, and the non-GPU-based in-line vector per string construction chosen as the final implementation method.

Progress monitoring

The Iterative Waterfall method of development was selected as suitable for this project, since it is predominantly research-based. This enabled a number of development cycles to occur within a limited period, and maximised the possibilities for revising existing code and following potential new design angles. Several milestones were identified as targets for completion of individual system components (initial test builds, string simulation engine and optimiser), with progress monitoring and supervisor feedback providing guidance and commentary during development.

At the end of the first development iteration, it was agreed that although good progress had been made, the currently simplistic method of body simulation via a bank of equalisation filters needed further work, and that the output possessed a metallic-sounding distortion, seemingly originating from the string simulator(s). Optimiser generations did not exhibit the anticipated range of tonal variation, and additional parameters should be sought for optimisation.

These changes were addressed during the second iteration, at the end of which it was agreed that the use of pitch-linked convolution vectors produced a good subjective match to the target sounds, and the addition of a damping vector optimiser routine was producing better variation in the output audio. However, the system had not yet been tested with a different target set, and generation of each simulation output remained slow, making the optimisation routines' execution time too great to be of practical use.

By the end of the third development iteration, it was agreed that the system was generally functioning as desired. However, its execution time remained high by comparison to *Artifastring*, and results had yet to be computed by the optimisation methods. Spectrogram comparison showed the output did not exactly match the target input sounds, a desirable feature if attainable.

During the final iteration, these issues were examined and solutions found where possible. Restructuring of the code had reduced the run-time to an acceptable level, and optimised parameters had been successfully used to generate audio for the subjective tests. But although a complete match of output to target sounds had been attempted, it was unfortunately not possible to research and develop a method to achieve this given the allotted time and resources. It was agreed that the project goals had been achieved, and that future work could potentially further improve the system's performance. I believe assessment at key points in this manner, and discussing progress in depth with the supervisor during development, was instrumental in producing these positive results.

Goals and achievements

Although the primary project goal was to improve the realism of a physically simulated instrument, many sub-goals existed within this, contributing towards it as a whole. I had hoped to have explored every aspect of the features developed, and perhaps achieved more concrete results than were possible. If I were able to attempt the project again, I would like to focus on one specific feature and develop it thoroughly, so that my results might form a foundational part of a greater project.

Instead, I feel the features implemented here reflect a template on which such foundations may be built. The human-like variations added could be replaced with accurately modelled bowing and fingering changes, and the simple calculation of convolution vectors replaced by an accurate alteration of timbre. I believe the results achieved are reasonable for a first attempt at improving realism, and it was not possible to know during development what effect these additions would have. However, perhaps an early assessment could have identified a single target as most worthy of research. For future projects, I will make a specific evaluation of all available developmental paths at the end of each iteration, yet keep in mind that pursuing every angle might still produce an ideal focus area.

I feel my personal and academic sub-goals were reached to a good standard. Prior to this project, I had no knowledge of Matlab, advanced digital signal processing techniques or numerical optimisation techniques – quickly grasping these topics was very important, and I managed to achieve the basic level needed to continue the project. Each deliverable required a functional block composed of several different components: the core required a string simulation, body and room processors and the logic needed to integrate them; the optimiser required parameter calculations, an evaluation function and optimisation methods. I managed to implement these by the milestone dates set, and through the flexibility of both myself and the supervisor was able to further improve some components whilst simultaneously creating others, in an attempt to achieve the most effective implementation within the available time.

A serious hold-up occurred early on, when I was searching for a suitable string model on which to base the simulation. Fully understanding and implementing the most promising published models from scratch would have taken too long, possibly rendering the principal goal unachievable. However, a model whose source code was available was discovered through research, and I believe selecting this as the project's string model enabled all further development to remain on track, and contributed in a great way to its success.

This experience has shown me there is a huge difference between understanding another's research, and making a contribution of the same level and competence yourself. I now have a far greater appreciation of the importance of experience, the research process and careful structuring and recording of a project's progress, which I believe will be of tremendous importance to me in the future.

Personal strengths and weaknesses

I have gained a better understanding of my own strengths and weaknesses from the challenges posed by the project. Foremost, I am now confident of my ability to work independently and to invest all my resources and effort in the attainment of a goal. I believe my ability to identify targets necessary to this end, and to plan progression in a constructive and realistic way, are strong points that will definitely aid my future work.

However, my lack of background experience in science and mathematics was problematic, and often caused me to suspend project progress in order to grasp necessary (and sometimes simple) techniques and concepts. I am concerned that the results were not as comprehensive as they could be due to this deficiency. I recognise that self-study and experience are vital in addressing this, and will be conscious of including these as important aspects of ongoing learning as part of my personal development.

I also noticed I have a tendency for narrow focus when I am working hard on solving a difficult problem, often based on an aspect of a subject I am unfamiliar with, which can cause me to lose track of the bigger picture and fail to make decisions best for the project overall. Identifying this will help me stay aware and prevent my weakness from jeopardising future work, for which decisions affecting a greater scope may be more important than those best for individual sections.

Professional development

I feel I have benefited greatly from this project in professional terms. Its subject matter have exposed me to researchers and their work, professional tools and techniques, and scientific documentation and methods, many directly connected to the field I am endeavouring to build my academic career in. It was fascinating and helpful to speak personally with Professor Bilbao when discussing his model, and this has helped me realise that even those with extensive experience and knowledge are often happy to provide advice or assistance to those working on minor projects.

I have learned that Matlab features many built-in and third-party tools that are regularly used in professional research and development, and have gained invaluable experience of their use. I have also learned how to use research databases to discover relevant papers, and how to interpret them in the context of my own goals. The knowledge I have gained with regards to simulations, wave physics and important mathematical techniques have contributed strongly to my skills base, and helped provide me with tools necessary in a professional workplace.

I have also had the opportunity to refine my time-management skills, which I believe will be very important indeed for my future work. This has helped me continue to develop a professional work ethic which will sustain me through the challenges, failures and successes I feel sure future projects will produce.

Future skills and learning

My knowledge of mathematics and physics would benefit from further study, and I believe I could certainly improve on my ability to produce technical reports. Experience will be invaluable in this endeavour, and I plan to personally study these subjects and build up a comprehensive understanding from textbooks, online resources and discussions with other lecturers and students, whilst continuing to conduct work that will help me progress in my chosen field. I have observed a big difference between designing and actually implementing a research project, and will seek advice and greater experience of how to accomplish the latter to increase the chances of success for my future projects.

I was quite surprised by how critical time-management was for this project, since only one or two minor hold-ups may have caused important deadlines to be missed, likely resulting in failure. Clearly maximising the progress possible in any given period is an important and desirable skill, and I believe the experience I will gain as I move forward will contribute strongly towards my ability to do so. This project has given me a great opportunity to continue developing in this regard, and again I believe this will prove invaluable in the future.

Outcome

Overall, I believe the project was as successful as possible given my current skill level and experience, and the primary goal was achieved. Had I been able to continue development for a greater period of time I believe I would have been able to achieve more, but nonetheless it was a worthwhile experience and I am pleased with what I managed to accomplish. In terms of personal development, it has been invaluable. With reflection, many differences may have been made, and yet many decisions were instrumental in the positive outcomes, as discussed above. I am very excited about my future work, and feel certain that lessons learned from this project will form a great part of my ability to do well in all their tasks.

Appendices

Appendix A: Source code

```
% *****
% CELLO SIMULATION
%
% Name:    CelloSim.m
% Date:    02/05/2016
% Author:  Dave Humphreys, c1322278
% Desc:    Main entry point for cello simulator.
%           When executed, generates a simulated
%           cello output in 'output/output.wav'.
%           Parameters may be set in CelloSim.h.
% *****

% Clear any existing data from the workspace
clear;

% Load simulation variables
CelloSim_h();

% Setup and Run Simulation
CelloSim_run_sim();

% Display final status
CelloSim_final_status();
```

CelloSim h.m

```
% *****
% CELLO SIMULATION
%
% Name:    CelloSim_h.m
% Date:    02/05/2016
% Author:  Dave Humphreys, c1322278
% Desc:    Main parameters for CelloSim_engine.m,
%          allowing changes of active string and
%          finger position, activation of
%          variations etc.
%
% *****

%% CelloSim simulation variables

sim_time    = 3.0;           % seconds
fs          = 44100;         % sample rate (Hz)
buf_len     = 8196;          % samples buffer accepts before disk write
num_strings = 4;             % for IR_*
num_files   = 4;             % for IR_* - number of note groups per string
num_levels  = 2;             % for IR_* - number of amplitude samples per string
%
notify_div  = 20;            % 1/notify_div of a second between progress updates
str_gain    = 1.0e-14;       % string feedback gain
body_gain   = 1.0e-13;       % body feedback gain
mix_ratio   = 0.6;           % ratio of body to strings in final output mix
peak_ampl   = 3;             % peak amplitude of string, approximately
max_conv_ch = 1/(fs * 0.1); % maximum convolution vector alteration amount per step
% basic bowing / fingering controls
active_string = 0;           % set 0-3 for strings 1-4
active_pos    = 0;           % position 0 = open string etc.
stop_bowing   = true;        % stop bowing during simulation
stopbow_pos   = 0.85;        % stop bowing this far through sim_time
% 'realism' control flags
bow_noise_active = true;
vibrato_active   = false;
vary_bow_force   = true;     % parameters for all
vary_bow_velocity = true;    % these may be set within
vary_finger_pos  = true;     % String_h
% melody (effects etc.) control
active_melody = 0;           % 1 = glissando
```

CelloSim_run_sim.m

```
% *****
% CELLO SIMULATION
%
% Name:    CelloSim_run_sim.m
% Date:    02/05/2016
% Author:  Dave Humphreys, c1322278
% Desc:    Called by CelloSim.m; initialises the
%           components of the cello simulation,
%           calls CelloSim_engine.m, and does
%           some finalisation on its completion.
%
% *****

% Initialise all strings
Strings_init();                                % indexed 0-3
% Initialise convolution vector extractor
IR_construct();                                % indexed 1-4, for now
% Initialise body convolution code
Convolve_construct();
% Initialise FFT velocity feedback code
FFTv_construct();

% Set initial state
CelloSim_initial_state();

%% RUN SIMULATION
CelloSim_engine();

%% FINALISE
CelloSim_finalise();
```

CelloSim_final_status.m

```
% *****
% CELLO SIMULATION
%
% Name:   CelloSim_final_status.m
% Date:   02/05/2016
% Author: Dave Humphreys, c1322278
% Desc:   Display the final status of the string
%          buffers, on completion of
%          CelloSim_engine.m.
%
% *****

% Display final status
fprintf( '\nMaximum / Mean levels:\n' );
fprintf( '1    = %f\t/ %f\n', max( abs( string1 ) ), mean( abs( string1 ) ) );
fprintf( '2    = %f\t/ %f\n', max( abs( string2 ) ), mean( abs( string2 ) ) );
fprintf( '3    = %f\t/ %f\n', max( abs( string3 ) ), mean( abs( string3 ) ) );
fprintf( '4    = %f\t/ %f\n', max( abs( string4 ) ), mean( abs( string4 ) ) );
fprintf( 'body = %f\t/ %f\n', max( abs( body ) ), mean( abs( body ) ) );

% Plot the computed waveform
plot( output );

fprintf( '\nDone.\n' );
```

CelloSim_initial_state.m

```
% *****
% CELLO SIMULATION
%
% Name:    CelloSim_initial_state.m
% Date:    02/05/2016
% Author:  Dave Humphreys, c1322278
% Desc:    Set up the initial state of the buffers,
%           and bow and finger parameters, for
%           CelloSim_engine.m
%
% *****

%% Set string states

% Set all strings to released (free) mode
for s=0:3
    s_str( o_str * s + i_str_actions ) = e_str_RELEASE;
end

%
s_str( i_str_curStr ) = active_string;
stopbow_string       = active_string;

% Set one of a set of known good parameters:
if( active_string == 0 )
    % String1, open (C): bow( 0.12, 5, 0.4 );
    bow_ratio_from_bridge = 0.12; bow_force = 4.0; bow_velocity = 0.4;
end

if( active_string == 1 )
    % String2, open (G): bow( 0.12, 4, 0.4 );
    bow_ratio_from_bridge = 0.12; bow_force = 2.0; bow_velocity = 1.0;
end

if( active_string == 2 )
    % String3, open (D): bow( 0.12, 3.5, 0.4 );
    bow_ratio_from_bridge = 0.12; bow_force = 4.0; bow_velocity = 0.8;
end
```

```

if( active_string == 3 )
    % String4, open (A): bow( 0.12, 2.5, 0.4 );
    bow_ratio_from_bridge = 0.12; bow_force = 1.5; bow_velocity = 0.8;
end

% Set finger positions, if requested

% String1=C#; String2=Ab; String3=Eb; String4=Bb
if( active_pos == 1 )
    ratio_from_nut = 0.05; Kf_get = 5;
% String1=D ; String2=A ; String3=E ; String4=B
elseif( active_pos == 2 )
    ratio_from_nut = 0.11; Kf_get = 5;
% String1=Eb; String2=Bb; String3=F ; String4=C
elseif( active_pos == 3 )
    ratio_from_nut = 0.16; Kf_get = 5;
% String1=E ; String2=B ; String3=F#; String4=C#
elseif( active_pos == 4 )
    ratio_from_nut = 0.21; Kf_get = 5;
% String1=F ; String2=C ; String3=G ; String4=D
elseif( active_pos == 5 )
    ratio_from_nut = 0.25; Kf_get = 5;
% String1=F#; String2=C#; String3=Ab; String4=Eb
elseif( active_pos == 6 )
    ratio_from_nut = 0.29; Kf_get = 6;
% String1=G ; String2=D ; String3=A ; String4=E
elseif( active_pos == 7 )
    ratio_from_nut = 0.333; Kf_get = 5;
% String1=Ab; String2=Eb; String3=Bb; String4=F
elseif( active_pos == 8 )
    ratio_from_nut = 0.365; Kf_get = 5;
% String1=A ; String2=E ; String3=B ; String4=F#
elseif( active_pos == 9 )
    ratio_from_nut = 0.405; Kf_get = 5;
% String1=Bb; String2=F ; String3=C ; String4=G
elseif( active_pos == 10 )
    ratio_from_nut = 0.435; Kf_get = 5;
% String1=B ; String2=F#; String3=C#; String4=Ab
elseif( active_pos == 11 )
    ratio_from_nut = 0.47; Kf_get = 5;
% String1=C ; String2=G ; String3=D ; String4=A
elseif( active_pos == 12 )
    ratio_from_nut = 0.5; Kf_get = 5;
end

```

```

if( active_pos > 0 )
    String_finger();
end

String_bow();

%% Set simulation state (including audio data buffers)
total_samples = sim_time * fs;
notify_sample = round( fs / notify_div );
stopbow_sample = round( stopbow_pos * total_samples );
buf_start = 1;
buf_pos = 0;
buf_string = cell( 1, 4 );
buf_string{1} = zeros( 1, buf_len );
buf_string{2} = zeros( 1, buf_len );
buf_string{3} = zeros( 1, buf_len );
buf_string{4} = zeros( 1, buf_len );
buf_strings = zeros( 1, buf_len );
buf_body = zeros( 1, buf_len );

% Generate reference amplitude ratio, for volume-based convolution selection
ampl_ratio = 1 / peak_ampl;

% Store the history of the amplitude-based convolution vector transition
aA_sA_h = 0;
aA_sB_h = 1;
aB_sA_h = 0;
aB_sB_h = 1;

% Create clean temporary matfile
delete( 'output_temp.mat' );
f_output = matfile( 'output_temp.mat', 'Writable', true );

```

CelloSim_engine.m

```
% *****
% CELLO SIMULATION
%
% Name:    CelloSim_engine.m
% Date:    02/05/2016
% Author:  Dave Humphreys, c1322278
% Desc:    Cello simulation core engine code.
%          Requires setup:
%          CelloSim_h(), CelloSim_run_sim().
%          String code ported from "Artifastring",
%          (c) 2013 G. Percival.
%          Additional code added by D. Humphreys.
% *****

fprintf( '\nRUNNING SIMULATION...\n' );
tic
for t=1:total_samples
    %% Special actions (melody etc.)
    if( active_melody > 0 )
        CelloSim_melody();
    end

    %% Stop bowing? And add variations if not
    if( stop_bowing )
        if( t >= stopbow_sample )
            if( t == stopbow_sample )
                fprintf( '(Bowing ends.)\n' );
                s_str( o_str * stopbow_string + i_str_actions ) = e_str_RELEASE;
            end
        end
    end

    if( ~stop_bowing || t < stopbow_sample )
        % Non-mechanical variations in bow force, velocity, and finger pressure (if active)
        if( vary_bow_force )
            % bow force
            s_str( i_str_FbN_step ) = s_str( i_str_FbN_step ) + ...           % vary rate of change
                ( ( s_str( i_str_FbN_random ) * ( rand - 0.5 ) ) * s_str( i_str_FbN_delta ) );
        end
    end
end
```



```

    if( s_str( i_str_FbN_step ) > s_str( i_str_FbN_delta ) + ... % check !>max
        ( s_str( i_str_FbN_max_dev ) * s_str( i_str_FbN_delta ) ) )
        s_str( i_str_FbN_step ) = s_str( i_str_FbN_delta ) + ...
        ( s_str( i_str_FbN_max_dev ) * s_str( i_str_FbN_delta ) );
    elseif( s_str( i_str_FbN_step ) < s_str( i_str_FbN_delta ) - ... % check !<min
        ( s_str( i_str_FbN_max_dev ) * s_str( i_str_FbN_delta ) ) )
        s_str( i_str_FbN_step ) = s_str( i_str_FbN_delta ) - ...
        ( s_str( i_str_FbN_max_dev ) * s_str( i_str_FbN_delta ) );
    end
    s_str( i_str_FbN_state ) = s_str( i_str_FbN_state ) + s_str( i_str_FbN_step ); % increment state of
A_noise
    s_str( i_str_FbN_noise ) = sin( s_str( i_str_FbN_state ) ) * s_str( i_str_FbN_amt );
end
% bow velocity
if( vary_bow_velocity )
    s_str( i_str_vbN_step ) = s_str( i_str_vbN_step ) + ... % vary rate of change
    ( ( s_str( i_str_vbN_random ) * ( rand - 0.5 ) ) * s_str( i_str_vbN_delta ) );
    if( s_str( i_str_vbN_step ) > s_str( i_str_vbN_delta ) + ... % check !>max
        ( s_str( i_str_vbN_max_dev ) * s_str( i_str_vbN_delta ) ) )
        s_str( i_str_vbN_step ) = s_str( i_str_vbN_delta ) + ...
        ( s_str( i_str_vbN_max_dev ) * s_str( i_str_vbN_delta ) );
    elseif( s_str( i_str_vbN_step ) < s_str( i_str_vbN_delta ) - ... % check !<min
        ( s_str( i_str_vbN_max_dev ) * s_str( i_str_vbN_delta ) ) )
        s_str( i_str_vbN_step ) = s_str( i_str_vbN_delta ) - ...
        ( s_str( i_str_vbN_max_dev ) * s_str( i_str_vbN_delta ) );
    end
    s_str( i_str_vbN_state ) = s_str( i_str_vbN_state ) + s_str( i_str_vbN_step ); % increment state of
A_noise
    s_str( i_str_vbN_noise ) = sin( s_str( i_str_vbN_state ) ) * s_str( i_str_vbN_amt );
end
% finger position
if( vary_finger_pos )
    s_str( i_str_fpN_step ) = s_str( i_str_fpN_step ) + ... % vary rate of change
    ( ( s_str( i_str_fpN_random ) * ( rand - 0.5 ) ) * s_str( i_str_fpN_delta ) );
    if( s_str( i_str_fpN_step ) > s_str( i_str_fpN_delta ) + ... % check !>max
        ( s_str( i_str_fpN_max_dev ) * s_str( i_str_fpN_delta ) ) )
        s_str( i_str_fpN_step ) = s_str( i_str_fpN_delta ) + ...
        ( s_str( i_str_fpN_max_dev ) * s_str( i_str_fpN_delta ) );
    elseif( s_str( i_str_fpN_step ) < s_str( i_str_fpN_delta ) - ... % check !<min
        ( s_str( i_str_fpN_max_dev ) * s_str( i_str_fpN_delta ) ) )
        s_str( i_str_fpN_step ) = s_str( i_str_fpN_delta ) - ...
        ( s_str( i_str_fpN_max_dev ) * s_str( i_str_fpN_delta ) );
    end
end

```

```

A_noise    s_str( i_str_fpN_state ) = s_str( i_str_fpN_state ) + s_str( i_str_fpN_step );           % increment state of
    s_str( i_str_fpN_noise ) = sin( s_str( i_str_fpN_state ) ) * s_str( i_str_fpN_amt );
end
% simple vibrato
if( vibrato_active )
    s_str( i_str_vib_state ) = s_str( i_str_vib_state ) + s_str( i_str_vib_step );
    s_str( i_str_vib_value ) = ( real( log( sin( s_str( i_str_vib_state ) ) ) ) / 2 + 1 ) * ...
        s_str( i_str_vib_amt );
end
end

%% Process buffer index
buf_pos = buf_pos + 1;
if( buf_pos == buf_len + 1 )
    % disk write
    f_output.strings( 1, buf_start : t - 1 ) = buf_strings( 1, 1 : buf_len );
    f_output.string1( 1, buf_start : t - 1 ) = buf_string{1}( 1, 1 : buf_len );
    f_output.string2( 1, buf_start : t - 1 ) = buf_string{2}( 1, 1 : buf_len );
    f_output.string3( 1, buf_start : t - 1 ) = buf_string{3}( 1, 1 : buf_len );
    f_output.string4( 1, buf_start : t - 1 ) = buf_string{4}( 1, 1 : buf_len );
    f_output.body( 1, buf_start : t - 1 ) = buf_body( 1, 1 : buf_len );
    buf_start = t;
    buf_pos = 1;
end

%% Generate output for all strings
buf_strings( buf_pos ) = 0;
for string_num=0:3
    buf_string{ string_num + 1 }( buf_pos ) = 0;

    s_str( i_str_curStr ) = string_num;

    % * From String_generate_output() * %
    % calculate offsets
    curSOfst = o_str * s_str( i_str_curStr );
    curScOfst = o_cnst * s_str( i_str_curStr );
    %

    y_temp = 0;

```

```

% apply bow and finger variations
% bow force
if( vary_bow_force )
    s_str( curSOfst + i_str_Fb ) = s_str( curSOfst + i_str_Fb_req ) + ...
        ( s_str( i_str_FbN_noise ) * s_str( curSOfst + i_str_Fb_req ) );
else
    s_str( curSOfst + i_str_Fb ) = s_str( curSOfst + i_str_Fb_req );
end
% bow velocity
if( vary_bow_velocity )
    s_str( curSOfst + i_str_vbN_diff ) = s_str( i_str_vbN_noise ) * s_str( curSOfst + i_str_vb_req );
    s_str( curSOfst + i_str_vb ) = s_str( curSOfst + i_str_vb ) + ...
        ( s_str( curSOfst + i_str_vbN_diff ) - s_str( curSOfst + i_str_vbN_diff_prev ) );
    s_str( curSOfst + i_str_vbN_diff_prev ) = s_str( curSOfst + i_str_vbN_diff );
else
    s_str( curSOfst + i_str_vb ) = s_str( curSOfst + i_str_vb_req );
end
% finger position
if( s_str( curSOfst + i_str_finger_position_req ) > 0 )
    if( vary_finger_pos )
        s_str( curSOfst + i_str_finger_position ) = s_str( curSOfst + i_str_finger_position_req ) + ...
            ( s_str( i_str_fpN_noise ) * s_cnst( curScOfst + i_cnst_L ) );
        s_str( curSOfst + i_str_recache ) = 1;
    else
        s_str( curSOfst + i_str_finger_position ) = s_str( curSOfst + i_str_finger_position_req );
    end
    % apply vibrato, only if a finger is active
    if( vibrato_active )
        s_str( curSOfst + i_str_finger_position ) = s_str( curSOfst + i_str_finger_position ) + ...
            ( s_str( i_str_vib_value ) * s_cnst( curScOfst + i_cnst_L ) );
    end
end

for pass=1:s_cnst( curScOfst + i_cnst_mlt )    % repeat (and average output) if sample rate multiplied
    y = 0;    % output nothing if actions == OFF

    if( s_str( curSOfst + i_str_recache ) == 1 )
        % * From String_cache_pa_c * %
        s_str( curSOfst + i_str_x1 ) = s_str( curSOfst + i_str_finger_position );
        s_str( curSOfst + i_str_K1 ) = s_str( curSOfst + i_str_Kf );
        s_str( curSOfst + i_str_R1 ) = s_str( curSOfst + i_str_R_FINGER ) * ...
            ( s_str( curSOfst + i_str_Kf ) / s_str( curSOfst + i_str_K_FINGER ) );
    end
end

```

```

if( s_str( curS0fst + i_str_actions ) == e_str_BOW )
    s_str( curS0fst + i_str_x0 ) = s_str( curS0fst + i_str_bow_pluck_position );
    s_str( curS0fst + i_str_x2 ) = 0;
end

if( s_str( curS0fst + i_str_actions ) == e_str_RELEASE )
    s_str( curS0fst + i_str_x2 ) = 0;
    s_str( curS0fst + i_str_K2 ) = 0;
    s_str( curS0fst + i_str_R2 ) = 0;

    if( s_str( curS0fst + i_str_finger_position ) == 0 )
        s_str( curS0fst + i_str_x0 ) = 0;
        s_str( curS0fst + i_str_K0 ) = 0;
        s_str( curS0fst + i_str_R0 ) = 0;
    else
        s_str( curS0fst + i_str_K0 ) = s_str( curS0fst + i_str_K1 );
        s_str( curS0fst + i_str_R0 ) = s_str( curS0fst + i_str_R1 );
        if( s_str( curS0fst + i_str_finger_position ) < ( s_cnst( curSc0fst + i_cnst_L ) - s_str( curS0fst +
i_str_FINGER_WIDTH ) ) )
            s_str( curS0fst + i_str_x0 ) = s_str( curS0fst + i_str_finger_position ) + s_str( curS0fst +
i_str_FINGER_WIDTH );
        else
            remaining_string = s_cnst( curSc0fst + i_cnst_L ) - s_str( curS0fst + i_str_finger_position );
            s_str( curS0fst + i_str_x0 ) = s_str( curS0fst + i_str_finger_position ) + 0.5 * remaining_string;
        end
    end
end

if( s_str( curS0fst + i_str_actions ) == e_str_PLUCK )
    s_str( curS0fst + i_str_x0 ) = s_str( curS0fst + i_str_bow_pluck_position );
    s_str( curS0fst + i_str_K0 ) = s_str( curS0fst + i_str_K_PLUCK );
    s_str( curS0fst + i_str_R0 ) = s_str( curS0fst + i_str_R_PLUCK );
    s_str( curS0fst + i_str_x2 ) = s_str( curS0fst + i_str_bow_pluck_position ) + s_str( curS0fst +
i_str_PLUCK_WIDTH );
    s_str( curS0fst + i_str_K2 ) = s_str( curS0fst + i_str_K_PLUCK );
    s_str( curS0fst + i_str_R2 ) = s_str( curS0fst + i_str_R_PLUCK );
end

% Below from cache_pa_c()
s_str( curS0fst + i_str_phix0 : curS0fst + i_str_phix0 + s_cnst( curSc0fst + i_cnst_N ) - 1 ) = ...
    s_str( curS0fst + i_str_sqrt_two_div_L ) * sin( s_str( curS0fst + i_str_x0 ) ...
    * s_str( curS0fst + i_str_inside_phi : curS0fst + i_str_inside_phi + s_cnst( curSc0fst + i_cnst_N ) - 1 ) );
s_str( curS0fst + i_str_phix1 : curS0fst + i_str_phix1 + s_cnst( curSc0fst + i_cnst_N ) - 1 ) = ...

```

```

    s_str( curSOfst + i_str_sqrt_two_div_L ) * sin( s_str( curSOfst + i_str_x1 ) ...
    * s_str( curSOfst + i_str_inside_phi : curSOfst + i_str_inside_phi + s_cnst( curScOfst + i_cnst_N ) - 1 ) );
A00 = sum( s_str( curSOfst + i_str_X3 : curSOfst + i_str_X3 + s_cnst( curScOfst + i_cnst_N ) - 1 ) .* ...
    s_str( curSOfst + i_str_phix0 : curSOfst + i_str_phix0 + s_cnst( curScOfst + i_cnst_N ) - 1 ) .* ...
    s_str( curSOfst + i_str_phix0 : curSOfst + i_str_phix0 + s_cnst( curScOfst + i_cnst_N ) - 1 ) );
A01 = sum( s_str( curSOfst + i_str_X3 : curSOfst + i_str_X3 + s_cnst( curScOfst + i_cnst_N ) - 1 ) .* ...
    s_str( curSOfst + i_str_phix0 : curSOfst + i_str_phix0 + s_cnst( curScOfst + i_cnst_N ) - 1 ) .* ...
    s_str( curSOfst + i_str_phix1 : curSOfst + i_str_phix1 + s_cnst( curScOfst + i_cnst_N ) - 1 ) );
A11 = sum( s_str( curSOfst + i_str_X3 : curSOfst + i_str_X3 + s_cnst( curScOfst + i_cnst_N ) - 1 ) .* ...
    s_str( curSOfst + i_str_phix1 : curSOfst + i_str_phix1 + s_cnst( curScOfst + i_cnst_N ) - 1 ) .* ...
    s_str( curSOfst + i_str_phix1 : curSOfst + i_str_phix1 + s_cnst( curScOfst + i_cnst_N ) - 1 ) );

B00 = sum( s_str( curSOfst + i_str_Y3 : curSOfst + i_str_Y3 + s_cnst( curScOfst + i_cnst_N ) - 1 ) .* ...
    s_str( curSOfst + i_str_phix0 : curSOfst + i_str_phix0 + s_cnst( curScOfst + i_cnst_N ) - 1 ) .* ...
    s_str( curSOfst + i_str_phix0 : curSOfst + i_str_phix0 + s_cnst( curScOfst + i_cnst_N ) - 1 ) );
B01 = sum( s_str( curSOfst + i_str_Y3 : curSOfst + i_str_Y3 + s_cnst( curScOfst + i_cnst_N ) - 1 ) .* ...
    s_str( curSOfst + i_str_phix0 : curSOfst + i_str_phix0 + s_cnst( curScOfst + i_cnst_N ) - 1 ) .* ...
    s_str( curSOfst + i_str_phix1 : curSOfst + i_str_phix1 + s_cnst( curScOfst + i_cnst_N ) - 1 ) );
B11 = sum( s_str( curSOfst + i_str_Y3 : curSOfst + i_str_Y3 + s_cnst( curScOfst + i_cnst_N ) - 1 ) .* ...
    s_str( curSOfst + i_str_phix1 : curSOfst + i_str_phix1 + s_cnst( curScOfst + i_cnst_N ) - 1 ) .* ...
    s_str( curSOfst + i_str_phix1 : curSOfst + i_str_phix1 + s_cnst( curScOfst + i_cnst_N ) - 1 ) );

if( s_str( curSOfst + i_str_actions ) == e_str_BOW )
    % bow coefficients
    L1 = 1 / ( ( B00*B11 - B01*B01 ) * s_str( curSOfst + i_str_R1 ) + ...
        ( A11*B00 - A01*B01 ) * s_str( curSOfst + i_str_K1 ) + B00 );
    s_str( curSOfst + i_str_D1 ) = ( B11 * s_str( curSOfst + i_str_R1 ) + ...
        A11 * s_str( curSOfst + i_str_K1 ) + 1 ) * L1;
    s_str( curSOfst + i_str_D4 ) = 0.5 / s_str( curSOfst + i_str_D1 );
    s_str( curSOfst + i_str_D2 ) = B01 * s_str( curSOfst + i_str_K1 ) * L1;
    s_str( curSOfst + i_str_D3 ) = B01 * s_str( curSOfst + i_str_R1 ) * L1;
    % finger-during-bowing coefficients
    L2 = -1 / ( B11 * s_str( curSOfst + i_str_R1 ) + A11 * s_str( curSOfst + i_str_K1 ) + 1 );
    s_str( curSOfst + i_str_D5 ) = ( B01 * s_str( curSOfst + i_str_R1 ) + ...
        A01 * s_str( curSOfst + i_str_K1 ) ) * L2;
    s_str( curSOfst + i_str_D6 ) = s_str( curSOfst + i_str_R1 ) * L2;
    s_str( curSOfst + i_str_D7 ) = s_str( curSOfst + i_str_K1 ) * L2;
end

```

```

if( s_str( curSOfst + i_str_actions ) == e_str_PLUCK )
    s_str( curSOfst + i_str_phix2 : curSOfst + i_str_phix2 + s_cnst( curScOfst + i_cnst_N ) - 1 ) = ...
        s_str( curSOfst + i_str_sqrt_two_div_L ) .* sin( s_str( curSOfst + i_str_x2 ) ...
            * s_str( curSOfst + i_str_inside_phi : curSOfst + i_str_inside_phi + s_cnst( curScOfst + i_cnst_N ) - 1 ) );
    A02 = sum( s_str( curSOfst + i_str_X3 : curSOfst + i_str_X3 + s_cnst( curScOfst + i_cnst_N ) - 1 ) .* ...
        s_str( curSOfst + i_str_phix0 : curSOfst + i_str_phix0 + s_cnst( curScOfst + i_cnst_N ) - 1 ) .* ...
        s_str( curSOfst + i_str_phix2 : curSOfst + i_str_phix2 + s_cnst( curScOfst + i_cnst_N ) - 1 ) );
    A12 = sum( s_str( curSOfst + i_str_X3 : curSOfst + i_str_X3 + s_cnst( curScOfst + i_cnst_N ) - 1 ) .* ...
        s_str( curSOfst + i_str_phix1 : curSOfst + i_str_phix1 + s_cnst( curScOfst + i_cnst_N ) - 1 ) .* ...
        s_str( curSOfst + i_str_phix2 : curSOfst + i_str_phix2 + s_cnst( curScOfst + i_cnst_N ) - 1 ) );
    A22 = sum( s_str( curSOfst + i_str_X3 : curSOfst + i_str_X3 + s_cnst( curScOfst + i_cnst_N ) - 1 ) .* ...
        s_str( curSOfst + i_str_phix2 : curSOfst + i_str_phix2 + s_cnst( curScOfst + i_cnst_N ) - 1 ) .* ...
        s_str( curSOfst + i_str_phix2 : curSOfst + i_str_phix2 + s_cnst( curScOfst + i_cnst_N ) - 1 ) );
    B02 = sum( s_str( curSOfst + i_str_Y3 : curSOfst + i_str_Y3 + s_cnst( curScOfst + i_cnst_N ) - 1 ) .* ...
        s_str( curSOfst + i_str_phix0 : curSOfst + i_str_phix0 + s_cnst( curScOfst + i_cnst_N ) - 1 ) .* ...
        s_str( curSOfst + i_str_phix2 : curSOfst + i_str_phix2 + s_cnst( curScOfst + i_cnst_N ) - 1 ) );
    B12 = sum( s_str( curSOfst + i_str_Y3 : curSOfst + i_str_Y3 + s_cnst( curScOfst + i_cnst_N ) - 1 ) .* ...
        s_str( curSOfst + i_str_phix1 : curSOfst + i_str_phix1 + s_cnst( curScOfst + i_cnst_N ) - 1 ) .* ...
        s_str( curSOfst + i_str_phix2 : curSOfst + i_str_phix2 + s_cnst( curScOfst + i_cnst_N ) - 1 ) );
    B22 = sum( s_str( curSOfst + i_str_Y3 : curSOfst + i_str_Y3 + s_cnst( curScOfst + i_cnst_N ) - 1 ) .* ...
        s_str( curSOfst + i_str_phix2 : curSOfst + i_str_phix2 + s_cnst( curScOfst + i_cnst_N ) - 1 ) .* ...
        s_str( curSOfst + i_str_phix2 : curSOfst + i_str_phix2 + s_cnst( curScOfst + i_cnst_N ) - 1 ) );

    matrix_A = [ B00 * s_str( curSOfst + i_str_R0 ) + A00 * s_str( curSOfst + i_str_K0 ) + 1, ...
        B01 * s_str( curSOfst + i_str_R0 ) + A01 * s_str( curSOfst + i_str_K0 ), ...
        B02 * s_str( curSOfst + i_str_R0 ) + A02 * s_str( curSOfst + i_str_K0 ); ...

        B01 * s_str( curSOfst + i_str_R1 ) + A01 * s_str( curSOfst + i_str_K1 ), ...
        B11 * s_str( curSOfst + i_str_R1 ) + A11 * s_str( curSOfst + i_str_K1 ) + 1, ...
        B12 * s_str( curSOfst + i_str_R1 ) + A12 * s_str( curSOfst + i_str_K1 ); ...

        B02 * s_str( curSOfst + i_str_R2 ) + A02 * s_str( curSOfst + i_str_K2 ), ...
        B12 * s_str( curSOfst + i_str_R2 ) + A12 * s_str( curSOfst + i_str_K2 ), ...
        B22 * s_str( curSOfst + i_str_R2 ) + A22 * s_str( curSOfst + i_str_K2 ) + 1; ];
    inv_A = inv( matrix_A );
    s_str( curSOfst + i_str_inv_A : curSOfst + i_str_inv_A + n_str_inv_A ) = [ inv_A(1,:) inv_A(2,:) inv_A(3,:) ];
end

if( s_str( curSOfst + i_str_actions ) == e_str_RELEASE )
    M00 = A00 * s_str( curSOfst + i_str_K1 ) + B00 * s_str( curSOfst + i_str_R1 ) + 1;
    M01 = A01 * s_str( curSOfst + i_str_K1 ) + B01 * s_str( curSOfst + i_str_R1 );
    M11 = A11 * s_str( curSOfst + i_str_K1 ) + B11 * s_str( curSOfst + i_str_R1 ) + 1;
    L3 = -1 / ( M00 * M11 - M01 * M01 );

```

```

s_str( curSOfst + i_str_D8 ) = -1 / M00;
s_str( curSOfst + i_str_D9 ) = M01 * s_str( curSOfst + i_str_D8 );
s_str( curSOfst + i_str_D10 ) = -M01 * L3;
s_str( curSOfst + i_str_D11 ) = M00 * L3;

```

fill_buffer)"

```

matrix_A = [ B00 * s_str( curSOfst + i_str_R0 ) + A00 * s_str( curSOfst + i_str_K0 ) + 1, ...
             B01 * s_str( curSOfst + i_str_R0 ) + A01 * s_str( curSOfst + i_str_K0 ), ...
             0; ...

             B01 * s_str( curSOfst + i_str_R1 ) + A01 * s_str( curSOfst + i_str_K1 ), ...
             B11 * s_str( curSOfst + i_str_R1 ) + A11 * s_str( curSOfst + i_str_K1 ) + 1, ...
             0; ...

             0, ...
             0, ...
             1; ];

```

```

inv_A = inv( matrix_A );
s_str( curSOfst + i_str_inv_A : curSOfst + i_str_inv_A + n_str_inv_A ) = [ inv_A(1,:) inv_A(2,:) inv_A(3,:) ];

```

end

% now that the recache is done, reset the flag

```
s_str( curSOfst + i_str_recache ) = 0;
```

end

% call appropriate output method: tick_bow(), tick_free() etc.

```
if( s_str( curSOfst + i_str_actions ) == e_str_BOW || ...
```

```
    s_str( curSOfst + i_str_actions ) == e_str_BOW_ACCEL )
```

*% * From String_tick_bow * %*

```

s_str( curSOfst + i_str_ah : curSOfst + i_str_ah + s_cnst( curScOfst + i_cnst_N ) - 1 ) = ...
    s_str( curSOfst + i_str_X1 : curSOfst + i_str_X1 + s_cnst( curScOfst + i_cnst_N ) - 1 ) .* ...
    s_str( curSOfst + i_str_a : curSOfst + i_str_a + s_cnst( curScOfst + i_cnst_N ) - 1 ) + ...
    s_str( curSOfst + i_str_X2 : curSOfst + i_str_X2 + s_cnst( curScOfst + i_cnst_N ) - 1 ) .* ...
    s_str( curSOfst + i_str_ad : curSOfst + i_str_ad + s_cnst( curScOfst + i_cnst_N ) - 1 );
s_str( curSOfst + i_str_adh : curSOfst + i_str_adh + s_cnst( curScOfst + i_cnst_N ) - 1 ) = ...
    s_str( curSOfst + i_str_Y1 : curSOfst + i_str_Y1 + s_cnst( curScOfst + i_cnst_N ) - 1 ) .* ...
    s_str( curSOfst + i_str_a : curSOfst + i_str_a + s_cnst( curScOfst + i_cnst_N ) - 1 ) + ...
    s_str( curSOfst + i_str_Y2 : curSOfst + i_str_Y2 + s_cnst( curScOfst + i_cnst_N ) - 1 ) .* ...
    s_str( curSOfst + i_str_ad : curSOfst + i_str_ad + s_cnst( curScOfst + i_cnst_N ) - 1 );

```

```

s_str( curS0fst + i_str_y1h ) = 0;
s_str( curS0fst + i_str_v1h ) = 0;
if( s_str( curS0fst + i_str_x1 ) > 0 )
    s_str( curS0fst + i_str_y1h ) = ...
        sum( s_str( curS0fst + i_str_phix1 : curS0fst + i_str_phix1 + s_cnst( curSc0fst + i_cnst_N ) - 1 ) .* ...
            s_str( curS0fst + i_str_ah : curS0fst + i_str_ah + s_cnst( curSc0fst + i_cnst_N ) - 1 ) );
    s_str( curS0fst + i_str_v1h ) = ...
        sum( s_str( curS0fst + i_str_phix1 : curS0fst + i_str_phix1 + s_cnst( curSc0fst + i_cnst_N ) - 1 ) .* ...
            s_str( curS0fst + i_str_adh : curS0fst + i_str_adh + s_cnst( curSc0fst + i_cnst_N ) - 1 ) );
end

% calculate excitation force
s_str( curS0fst + i_str_v0h ) = ...
    s_str( curS0fst + i_str_phix0 : curS0fst + i_str_phix0 + s_cnst( curSc0fst + i_cnst_N ) - 1 ) * ...
    s_str( curS0fst + i_str_adh : curS0fst + i_str_adh + s_cnst( curSc0fst + i_cnst_N ) - 1 )';

% DJH: Calculate new 'random' values (sine based) for this iteration
s_str( curS0fst + i_str_A_step ) = s_str( curS0fst + i_str_A_step ) + ... % vary rate of change
    ( ( s_str( curS0fst + i_str_A_random ) * ( rand - 0.5 ) ) * s_str( curS0fst + i_str_A_delta ) );

if( s_str( curS0fst + i_str_A_step ) > s_str( curS0fst + i_str_A_delta ) + ... % check !>max
    ( s_str( curS0fst + i_str_A_max_dev ) * s_str( curS0fst + i_str_A_delta ) ) )
    s_str( curS0fst + i_str_A_step ) = s_str( curS0fst + i_str_A_delta ) + ...
        ( s_str( curS0fst + i_str_A_max_dev ) * s_str( curS0fst + i_str_A_delta ) );
elseif( s_str( curS0fst + i_str_A_step ) < s_str( curS0fst + i_str_A_delta ) - ... % check !<min
    ( s_str( curS0fst + i_str_A_max_dev ) * s_str( curS0fst + i_str_A_delta ) ) )
    s_str( curS0fst + i_str_A_step ) = s_str( curS0fst + i_str_A_delta ) - ...
        ( s_str( curS0fst + i_str_A_max_dev ) * s_str( curS0fst + i_str_A_delta ) );
end

s_str( curS0fst + i_str_A_state ) = s_str( curS0fst + i_str_A_state ) + s_str( curS0fst + i_str_A_step ); %
increment state of A_noise
s_str( curS0fst + i_str_No ) = s_str( curS0fst + i_str_A_noise ) * sin( s_str( curS0fst + i_str_A_state ) ); %
set variation amount
%
```

```

% From compute_bow()
max_stable = s_cnst( curSc0fst + i_cnst_mu_s ) * s_str( curS0fst + i_str_Fb );

% disable bow noise, if requested
if( ~bow_noise_active )
    s_str( curS0fst + i_str_No ) = 0;
end

```



```

% * From String_compute_bowslip_pos * %
result_pos = 0;
ve0 = ( 1 - ( s_str( curS0fst + i_str_No ) * s_str( curS0fst + i_str_vb ) ) ) * s_cnst( curSc0fst + i_cnst_v0 );
% 'v hat 0' or 'v estimate 0'

% handle positive side of friction curve
c1 = s_str( curS0fst + i_str_D1 ) * ( s_str( curS0fst + i_str_vb ) - s_str( curS0fst + i_str_v0h ) + ve0 ) + ...
    s_str( curS0fst + i_str_D2 ) * s_str( curS0fst + i_str_y1h ) + ...
    s_str( curS0fst + i_str_D3 ) * s_str( curS0fst + i_str_v1h ) + ...
    s_str( curS0fst + i_str_Fb ) * s_cnst( curSc0fst + i_cnst_mu_d );
c0 = ve0 * ( s_str( curS0fst + i_str_D1 ) * ( s_str( curS0fst + i_str_vb ) - s_str( curS0fst + i_str_v0h ) ) + ...
    s_str( curS0fst + i_str_D2 ) * s_str( curS0fst + i_str_y1h ) + ...
    s_str( curS0fst + i_str_D3 ) * s_str( curS0fst + i_str_v1h ) + ...
    s_str( curS0fst + i_str_Fb ) * s_cnst( curSc0fst + i_cnst_mu_s ) );

% 'a real solution exists'
Delta = c1 * c1 - 4 * c0 * s_str( curS0fst + i_str_D1 );
if( Delta >= 0 )
    result_pos = s_str( curS0fst + i_str_D4 ) * ( -c1 - sqrt( Delta ) );
end

% * From String_compute_bowslip_neg * %
result_neg = 0;
ve0 = ( 1 - ( s_str( curS0fst + i_str_No ) * s_str( curS0fst + i_str_vb ) ) ) * s_cnst( curSc0fst + i_cnst_v0 );
% 'v hat 0' or 'v estimate 0'

% handle negative side of friction curve
c1 = -s_str( curS0fst + i_str_D1 ) * ( s_str( curS0fst + i_str_vb ) - s_str( curS0fst + i_str_v0h ) - ve0 ) - ...
    s_str( curS0fst + i_str_D2 ) * s_str( curS0fst + i_str_y1h ) - ...
    s_str( curS0fst + i_str_D3 ) * s_str( curS0fst + i_str_v1h ) + ...
    s_str( curS0fst + i_str_Fb ) * s_cnst( curSc0fst + i_cnst_mu_d );
c0 = ve0 * ( s_str( curS0fst + i_str_D1 ) * ( s_str( curS0fst + i_str_vb ) - s_str( curS0fst + i_str_v0h ) ) + ...
    s_str( curS0fst + i_str_D2 ) * s_str( curS0fst + i_str_y1h ) + ...
    s_str( curS0fst + i_str_D3 ) * s_str( curS0fst + i_str_v1h ) - ...
    s_str( curS0fst + i_str_Fb ) * s_cnst( curSc0fst + i_cnst_mu_s ) );

% 'a real solution exists'
Delta = c1 * c1 + 4 * c0 * s_str( curS0fst + i_str_D1 );
if( Delta >= 0 )
    result_neg = s_str( curS0fst + i_str_D4 ) * ( c1 - sqrt( Delta ) );
end

```

```

% Compute F0 for stable state
F0_stable = s_str( curS0fst + i_str_D1 ) * ...
    ( s_str( curS0fst + i_str_vb ) - s_str( curS0fst + i_str_v0h ) ) + ...
    s_str( curS0fst + i_str_D2 ) * s_str( curS0fst + i_str_y1h ) + ...
    s_str( curS0fst + i_str_D3 ) * s_str( curS0fst + i_str_v1h );

% Stable state
if( abs( F0_stable - max_stable ) <= eps )
    s_str( curS0fst + i_str_F0 ) = F0_stable;
    if( s_str( curS0fst + i_str_slipstate ) ~= 0 )
        s_str( curS0fst + i_str_F0 ) = 0;
    end
    s_str( curS0fst + i_str_slipstate ) = 0;
% Positive slip
elseif( result_pos > 0 )
    s_str( curS0fst + i_str_slipstate ) = 1;
    s_str( curS0fst + i_str_F0 ) = s_str( curS0fst + i_str_D1 ) * ...
        ( s_str( curS0fst + i_str_vb ) + result_pos - s_str( curS0fst + i_str_v0h ) ) + ...
        s_str( curS0fst + i_str_D2 ) * s_str( curS0fst + i_str_y1h ) + ...
        s_str( curS0fst + i_str_D3 ) * s_str( curS0fst + i_str_v1h );
% Negative slip
else % result_neg must be <0!
    s_str( curS0fst + i_str_slipstate ) = -1;
    s_str( curS0fst + i_str_F0 ) = s_str( curS0fst + i_str_D1 ) * ...
        ( s_str( curS0fst + i_str_vb ) + result_neg - s_str( curS0fst + i_str_v0h ) ) + ...
        s_str( curS0fst + i_str_D2 ) * s_str( curS0fst + i_str_y1h ) + ...
        s_str( curS0fst + i_str_D3 ) * s_str( curS0fst + i_str_v1h );
end

F1 = s_str( curS0fst + i_str_D5 ) * s_str( curS0fst + i_str_F0 ) + ...
    s_str( curS0fst + i_str_D6 ) * s_str( curS0fst + i_str_v1h ) + ...
    s_str( curS0fst + i_str_D7 ) * s_str( curS0fst + i_str_y1h );

% apply forces
fn = s_str( curS0fst + i_str_phix0 : curS0fst + i_str_phix0 + s_cnst( curSc0fst + i_cnst_N ) - 1 ) * ...
    s_str( curS0fst + i_str_F0 ) + ...
    s_str( curS0fst + i_str_phix1 : curS0fst + i_str_phix1 + s_cnst( curSc0fst + i_cnst_N ) - 1 ) * F1;
s_str( curS0fst + i_str_a : curS0fst + i_str_a + s_cnst( curSc0fst + i_cnst_N ) - 1 ) = ...
    s_str( curS0fst + i_str_ah : curS0fst + i_str_ah + s_cnst( curSc0fst + i_cnst_N ) - 1 ) + ...
    s_str( curS0fst + i_str_X3 : curS0fst + i_str_X3 + s_cnst( curSc0fst + i_cnst_N ) - 1 ) .* fn;
s_str( curS0fst + i_str_ad : curS0fst + i_str_ad + s_cnst( curSc0fst + i_cnst_N ) - 1 ) = ...
    s_str( curS0fst + i_str_adh : curS0fst + i_str_adh + s_cnst( curSc0fst + i_cnst_N ) - 1 ) + ...
    s_str( curS0fst + i_str_Y3 : curS0fst + i_str_Y3 + s_cnst( curSc0fst + i_cnst_N ) - 1 ) .* fn;

```

```

% s_str( curSOfst + i_str_tick_output_force ) = s_str( curSOfst + i_str_F0 );    % can use directly!

% From compute_bridge_force()
% DJH - apply LPF to derivative only (so next sample will be affected)
s_str( curSOfst + i_str_ad : curSOfst + i_str_ad + s_cnst( curScOfst + i_cnst_N ) - 1 ) = ...
    s_str( curSOfst + i_str_ad : curSOfst + i_str_ad + s_cnst( curScOfst + i_cnst_N ) - 1 ) .* ...
    s_cnst( curScOfst + i_cnst_lpf : curScOfst + i_cnst_lpf + s_cnst( curScOfst + i_cnst_N ) - 1 );
y = s_str( curSOfst + i_str_a : curSOfst + i_str_a + s_cnst( curScOfst + i_cnst_N ) - 1 ) * ...
    s_str( curSOfst + i_str_G : curSOfst + i_str_G + s_cnst( curScOfst + i_cnst_N ) - 1 )';

if( s_str( curSOfst + i_str_actions ) == e_str_BOW_ACCEL )
    % * From String_tick_bow_accel (minus String_tick_bow() call)
    % from update_bow_accel()
    % accelerate bow
    s_str( curSOfst + i_str_vb ) = s_str( curSOfst + i_str_vb ) + s_str( curSOfst + i_str_va );
    % DJH: corrected the following code (same assignments ran in all cases!)
    if( s_str( curSOfst + i_str_vb ) > s_str( curSOfst + i_str_vb_target ) || ...
        s_str( curSOfst + i_str_vb ) < s_str( curSOfst + i_str_vb_target ) )
        % DJH: also apply a variation
        s_str( curSOfst + i_str_vb ) = s_str( curSOfst + i_str_vb_target ) + ...
            ( s_str( i_str_vbN_noise ) * s_str( curSOfst + i_str_vb_target ) );
        s_str( curSOfst + i_str_actions ) = e_str_BOW;
    end
end

elseif( s_str( curSOfst + i_str_actions ) == e_str_PLUCK )
    % * From String_tick_pluck * %
    % from tick_pluck()
    % tick_pluck() says: 'hands-free' modes
    ah = s_str( curSOfst + i_str_X1 : curSOfst + i_str_X1 + s_cnst( curScOfst + i_cnst_N ) - 1 ) .* ...
        s_str( curSOfst + i_str_a : curSOfst + i_str_a + s_cnst( curScOfst + i_cnst_N ) - 1 ) + ...
        s_str( curSOfst + i_str_X2 : curSOfst + i_str_X2 + s_cnst( curScOfst + i_cnst_N ) - 1 ) .* ...
        s_str( curSOfst + i_str_ad : curSOfst + i_str_ad + s_cnst( curScOfst + i_cnst_N ) - 1 );
    adh = s_str( curSOfst + i_str_Y1 : curSOfst + i_str_Y1 + s_cnst( curScOfst + i_cnst_N ) - 1 ) .* ...
        s_str( curSOfst + i_str_a : curSOfst + i_str_a + s_cnst( curScOfst + i_cnst_N ) - 1 ) + ...
        s_str( curSOfst + i_str_Y2 : curSOfst + i_str_Y2 + s_cnst( curScOfst + i_cnst_N ) - 1 ) .* ...
        s_str( curSOfst + i_str_ad : curSOfst + i_str_ad + s_cnst( curScOfst + i_cnst_N ) - 1 );

    % 'hands-free' string displacement under force locations
    s_str( curSOfst + i_str_y0h ) = ...
        s_str( curSOfst + i_str_phix0 : curSOfst + i_str_phix0 + s_cnst( curScOfst + i_cnst_N ) - 1 ) * ah';

```

```

s_str( curS0fst + i_str_y1h ) = ...
    s_str( curS0fst + i_str_phix1 : curS0fst + i_str_phix1 + s_cnst( curSc0fst + i_cnst_N ) - 1 ) * ah';
y2h = s_str( curS0fst + i_str_phix2 : curS0fst + i_str_phix2 + s_cnst( curSc0fst + i_cnst_N ) - 1 ) * ah';
s_str( curS0fst + i_str_v0h ) = ...
    s_str( curS0fst + i_str_phix0 : curS0fst + i_str_phix0 + s_cnst( curSc0fst + i_cnst_N ) - 1 ) * adh';
s_str( curS0fst + i_str_v1h ) = ...
    s_str( curS0fst + i_str_phix1 : curS0fst + i_str_phix1 + s_cnst( curSc0fst + i_cnst_N ) - 1 ) * adh';
v2h = s_str( curS0fst + i_str_phix2 : curS0fst + i_str_phix2 + s_cnst( curSc0fst + i_cnst_N ) - 1 ) * adh';

matrix_b = [ -s_str( curS0fst + i_str_v0h ) * s_str( curS0fst + i_str_R0 ) - ...
              ( s_str( curS0fst + i_str_y0h ) - s_str( curS0fst + i_str_y_pluck ) ) * ...
              s_str( curS0fst + i_str_K0 ); ...
              -s_str( curS0fst + i_str_v1h ) * s_str( curS0fst + i_str_R1 ) - ...
              s_str( curS0fst + i_str_y1h ) * s_str( curS0fst + i_str_K1 ); ...
              -v2h * s_str( curS0fst + i_str_R2 ) - ( y2h - s_str( curS0fst + i_str_y_pluck ) ) ];

% "forces at those locations"
% mFs = inv_A * matrix_b; % original Eigen command
mFs_a = s_str( curS0fst + i_str_inv_A : curS0fst + i_str_inv_A + 2 ) * matrix_b;
mFs_b = s_str( curS0fst + i_str_inv_A + 3 : curS0fst + i_str_inv_A + 5 ) * matrix_b;
mFs_c = s_str( curS0fst + i_str_inv_A + 6 : curS0fst + i_str_inv_A + 8 ) * matrix_b;

% apply forces
fn = s_str( curS0fst + i_str_phix0 : curS0fst + i_str_phix0 + s_cnst( curSc0fst + i_cnst_N ) - 1 ) * mFs_a + ...
    s_str( curS0fst + i_str_phix1 : curS0fst + i_str_phix1 + s_cnst( curSc0fst + i_cnst_N ) - 1 ) * mFs_b + ...
    s_str( curS0fst + i_str_phix2 : curS0fst + i_str_phix2 + s_cnst( curSc0fst + i_cnst_N ) - 1 ) * mFs_c;
s_str( curS0fst + i_str_a : curS0fst + i_str_a + s_cnst( curSc0fst + i_cnst_N ) - 1 ) = ...
    ah + s_str( curS0fst + i_str_X3 ) .* fn;
s_str( curS0fst + i_str_ad : curS0fst + i_str_ad + s_cnst( curSc0fst + i_cnst_N ) - 1 ) = ...
    adh + s_str( curS0fst + i_str_Y3 ) .* fn;

% from compute_bridge_force()
% DJH - apply LPF to derivative only (so next sample will be affected)
s_str( curS0fst + i_str_ad : curS0fst + i_str_ad + s_cnst( curSc0fst + i_cnst_N ) - 1 ) = ...
    s_str( curS0fst + i_str_ad : curS0fst + i_str_ad + s_cnst( curSc0fst + i_cnst_N ) - 1 ) .* ...
    s_cnst( curSc0fst + i_cnst_lpf : curSc0fst + i_cnst_lpf + s_cnst( curSc0fst + i_cnst_N ) - 1 );
y = s_str( curS0fst + i_str_a : curS0fst + i_str_a + s_cnst( curSc0fst + i_cnst_N ) - 1 ) * ...
    s_str( curS0fst + i_str_G : curS0fst + i_str_G + s_cnst( curSc0fst + i_cnst_N ) - 1 )';

% from generate_output()
if( s_str( curS0fst + i_str_pluck_samples_remaining ) > 0 )
    % from update_pluck_actions();

```

```

s_str( curSOfst + i_str_pluck_samples_remaining ) = s_str( curSOfst + i_str_pluck_samples_remaining ) - 1;
if( s_str( curSOfst + i_str_y_pluck ) < s_str( curSOfst + i_str_y_pluck_target ) )
    s_str( curSOfst + i_str_y_pluck ) = s_str( curSOfst + i_str_y_pluck ) + ...
        ( s_str( curSOfst + i_str_PLUCK_VELOCITY ) * s_str( curSOfst + i_str_dt ) );
end
if( s_str( curSOfst + i_str_pluck_samples_remaining ) == 0 )
    % from string_release()
    s_str( curSOfst + i_str_actions ) = e_str_RELEASE;
    s_str( curSOfst + i_str_y_pluck ) = 0;
    % below necessary? "can't use lazy evaluation because not guaranteed to fill_buffer on a 1024-sample boundary"

    % * From String_cache_pa_c * %
    s_str( curSOfst + i_str_x1 ) = s_str( curSOfst + i_str_finger_position );
    s_str( curSOfst + i_str_K1 ) = s_str( curSOfst + i_str_Kf );
    s_str( curSOfst + i_str_R1 ) = s_str( curSOfst + i_str_R_FINGER ) * ...
        ( s_str( curSOfst + i_str_Kf ) / s_str( curSOfst + i_str_K_FINGER ) );

    s_str( curSOfst + i_str_x2 ) = 0;
    s_str( curSOfst + i_str_K2 ) = 0;
    s_str( curSOfst + i_str_R2 ) = 0;
    if( s_str( curSOfst + i_str_finger_position ) == 0 )
        s_str( curSOfst + i_str_x0 ) = 0;
        s_str( curSOfst + i_str_K0 ) = 0;
        s_str( curSOfst + i_str_R0 ) = 0;
    else
        s_str( curSOfst + i_str_K0 ) = s_str( curSOfst + i_str_K1 );
        s_str( curSOfst + i_str_R0 ) = s_str( curSOfst + i_str_R1 );
        if( s_str( curSOfst + i_str_finger_position ) < ( s_cnst( curScOfst + i_cnst_L ) - s_str( curSOfst +
i_str_FINGER_WIDTH ) ) )
            s_str( curSOfst + i_str_x0 ) = s_str( curSOfst + i_str_finger_position ) + s_str( curSOfst +
i_str_FINGER_WIDTH );
        else
            remaining_string = s_cnst( curScOfst + i_cnst_L ) - s_str( curSOfst + i_str_finger_position );
            s_str( curSOfst + i_str_x0 ) = s_str( curSOfst + i_str_finger_position ) + 0.5 * remaining_string;
        end
    end
end

% Below from cache_pa_c()
s_str( curSOfst + i_str_phix0 : curSOfst + i_str_phix0 + s_cnst( curScOfst + i_cnst_N ) - 1 ) = ...
    s_str( curSOfst + i_str_sqrt_two_div_L ) * sin( s_str( curSOfst + i_str_x0 ) ...
        * s_str( curSOfst + i_str_inside_phi : curSOfst + i_str_inside_phi + s_cnst( curScOfst + i_cnst_N ) -
1 ) );

```

```

s_str( curSOfst + i_str_phix1 : curSOfst + i_str_phix1 + s_cnst( curScOfst + i_cnst_N ) - 1 ) = ...
    s_str( curSOfst + i_str_sqrt_two_div_L ) * sin( s_str( curSOfst + i_str_x1 ) ...
        * s_str( curSOfst + i_str_inside_phi : curSOfst + i_str_inside_phi + s_cnst( curScOfst + i_cnst_N ) -
1 ) );

A00 = sum( s_str( curSOfst + i_str_X3 : curSOfst + i_str_X3 + s_cnst( curScOfst + i_cnst_N ) - 1 ) .* ...
    s_str( curSOfst + i_str_phix0 : curSOfst + i_str_phix0 + s_cnst( curScOfst + i_cnst_N ) - 1 ) .* ...
    s_str( curSOfst + i_str_phix0 : curSOfst + i_str_phix0 + s_cnst( curScOfst + i_cnst_N ) - 1 ) );
A01 = sum( s_str( curSOfst + i_str_X3 : curSOfst + i_str_X3 + s_cnst( curScOfst + i_cnst_N ) - 1 ) .* ...
    s_str( curSOfst + i_str_phix0 : curSOfst + i_str_phix0 + s_cnst( curScOfst + i_cnst_N ) - 1 ) .* ...
    s_str( curSOfst + i_str_phix1 : curSOfst + i_str_phix1 + s_cnst( curScOfst + i_cnst_N ) - 1 ) );
A11 = sum( s_str( curSOfst + i_str_X3 : curSOfst + i_str_X3 + s_cnst( curScOfst + i_cnst_N ) - 1 ) .* ...
    s_str( curSOfst + i_str_phix1 : curSOfst + i_str_phix1 + s_cnst( curScOfst + i_cnst_N ) - 1 ) .* ...
    s_str( curSOfst + i_str_phix1 : curSOfst + i_str_phix1 + s_cnst( curScOfst + i_cnst_N ) - 1 ) );

B00 = sum( s_str( curSOfst + i_str_Y3 : curSOfst + i_str_Y3 + s_cnst( curScOfst + i_cnst_N ) - 1 ) .* ...
    s_str( curSOfst + i_str_phix0 : curSOfst + i_str_phix0 + s_cnst( curScOfst + i_cnst_N ) - 1 ) .* ...
    s_str( curSOfst + i_str_phix0 : curSOfst + i_str_phix0 + s_cnst( curScOfst + i_cnst_N ) - 1 ) );
B01 = sum( s_str( curSOfst + i_str_Y3 : curSOfst + i_str_Y3 + s_cnst( curScOfst + i_cnst_N ) - 1 ) .* ...
    s_str( curSOfst + i_str_phix0 : curSOfst + i_str_phix0 + s_cnst( curScOfst + i_cnst_N ) - 1 ) .* ...
    s_str( curSOfst + i_str_phix1 : curSOfst + i_str_phix1 + s_cnst( curScOfst + i_cnst_N ) - 1 ) );
B11 = sum( s_str( curSOfst + i_str_Y3 : curSOfst + i_str_Y3 + s_cnst( curScOfst + i_cnst_N ) - 1 ) .* ...
    s_str( curSOfst + i_str_phix1 : curSOfst + i_str_phix1 + s_cnst( curScOfst + i_cnst_N ) - 1 ) .* ...
    s_str( curSOfst + i_str_phix1 : curSOfst + i_str_phix1 + s_cnst( curScOfst + i_cnst_N ) - 1 ) );

M00 = A00 * s_str( curSOfst + i_str_K1 ) + B00 * s_str( curSOfst + i_str_R1 ) + 1;
M01 = A01 * s_str( curSOfst + i_str_K1 ) + B01 * s_str( curSOfst + i_str_R1 );
M11 = A11 * s_str( curSOfst + i_str_K1 ) + B11 * s_str( curSOfst + i_str_R1 ) + 1;

L3 = -1 / ( M00 * M11 - M01 * M01 );
s_str( curSOfst + i_str_D8 ) = -1 / M00;
s_str( curSOfst + i_str_D9 ) = M01 * s_str( curSOfst + i_str_D8 );
s_str( curSOfst + i_str_D10 ) = -M01 * L3;
s_str( curSOfst + i_str_D11 ) = M00 * L3;

% "extra inv_A for any remaining tick_pluck() which occurs before a new buffer is called (i.e. the switch/case
in fill_buffer)"

matrix_A = [    B00 * s_str( curSOfst + i_str_R0 ) + A00 * s_str( curSOfst + i_str_K0 ) + 1, ...
                B01 * s_str( curSOfst + i_str_R0 ) + A01 * s_str( curSOfst + i_str_K0 ), ...
                0; ...

                B01 * s_str( curSOfst + i_str_R1 ) + A01 * s_str( curSOfst + i_str_K1 ), ...
                B11 * s_str( curSOfst + i_str_R1 ) + A11 * s_str( curSOfst + i_str_K1 ) + 1, ...
                0; ...

```

```

0, ...
0, ...
1; ];

inv_A = inv( matrix_A );
s_str( curSOfst + i_str_inv_A : curSOfst + i_str_inv_A + n_str_inv_A ) = [ inv_A(1,:) inv_A(2,:) inv_A(3,:) ];

% now that the recache is done, reset the flag
s_str( curSOfst + i_str_recache ) = 0;

end
end

elseif( s_str( curSOfst + i_str_actions ) == e_str_RELEASE )
% * From String_tick_release * %
% tick_release() says: 'hands-free' modes
ah = s_str( curSOfst + i_str_X1 : curSOfst + i_str_X1 + s_cnst( curScOfst + i_cnst_N ) - 1 ) .* ...
s_str( curSOfst + i_str_a : curSOfst + i_str_a + s_cnst( curScOfst + i_cnst_N ) - 1 ) + ...
s_str( curSOfst + i_str_X2 : curSOfst + i_str_X2 + s_cnst( curScOfst + i_cnst_N ) - 1 ) .* ...
s_str( curSOfst + i_str_ad : curSOfst + i_str_ad + s_cnst( curScOfst + i_cnst_N ) - 1 );
adh = s_str( curSOfst + i_str_Y1 : curSOfst + i_str_Y1 + s_cnst( curScOfst + i_cnst_N ) - 1 ) .* ...
s_str( curSOfst + i_str_a : curSOfst + i_str_a + s_cnst( curScOfst + i_cnst_N ) - 1 ) + ...
s_str( curSOfst + i_str_Y2 : curSOfst + i_str_Y2 + s_cnst( curScOfst + i_cnst_N ) - 1 ) .* ...
s_str( curSOfst + i_str_ad : curSOfst + i_str_ad + s_cnst( curScOfst + i_cnst_N ) - 1 );

if( s_str( curSOfst + i_str_finger_position ) == 0 )
% from tick_free()
% necessary to avoid aliasing
s_str( curSOfst + i_str_a : curSOfst + i_str_a + s_cnst( curScOfst + i_cnst_N ) - 1 ) = ah;
s_str( curSOfst + i_str_ad : curSOfst + i_str_ad + s_cnst( curScOfst + i_cnst_N ) - 1 ) = adh;
else
% from tick_release()
% 'hands-free' string displacement under force locations
s_str( curSOfst + i_str_y0h ) = ...
s_str( curSOfst + i_str_phix0 : curSOfst + i_str_phix0 + s_cnst( curScOfst + i_cnst_N ) - 1 ) * ah';
s_str( curSOfst + i_str_y1h ) = ...
s_str( curSOfst + i_str_phix1 : curSOfst + i_str_phix1 + s_cnst( curScOfst + i_cnst_N ) - 1 ) * ah';
s_str( curSOfst + i_str_v0h ) = ...
s_str( curSOfst + i_str_phix0 : curSOfst + i_str_phix0 + s_cnst( curScOfst + i_cnst_N ) - 1 ) * adh';
s_str( curSOfst + i_str_v1h ) = ...
s_str( curSOfst + i_str_phix1 : curSOfst + i_str_phix1 + s_cnst( curScOfst + i_cnst_N ) - 1 ) * adh';

ans0 = s_str( curSOfst + i_str_K1 ) * s_str( curSOfst + i_str_y0h ) + s_str( curSOfst + i_str_R1 ) *
s_str( curSOfst + i_str_v0h );

```

```

        ans1 = s_str( curS0fst + i_str_K1 ) * s_str( curS0fst + i_str_y1h ) + s_str( curS0fst + i_str_R1 ) *
s_str( curS0fst + i_str_v1h );
        F1 = ans0 * s_str( curS0fst + i_str_D10 ) + ans1 * s_str( curS0fst + i_str_D11 );
        s_str( curS0fst + i_str_F0 ) = ans0 * s_str( curS0fst + i_str_D8 ) + F1 * s_str( curS0fst + i_str_D9 );

        % apply forces
        fn = s_str( curS0fst + i_str_phix0 : curS0fst + i_str_phix0 + s_cnst( curSc0fst + i_cnst_N ) - 1 ) * ...
            s_str( curS0fst + i_str_F0 ) + ...
            s_str( curS0fst + i_str_phix1 : curS0fst + i_str_phix1 + s_cnst( curSc0fst + i_cnst_N ) - 1 ) * F1;
        s_str( curS0fst + i_str_a : curS0fst + i_str_a + s_cnst( curSc0fst + i_cnst_N ) - 1 ) = ...
            ah + s_str( curS0fst + i_str_X3 : curS0fst + i_str_X3 + s_cnst( curSc0fst + i_cnst_N ) - 1 ) .* fn;
        s_str( curS0fst + i_str_ad : curS0fst + i_str_ad + s_cnst( curSc0fst + i_cnst_N ) - 1 ) = ...
            adh + s_str( curS0fst + i_str_Y3 : curS0fst + i_str_Y3 + s_cnst( curSc0fst + i_cnst_N ) - 1 ) .* fn;
    end

    % from compute_bridge_force()
    % DJH - apply LPF to derivative only (so next sample will be affected)
    s_str( curS0fst + i_str_ad : curS0fst + i_str_ad + s_cnst( curSc0fst + i_cnst_N ) - 1 ) = ...
        s_str( curS0fst + i_str_ad : curS0fst + i_str_ad + s_cnst( curSc0fst + i_cnst_N ) - 1 ) .* ...
        s_cnst( curSc0fst + i_cnst_lpf : curSc0fst + i_cnst_lpf + s_cnst( curSc0fst + i_cnst_N ) - 1 );
    y = s_str( curS0fst + i_str_a : curS0fst + i_str_a + s_cnst( curSc0fst + i_cnst_N ) - 1 ) * ...
        s_str( curS0fst + i_str_G : curS0fst + i_str_G + s_cnst( curSc0fst + i_cnst_N ) - 1 )';
end

    y_temp = y_temp + y;
end

    y = y_temp / s_cnst( curSc0fst + i_cnst_mlt ); % average the result (so generating 1 output per tick)

    % Collect each string's output sample
    buf_string{ string_num + 1 }( buf_pos ) = y;

end

% Average the resulting string output samples
buf_strings( buf_pos ) = mean( [ buf_string{1}( buf_pos ), buf_string{2}( buf_pos ), ...
                                buf_string{3}( buf_pos ), buf_string{4}( buf_pos ) ] );

%% Set and apply the correct convolution vectors
% increment the counter and calculate segments
convPos = mod( convPos, convBufLen ) + 1;
convBufSegA_end = min( convPos + convLen - 1, convBufLen );

```



```

convBufSegB_end = convLen - ( convBufSegA_end - ( convPos - 1 ) );
convSegA_end = ( convBufSegA_end - ( convPos - 1 ) );
convSegB_start = convSegA_end + 1;

for string_num=0:3
    s_str( i_str_curStr ) = string_num;

    % * From String_fundamental * %
    % calculate offsets
    curSOfst = o_str * s_str( i_str_curStr );
    curScOfst = o_cnst * s_str( i_str_curStr );
    %

    % Get current fundamental frequency
    freq = s_str( curSOfst + i_str_w0 ) / (2*pi);
    if( s_str( curSOfst + i_str_finger_position ) > 0 )
        freq = freq * ( 1 / ( s_str( curSOfst + i_str_finger_position ) / s_cnst( curScOfst + i_cnst_L ) ) );
    end

    % Get current amplitude, in units RMS
    ampl = sqrt( mean( [ buf_string{ string_num + 1 }( buf_pos : end ) ...
        buf_string{ string_num + 1 }( 1 : buf_pos - 1 ) ].^2 ) ) * ampl_ratio;

    freq_change = freq ~= convVecsFreqs( i_conv_val, string_num + 1 );
    amp_change = ampl ~= convVecsAmps( i_conv_val, string_num + 1 );

    % if either freq or lev different...
    if( freq_change || amp_change )

        if( freq_change )
            % * From IR_activeConv * %
            freqVector = convFundStrings{ string_num + 1 };

            % * From freqLocate * %
            % Set up some default values
            sA = 0;
            sB = 0;
            iA = 1;
            iB = length( freqVector );

            % Handle boundary cases
            if( freqVector( 1 ) >= freq )
                sA = 1;
                iB = 2;
            % We're at the base of the list
            % so use 100% of frequency A (and 0% of B)
        end
    end
end

```

```

else
    if( freqVector( end ) <= freq ) % We're at the top of the list
        SB = 1; % so use 100% of frequency B (and 0% of A)
        iA = iB - 1;
    else % We're somewhere within the list
        % Locate the upper frequency
        iB = 2;
        while( freqVector( iB ) <= freq && iB < length( freqVector ) )
            iB = iB + 1;
        end
        iA = iB - 1;

        range_dist = freqVector( iB ) - freqVector( iA );
        freq_dist = freq - freqVector( iA ); % distance from A
        SB = 1 / ( range_dist / freq_dist );
        SA = 1 - SB;
    end
end

convVecsFreqs( i_conv_idxA, string_num + 1 ) = iA;
convVecsFreqs( i_conv_idxB, string_num + 1 ) = iB;
convVecsFreqs( i_conv_amtA, string_num + 1 ) = SA;
convVecsFreqs( i_conv_amtB, string_num + 1 ) = SB;
convVecsFreqs( i_conv_val, string_num + 1 ) = freq;
end

if( amp_change )
    for idx=i_conv_idxA:i_conv_idxB
        f_idx = convVecsFreqs( idx, string_num + 1 ); % Get freq index A, then B (next pass)

        ampVector = convAmpsStrings{ string_num + 1 }( :, f_idx ); % Get amp list, using freq index

        % Set up some default values
        SA = 0;
        SB = 0;
        iA = 1;
        iB = length( ampVector );

        % Handle boundary cases
        if( ampVector( 1 ) >= ampl ) % We're at the base of the list
            SA = 1; % so use 100% of amplitude A (and 0% of B)
            iB = 2;
        else
            if( ampVector( end ) <= ampl ) % We're at the top of the list

```

```

        sB = 1; % so use 100% of amplitude B (and 0% of A)
        iA = iB - 1;
    else % We're somewhere within the list
        % Locate the upper amplitude
        iB = 2;
        while( ampVector( iB ) <= ampl && iB < length( ampVector ) )
            iB = iB + 1;
        end
        iA = iB - 1;

        range_dist = ampVector( iB ) - ampVector( iA );
        ampl_dist = ampl - ampVector( iA ); % distance from A
        sB = 1 / ( range_dist / ampl_dist );
        sA = 1 - sB;
    end
end

% Store this index's values
convVecsAmps( idx, string_num + 1 ) = iA;
convVecsAmps( idx + o_conv_idx, string_num + 1 ) = iB; % so i_conv_idxA holds set_A(idx A and B)
convVecsAmps( idx + o_conv_amt, string_num + 1 ) = sA;
convVecsAmps( idx + o_conv_idx + o_conv_amt, string_num + 1 ) = sB; % so i_conv_amtA holds set_A(amt A and B)
end

convVecsAmps( i_conv_val, string_num + 1 ) = ampl;
end

% Generate a convolution vector proportional to the frequency and amplitude of the string
% Vector based on first frequency (A)
f_iA = convVecsFreqs( i_conv_idxA, string_num + 1 );
aA_iA = convVecsAmps( i_conv_idxA, string_num + 1 );
aA_iB = convVecsAmps( i_conv_idxA + o_conv_idx, string_num + 1 );
aA_sA = convVecsAmps( i_conv_idxA + o_conv_amt, string_num + 1 );
aA_sB = convVecsAmps( i_conv_idxA + o_conv_idx + o_conv_amt, string_num + 1 );
% Limit rate of change between vectors
if( abs( aA_sA - aA_sA_h ) > max_conv_ch )
    aA_sA = aA_sA_h + ( sign( aA_sA - aA_sA_h ) * max_conv_ch );
end
if( abs( aA_sB - aA_sB_h ) > max_conv_ch )
    aA_sB = aA_sB_h + ( sign( aA_sB - aA_sB_h ) * max_conv_ch );
end
vecAmpA = ( aA_sA * convVecsStrings{ aA_iA, string_num + 1 }( :, f_iA ) ) + ...
    ( aA_sB * convVecsStrings{ aA_iB, string_num + 1 }( :, f_iA ) );

```

```

% Vector based on second frequency (B)
f_iB = convVecsFreqs( i_conv_idxB, string_num + 1 );
aB_iA = convVecsAmps( i_conv_idxB, string_num + 1 );
aB_iB = convVecsAmps( i_conv_idxB + o_conv_idx, string_num + 1 );
aB_sA = convVecsAmps( i_conv_idxB + o_conv_amt, string_num + 1 );
aB_sB = convVecsAmps( i_conv_idxB + o_conv_idx + o_conv_amt, string_num + 1 );
% Limit rate of change between vectors
if( abs( aB_sA - aB_sA_h ) > max_conv_ch )
    aB_sA = aB_sA_h + ( sign( aB_sA - aB_sA_h ) * max_conv_ch );
end
if( abs( aB_sB - aB_sB_h ) > max_conv_ch )
    aB_sB = aB_sB_h + ( sign( aB_sB - aB_sB_h ) * max_conv_ch );
end
vecAmpB = ( aB_sA * convVecsStrings{ aB_iA, string_num + 1 }( :, f_iB ) ) + ...
    ( aB_sB * convVecsStrings{ aB_iB, string_num + 1 }( :, f_iB ) );
% Store state of amplitude-based convolution vectors
aA_sA_h = aA_sA;
aA_sB_h = aA_sB;
aB_sA_h = aB_sA;
aB_sB_h = aB_sB;
% Final vector, based on both frequencies proportionally
f_sA = convVecsFreqs( i_conv_amtA, string_num + 1 );
f_sB = convVecsFreqs( i_conv_amtB, string_num + 1 );
convVecs( :, string_num + 1 ) = ( f_sA * vecAmpA ) + ( f_sB * vecAmpB );

end

% clear the convolution buffer current position (will be the new last position)
convBuf{ string_num + 1 }( convDlyPos ) = 0;
% perform the convolution for this sample, for each string
convBuf{ string_num + 1 }( convPos : convBufSegA_end ) = convBuf{ string_num + 1 }( convPos : convBufSegA_end ) + ...
    ( buf_string{ string_num + 1 }( buf_pos ) * convVecs( 1 : convSegA_end, string_num + 1 ) );
convBuf{ string_num + 1 }( 1 : convBufSegB_end ) = convBuf{ string_num + 1 }( 1 : convBufSegB_end ) + ...
    ( buf_string{ string_num + 1 }( buf_pos ) * convVecs( convSegB_start : convLen, string_num + 1 ) );

end

% delayed output to buf_body:
convDlyPos = mod( convPos - convDelay, convBufLen ) + 1;
buf_body( buf_pos ) = ( convBuf{1}( convDlyPos ) + convBuf{2}( convDlyPos ) + ...
    convBuf{3}( convDlyPos ) + convBuf{4}( convDlyPos ) ) / 4;

%% Obtain the FFTv vectors for body and all strings
% increment the counter and calculate segments
fftvPos_prev = fftvPos;

```

```

fftvPos    = mod( fftvPos, fftvLen ) + 1;

for string_num=1:4
    % append the new samples to the (soon to be new) end of the vector
    fftvTString{ string_num }( fftvPos_prev ) = buf_string{ string_num }( buf_pos );
    % obtain frequency velocities for all vectors
    fftvFString{ string_num } = ...
        fft( [ fftvTString{ string_num }( fftvPos : end ), fftvTString{ string_num }( 1 : fftvPos - 1 ) ] ) ...
        - fftvFString{ string_num };
end

fftvTBody( fftvPos_prev ) = buf_body( buf_pos );
fftvFBody    = fft( [ fftvTBody( fftvPos : end ), fftvTBody( 1 : fftvPos - 1 ) ] ) - fftvFBody;

% feed them back to the strings
% 2 ways to approach - 1) accurate (by proportional mix of closest two frequencies), or
%                        2) fast      (by matching CLOSEST frequencies)
% - using method 2 for now...

% create mode velocity data for all strings
fftvFb{1} = real( fftvFString{2}( fftvBMatches{ 1 } ) ) + real( fftvFString{3}( fftvBMatches{ 1 } ) ) + real( fftvFString{4}
( fftvBMatches{ 1 } ) );
fftvFb{2} = real( fftvFString{1}( fftvBMatches{ 2 } ) ) + real( fftvFString{3}( fftvBMatches{ 2 } ) ) + real( fftvFString{4}
( fftvBMatches{ 2 } ) );
fftvFb{3} = real( fftvFString{1}( fftvBMatches{ 3 } ) ) + real( fftvFString{2}( fftvBMatches{ 3 } ) ) + real( fftvFString{4}
( fftvBMatches{ 3 } ) );
fftvFb{4} = real( fftvFString{1}( fftvBMatches{ 4 } ) ) + real( fftvFString{2}( fftvBMatches{ 4 } ) ) + real( fftvFString{3}
( fftvBMatches{ 4 } ) );

for string_num=0:3
    fftvStr = ( fftvFb{ string_num + 1 } * str_gain ) + ...
        ( real( fftvFBody( fftvBMatches{ string_num + 1 } ) ) * body_gain );
    % apply velocity vectors to all strings
    s_str( o_str * string_num + i_str_ad : o_str * string_num + i_str_ad + length( fftvStr ) - 1 ) = ...
        s_str( o_str * string_num + i_str_ad : o_str * string_num + i_str_ad + length( fftvStr ) - 1 ) + ...
        ( fftvStr .* s_str( curSofst + i_str_G : curSofst + i_str_G + length( fftvStr ) - 1 ) );
end

%% Show progress
if( mod( t, notify_sample ) == 0 )
    fprintf( 'Processing %.2fs...\n', t / fs )
end

end
toc

```

CelloSim_finalise.m

```
% *****
% CELLO SIMULATION
%
% Name:    CelloSim_finalise.m
% Date:    02/05/2016
% Author:  Dave Humphreys, c1322278
% Desc:    Complete storage of data generated by
%           CelloSim_engine.m, and write resulting
%           audio to PCM files.
% *****

% Write remaining buffers to disk
f_output.strings( 1, buf_start : t - 1 ) = buf_strings( 1, 1 : t - buf_start );
f_output.string1( 1, buf_start : t - 1 ) = buf_string{1}( 1, 1 : t - buf_start );
f_output.string2( 1, buf_start : t - 1 ) = buf_string{2}( 1, 1 : t - buf_start );
f_output.string3( 1, buf_start : t - 1 ) = buf_string{3}( 1, 1 : t - buf_start );
f_output.string4( 1, buf_start : t - 1 ) = buf_string{4}( 1, 1 : t - buf_start );
f_output.body( 1, buf_start : t - 1 ) = buf_body( 1, 1 : t - buf_start );

% Save generated audio as PCM files
load( 'output_temp.mat', 'strings' );
audiowrite( 'output/strings.wav', ( strings / max( abs( strings ) ) )', fs );
load( 'output_temp.mat', 'string1' );
audiowrite( 'output/string1.wav', ( string1 / max( abs( string1 ) ) )', fs );
load( 'output_temp.mat', 'string2' );
audiowrite( 'output/string2.wav', ( string2 / max( abs( string2 ) ) )', fs );
load( 'output_temp.mat', 'string3' );
audiowrite( 'output/string3.wav', ( string3 / max( abs( string3 ) ) )', fs );
load( 'output_temp.mat', 'string4' );
audiowrite( 'output/string4.wav', ( string4 / max( abs( string4 ) ) )', fs );
load( 'output_temp.mat', 'body' );
audiowrite( 'output/body.wav', ( body / max( abs( body ) ) )', fs );
output = ( body * mix_ratio ) + ( strings * ( 1 - mix_ratio ) );
audiowrite( 'output/output.wav', ( output / max( abs( output ) ) )', fs );
```

CelloSim_melody.m

```
% *****
% CELLO SIMULATION
%
% Name:    CelloSim_melody.m
% Date:    02/05/2016
% Author:  Dave Humphreys, c1322278
% Desc:    Controls various engine parameters during
%           execution of CelloSim_engine.m, which
%           produce effects such as melodies,
%           glissando / legato effects etc.
%
% *****

% Get current string's vector offsets
curSOfst = o_str * active_string;
curScOfst = o_cnst * active_string;

% "melody 1" - glissando
glissando_start = 1.0 * fs;
glissando_end   = 2.5 * fs;
glissando_range = 0.5 * s_cnst( curScOfst + i_cnst_L );
glissando_step  = glissando_range / ( glissando_end - glissando_start );

if( active_melody == 1 )
    if( t - 1 <= glissando_start )
        s_str( curSOfst + i_str_finger_position_req ) = s_cnst( curScOfst + i_cnst_L );
    end
    if( t >= glissando_start && t < glissando_end )
        s_str( curSOfst + i_str_finger_position_req ) = s_str( curSOfst + i_str_finger_position_req ) - glissando_step;
        if( s_str( curSOfst + i_str_finger_position_req ) < 0 )
            s_str( curSOfst + i_str_finger_position_req ) = 0;
        end
        s_str( curSOfst + i_str_Kf ) = 5.0 * s_str( curSOfst + i_str_K_FINGER );
        s_str( curSOfst + i_str_recache ) = 1;
    end
    if( t >= glissando_end )
        vibrato_active = true;
    end
end
```

Convolve_construct.m

```
% *****
% CELLO SIMULATION
%
% Name:    Convolve_construct.m
% Date:    02/05/2016
% Author:  Dave Humphreys, c1322278
% Desc:    Set up variables needed by the in-line
%           version of the Convolve class within
%           CelloSim_engine.m
%
% *****

%% Convolve construction code
% (convLen is define within IR_extractSets)

convDelay_t = 0.0008; % delay time, in seconds
convDelay   = round( convDelay_t * fs ) + 1;

convBufLen = convLen + convDelay;

convBuf     = cell( 1, 4 );
convBuf{1} = zeros( convBufLen, 1 );
convBuf{2} = zeros( convBufLen, 1 );
convBuf{3} = zeros( convBufLen, 1 );
convBuf{4} = zeros( convBufLen, 1 );
%
convPos     = 1;
convDlyPos  = mod( convPos - convDelay, convBufLen );
```


FFTv_construct.m

```
% *****
% CELLO SIMULATION
%
% Name:   FFTv_construct.m
% Date:   02/05/2016
% Author: Dave Humphreys, c1322278
% Desc:   Set up variables needed by the in-line
%          version of the FFTv class within
%          CelloSim_engine.m. Matches FFT bins to
%          the modes within each string's vector.
%
% *****

%% Setup code

fftvLen = 2048;

% time-domain buffers
fftvTString = cell( 1, 4 );
fftvTString{1} = zeros( 1, fftvLen );
fftvTString{2} = zeros( 1, fftvLen );
fftvTString{3} = zeros( 1, fftvLen );
fftvTString{4} = zeros( 1, fftvLen );
fftvTBody = zeros( 1, fftvLen );

% frequency-domain buffers
fftvFString = cell( 1, 4 );
fftvFString{1} = zeros( 1, fftvLen );
fftvFString{2} = zeros( 1, fftvLen );
fftvFString{3} = zeros( 1, fftvLen );
fftvFString{4} = zeros( 1, fftvLen );
fftvFBody = zeros( 1, fftvLen );

% string-specific velocity buffers
fftvFb = cell( 1, 4 );

fftvPos = 2;

% frequencies represented by the fft bins
fftvBinFs = ( 0 : ( fftvLen / 2 ) - 1 ) * ( fs / fftvLen );
```

```

%% Match a bin to each mode for all strings
fprintf( 'FFTv_construct(): Matching FFT bins to string modes...\n' );
fftvBMatches = cell( 4 ); %s_cnst( o_cnst_1 + i_cnst_N );

for s=0:3
    i_target = 0;
    err_prev = flintmax;
    err_max = 0;
    err_avg = 0;
    err_ctr = 0;
    matches = [];

    for ii=2:length( fftvBinFs )
        % Only process if an index remains within w0 to target
        if( i_target < s_cnst( o_cnst * s + i_cnst_N ) )
            % Check to see if target frequency is less than the Nyquist frequency
            if( s_str( o_str * s + i_str_w0 + i_target ) / (2*pi) < (fs/2) )
                % Generate an error figure for this match
                err = abs( ( s_str( o_str * s + i_str_w0 + i_target ) / (2*pi) ) - fftvBinFs( ii ) );

                % If it's worse than the previous error figure, accept this as a match
                if( err >= err_prev )
                    matches( i_target + 1 ) = ii - 1;
                    if( err_prev > err_max )
                        err_max = err_prev;
                    end
                    err_avg = err_avg + err_prev;
                    err_ctr = err_ctr + 1;
                    i_target = i_target + 1;
                    err_prev = flintmax;
                % otherwise, just store the error figure for use in the next iteration
                else
                    err_prev = err;
                end
            end
        end
    end
    fftvBMatches{ s + 1 } = matches;

    % Report error figures
    fprintf( 'String %d complete: average error = %f, maximum error = %f\t(Hz)\n', s, err_avg / err_ctr, err_max );
end
fprintf( '\n' );

```

IR_construct.m

```
% *****
% CELLO SIMULATION
%
% Name:    IR_construct.m
% Date:    02/05/2016
% Author:  Dave Humphreys, c1322278
% Desc:    Set up variables needed by the in-line
%           version of the IR class within
%           CelloSim_engine.m.
% *****

i_conv_idxA = 1;
i_conv_idxB = 2;
o_conv_amt = 2;          % offset from indices to amounts
i_conv_amtA = i_conv_idxA + o_conv_amt;
i_conv_amtB = i_conv_idxB + o_conv_amt;
i_conv_val = i_conv_amtB + 1;
o_conv_idx = 5;          % offset from first indices set to second
%i_conv_idxA (second) = 6; (i.e. o_conv_idx + i_conv_idxA )
%i_conv_idxB (second) = 7; (i.e. o_conv_idx + i_conv_idxB )
%i_conv_amtA (second) = 8; (i.e. o_conv_idx + i_conv_amtA )
%i_conv_amtB (second) = 9; (i.e. o_conv_idx + i_conv_amtB )

convVecsFreqs = zeros( 5, num_strings );          % Holds the frequency data matching each currently selected conv. vector
convVecsAmps = ones( 9, num_strings ) * flintmax; % Holds the amplitude data matching each currently selected conv. vector
IR_extractSets();
```

IR_extractSets.m

```
% *****
% CELLO SIMULATION
%
% Name:    IR_extractSets.m
% Date:    02/05/2016
% Author:  Dave Humphreys, c1322278
% Desc:    For use by the in-line version of the IR
%           class. Calculates convolution vectors
%           for each amplitude level and note of
%           each string.
%
% *****

fade_time    = 0.05;                % time to fade convolution vector over (s)
convLen      = round( fs * fade_time ); % length of vectors when faded to fade_time s

fprintf( 'IR: extracting IR sets...\n' );

convVecsStrings = cell( num_levels, num_strings ); % ( velocity, string ) = [ freq1; freq2; ... ]
convFundStrings = cell( 1, num_strings );
for string_num=1:num_strings
    IR_extractSet();
end

convVecs = zeros( convLen, num_strings );

fprintf( 'IR: IR sets extracted.\n\n' );
```

IR_extractSet.m

```
% *****
% CELLO SIMULATION
%
% Name:    IR_extractSet.m
% Date:    02/05/2016
% Author:  Dave Humphreys, c1322278
% Desc:    For use by the in-line version of the IR
%           class. Creates a set of matrices of raw
%           (dry) and real (wet) sounds for each
%           level and note of a given string by
%           loading them in, then calculates a
%           matrix containing the appropriate
%           convolution vectors.
%
% *****

% Load all samples ready for convolution extractions
audioDry = cell( 1, num_levels );
audioWet = cell( 1, num_levels );
for l=1:num_levels
    for ii=1:num_files
        [ in, srcRate ] = audioread( strcat( 'inputSim/sim_', num2str( string_num ), '_', num2str( ii ), '_', num2str( l ),
        '.wav' ) );
        if( srcRate ~= fs )
            IR_resample();
            in = out;
        end
        mIn = audioDry{ l };
        mInRef = {};
        vIn = in;
        IR_extend();
        audioDry{ l } = mOut;
    end

    for ii=1:num_files
        fprintf( strcat( 'Reading: inputTarget/target_', num2str( string_num ), '_', num2str( ii ), '_', num2str( l ),
        '.wav...\n' ) );
        [ in, srcRate ] = audioread( strcat( 'inputTarget/target_', num2str( string_num ), '_', num2str( ii ), '_', num2str( l ),
        '.wav' ) );
```

```

    if( srcRate ~= fs )
        IR_resample();
        in = out;
    end
    mIn = audioWet{ 1 };
    mInRef = audioDry;
    vIn = in;
    IR_extend();
    audioWet{ 1 } = mOut;
end

s = min( size( audioDry{ 1 }, 1 ), size( audioWet{ 1 }, 1 ) );
audioDry{ 1 } = audioDry{ 1 }( 1:s, : );
audioWet{ 1 } = audioWet{ 1 }( 1:s, : );
end

% Obtain levels and fundamental frequencies, for convolution vector mapping
IR_setFundsAmps();

% Obtain convolution vectors and their respective fundamentals
for l=1:num_levels
    IR_extract();
end

```

IR_extract.m

```
% *****
% CELLO SIMULATION
%
% Name:    IR_extract.m
% Date:    02/05/2016
% Author:  Dave Humphreys, c1322278
% Desc:    For use by the in-line version of the IR
%           class. Accepts two matrices, one of raw
%           string samples and one of target (real)
%           samples, and returns a matrix of
%           convolution vectors needed to transform
%           raw sounds to real sounds.
% *****

% Set up convolution vector fade
fade = linspace( 1, 0, convLen )'; % basic linear fade here

for ii=1:size( audioDry{ round( num_levels / 2 ) }, 2 )
    % Fade sample start and ends proportionally to the sample frequency
    fade_se_t = 1 / convFundStrings{ string_num }( ii );
    fade_se_d = [ linspace( 0, 1, round( fade_se_t * fs ) ) ...
                  ones( 1, size( audioDry{ 1 }, 1 ) - 2 * round( fade_se_t * fs ) ) ...
                  linspace( 1, 0, round( fade_se_t * fs ) ) ]';
    fade_se_w = [ linspace( 0, 1, round( fade_se_t * fs ) ) ...
                  ones( 1, size( audioWet{ 1 }, 1 ) - 2 * round( fade_se_t * fs ) ) ...
                  linspace( 1, 0, round( fade_se_t * fs ) ) ]';

    % Convert inputs into frequency domain
    audioDry_temp( 1 : size( audioDry{ 1 }( :, ii ), 1 ), ii ) = fft( audioDry{ 1 }( :, ii ) .* fade_se_d );
    audioWet_temp( 1 : size( audioWet{ 1 }( :, ii ), 1 ), ii ) = fft( audioWet{ 1 }( :, ii ) .* fade_se_w );
    % Produce convolution vectors, in time domain
    convVecsStrings{ 1, string_num }( :, ii ) = real( ifft( audioWet_temp( :, ii ) ./ audioDry_temp( :, ii ) ) );
    % fade the response (to smooth convolution)
    fade_temp = [ fade; zeros( length( convVecsStrings{ 1, string_num }( :, ii ) ) - convLen, 1 ) ];
    convVecsStrings{ 1, string_num }( :, ii ) = convVecsStrings{ 1, string_num }( :, ii ) .* fade_temp;
end

% Remove all post-fade content
convVecsStrings{ 1, string_num } = convVecsStrings{ 1, string_num }( 1 : convLen, : );
```

IR_extend.m

```
% *****
% CELLO SIMULATION
%
% Name:    IR_extend.m
% Date:    02/05/2016
% Author:  Dave Humphreys, c1322278
% Desc:    For use by the in-line version of the IR
%           class. Accepts matrices mIn and mInRef,
%           and vector vIn. Returns mOut, which
%           matches mInRef in dimensions and
%           additionally contains vIn.
% *****

mOut = mIn;
vOut = vIn;

% Make vector at least equal to reference length
if( length( vIn ) < size( mInRef, 1 ) )
    vOut = [ vIn; zeros( size( mInRef, 1 ) - length( vIn ), 1 ) ];
end

% Find difference
d = size( mIn, 1 ) - length( vOut );

% If we've not enough elements available in the matrix
if( d < 0 )
    mOut = zeros( length( vOut ), size( mIn, 2 ) );
    for ii=1:size( mIn, 2 ) % per column
        mOut( :, ii ) = [ mIn( :, ii ); zeros( abs(d), 1 ) ];
    end
end

% IF we've not enough elements available in the vector
if( d > 0 )
    vOut = [ vOut; zeros( d, 1 ) ];
end

mOut( :, size( mOut, 2 ) + 1 ) = vOut;
```


IR_resample.m

```
% *****  
% CELLO SIMULATION  
%  
% Name:    IR_resample.m  
% Date:    02/05/2016  
% Author:  Dave Humphreys, c1322278  
% Desc:    For use by the in-line version of the IR  
%          class. Accepts an audio sample, source  
%          and destination sample frequencies, and  
%          returns a re-sampled version.  
%  
% *****  
  
out = [];  
target_len = round( length( in ) * ( fs / srcRate ) );  
step = length( in ) / target_len;  
jj = 1;  
for ii=1:target_len  
    out( ii ) = in( floor( jj ) );  
    jj = jj + step;  
end  
out = out';
```

IR_setFundsAmps.m

```
% *****
% CELLO SIMULATION
%
% Name:    IR_setFundsAmps.m
% Date:    02/05/2016
% Author:  Dave Humphreys, c1322278
% Desc:    For use by the in-line version of the IR
%           class. Accepts two matrices, one of raw
%           string samples (dry), the other of real
%           cello samples (wet), and extracts the
%           fundamental pitch and RMS amplitudes of
%           each.
% *****

min_fs = 10;           % lowest frequency to consider as fundamental
max_fs = 1350;         % highest frequency to consider as fundamental

% Set up fundamentals and amplitudes vectors
convFundStrings{ string_num } = zeros( size( audioDry{ round( num_levels / 2 ) }, 2 ), 1 ); % Stores frequencies matching
each extracted conv. vector
convAmpsStrings{ string_num } = zeros( num_levels, size( audioDry{ round( num_levels / 2 ) }, 2 ) ); % Stores amplitudes matching
each extracted conv. vector

for ii=1:size( audioDry{ round( num_levels / 2 ) }, 2 )
    % Extract fundamental frequencies (from likely medium-level, dry strings, for clarity)
    audioDry_fft = fft( audioDry{ round( num_levels / 2 ) }( :, ii ) );
    fs_step = fs / length( audioDry_fft );
    start_bin = ceil( min_fs / fs_step );
    end_bin = ceil( max_fs / fs_step );
    [ m, idx ] = max( audioDry_fft( start_bin : end_bin ) ); % get maximum value in FFT
    convFundStrings{ string_num }( ii ) = fs_step * ( idx + start_bin );
    fprintf( 'string %d, note %d: Found note @ %f Hz\n', string_num, ii, convFundStrings{ string_num }( ii ) );

    for l=1:num_levels
        % Extract amplitudes (from dry strings)
        convAmpsStrings{ string_num }( l, ii ) = sqrt( mean( audioDry{ l }( :, ii ).^2 ) );
        fprintf( 'string %d, note %d, lev %d: Found amplitude %f units RMS\n', string_num, ii, l, convAmpsStrings{ string_num }( l, ii ) );
    end
end
end
```

Strings_init.m

```
% *****
% STRING SIMULATION
%
% Name:   Strings_init.m
% Date:   02/05/2016
% Author: Dave Humphreys, c1322278
% Desc:   In-line version for Cello Simulator. Loads
%          constants, and constructs four individual
%          strings, applying custom settings (string
%          damping, diameter, linear density and
%          Young's modulus) where requested.
%          String code ported from "Artifastring",
%          (c) 2013 G. Percival.
%
% *****

fprintf( 'Strings_init: setting up strings...\n' );

% Load string constants
String_constants();
String_h();

use_optimised = true;

% START OF OPTIMISED SETTINGS
if( use_optimised )
    % (example custom settings:)
    fprintf( '** USING OPTIMISED SETTINGS... **\n' );
    if( active_string == 0 )
        if( active_pos == 0 )
            s_cnst( o_cnst_1 + i_cnst_d ) = 0.000100;
            s_cnst( o_cnst_1 + i_cnst_pl ) = 0.038075;
            s_cnst( o_cnst_1 + i_cnst_E ) = 22872528309.804886;
            s_cnst( o_cnst_1 + i_cnst_T ) = 290.229319;
            String_reset();
            s_cnst( o_cnst_1 + i_cnst_lpf : o_cnst_1 + i_cnst_lpf + s_cnst( o_cnst_1 + i_cnst_N ) - 1 ) = ...
                [ 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, ...
                  1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, ...
                  1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, ...
                  1.000000, 1.000000, 1.000000, 1.000000, 0.920255, 0.887579, 0.863037, 0.843018, ...
```

```

0.826167, 0.811818, 0.799596, 0.789273, 0.780702, 0.773786, 0.768462, 0.764685, ...
0.762429, 0.761678, 0.762429, 0.764685, 0.768462, 0.773786, 0.780702, 0.789273, ...
0.777759, 0.766311, 0.756806, 0.749137, 0.743232, 0.739043, 0.736541, 0.735709, ...
0.736541, 0.739043, 0.743232, 0.749137, 0.756806, 0.766311, 0.777759, 0.791312, ...
0.807225, 0.825912, 0.848112, 0.875328, 0.911565, 0.983078, 0.983132, 0.983292, ...
0.983560, 0.983938, 0.984429, 0.985038, 0.985771, 0.986639, 0.987657, 0.988854, ...
0.990275, 0.992018, 0.993811, 0.993830, 0.993889, 0.993987, 0.994125, 0.994305, ...
0.994527, 0.994796, 0.995113, 0.995486, 0.995923, 0.996443, 0.997080, 0.997929, ];

    end
end
end
% END OF OPTIMISED SETTINGS

% Call constructor on all four strings
for curStr=0:3
    s_str( i_str_curStr ) = curStr;
    String_construct();
end

% DJH: set bow and finger variation properties
s_str( i_str_FbN_delta ) = ( 2 * pi ) / ( fs / s_str( i_str_FbN_rate ) );
s_str( i_str_FbN_step ) = s_str( i_str_FbN_delta );
s_str( i_str_vbN_delta ) = ( 2 * pi ) / ( fs / s_str( i_str_vbN_rate ) );
s_str( i_str_vbN_step ) = s_str( i_str_vbN_delta );
s_str( i_str_fpN_delta ) = ( 2 * pi ) / ( fs / s_str( i_str_fpN_rate ) );
s_str( i_str_fpN_step ) = s_str( i_str_fpN_delta );
% set vibrato properties
s_str( i_str_vib_step ) = ( 2 * pi ) / ( fs / s_str( i_str_vib_rate ) );

fprintf( '\n' );

```

String_constants.m

```
% *****
% STRING SIMULATION
%
% Name:   String_constants.m
% Date:   02/05/2016
% Author: Dave Humphreys, c1322278
% Desc:   Matlab implementation of Artifastring's
%         string constants, for use within Cello
%         Simulator. Initialises string-specific
%         simulation variables, and vector indices
%         and offsets.
%         String code ported from "Artifastring",
%         (c) 2013 G. Percival.
%
% *****

% Constant naming:
% i_cnst = index_constants
% o_cnst = offset_constants (to next string)
% s_cnst = string constants (all in one array)

% quantities
n_strings = 4;

% sizes
n_cnst_rn   = 128 - 1;           % 1 + 127 = 128 (next value)
n_cnst_Nmax = 96  - 1;           % allow space for biggest N

% indexes
i_cnst_T = 1;
i_cnst_L = 2;
i_cnst_d = 3;
i_cnst_pl = 4;
i_cnst_E = 5;
i_cnst_mu_s = 6;
i_cnst_mu_d = 7;
i_cnst_v0 = 8;
i_cnst_cutoff = 9;
i_cnst_N = 10;
```

```

i_cnst_rn = 11;
i_cnst_mlt = i_cnst_rn + n_cnst_rn + 1;
% added by DJH
i_cnst_lpf = i_cnst_mlt + 1;

% offsets
o_cnst = i_cnst_lpf + n_cnst_Nmax + 1;      % offset to start of _next_ string's values

% quick multipliers!
o_cnst_1 = 0;          % string 1 constants start
o_cnst_2 = o_cnst;      % string 2 constants start
o_cnst_3 = o_cnst * 2;  % string 3 constants start
o_cnst_4 = o_cnst * 3;  % string 4 constants start

% ZERO EVERYTHING
s_cnst = zeros( 1, o_cnst * n_strings );

%% from constants/string_cello.h

%% String 1 (C)
s_cnst( o_cnst_1 + i_cnst_T )      = 161.5;      % T
s_cnst( o_cnst_1 + i_cnst_L )      = 0.673;      % L
s_cnst( o_cnst_1 + i_cnst_d )      = 1.46e-03;    % d
s_cnst( o_cnst_1 + i_cnst_pl )     = 2.12e-02;    % pl
s_cnst( o_cnst_1 + i_cnst_E )      = 2.06e+10;    % E
s_cnst( o_cnst_1 + i_cnst_mu_s )   = 0.823;      % static friction
s_cnst( o_cnst_1 + i_cnst_mu_d )   = 0.427;      % dynamic friction
s_cnst( o_cnst_1 + i_cnst_v0 )     = 0.290;      % friction curve steepness
s_cnst( o_cnst_1 + i_cnst_cutoff ) = 0.06041;     % insignificant string vibrations
s_cnst( o_cnst_1 + i_cnst_N )      = 96;          % number of modes N
s_cnst( o_cnst_1 + i_cnst_rn : o_cnst_1 + i_cnst_rn + n_cnst_rn ) = ... % decays: these were calculated by mode-detect/all-
modes.py
    [ 3.959e-01, 2.221e+00, 2.834e+00, 2.681e+00, ...
      3.967e+00, 6.563e+00, 6.368e+00, 9.470e+00, ...
      1.234e+01, 1.531e+01, 1.004e+01, 1.744e+01, ...
      2.031e+01, 2.344e+01, 2.681e+01, 3.043e+01, ... %16
      3.430e+01, 3.842e+01, 4.279e+01, 4.742e+01, ...
      5.229e+01, 5.741e+01, 6.278e+01, 6.840e+01, ...
      7.427e+01, 8.039e+01, 8.676e+01, 9.338e+01, ...
      1.002e+02, 1.074e+02, 1.147e+02, 1.224e+02, ... % 32
      1.302e+02, 1.383e+02, 1.467e+02, 1.553e+02, ...

```

```

1.642e+02, 1.733e+02, 1.827e+02, 1.923e+02, ...
2.022e+02, 2.123e+02, 2.226e+02, 2.333e+02, ...
2.441e+02, 2.552e+02, 2.666e+02, 2.782e+02, ... % 48
2.901e+02, 3.022e+02, 3.146e+02, 3.272e+02, ...
3.400e+02, 3.532e+02, 3.665e+02, 3.801e+02, ...
3.940e+02, 4.081e+02, 4.225e+02, 4.371e+02, ...
4.520e+02, 4.671e+02, 4.824e+02, 4.980e+02, ... % 64
5.139e+02, 5.300e+02, 5.464e+02, 5.630e+02, ...
5.798e+02, 5.970e+02, 6.143e+02, 6.319e+02, ...
6.498e+02, 6.679e+02, 6.863e+02, 7.049e+02, ...
7.237e+02, 7.428e+02, 7.622e+02, 7.818e+02, ... % 80
8.017e+02, 8.218e+02, 8.421e+02, 8.627e+02, ...
8.836e+02, 9.047e+02, 9.261e+02, 9.477e+02, ...
9.695e+02, 9.916e+02, 1.014e+03, 1.037e+03, ...
1.059e+03, 1.083e+03, 1.106e+03, 1.130e+03, ... % 96
1.153e+03, 1.177e+03, 1.202e+03, 1.226e+03, ...
1.251e+03, 1.276e+03, 1.302e+03, 1.327e+03, ...
1.353e+03, 1.379e+03, 1.406e+03, 1.432e+03, ...
1.459e+03, 1.486e+03, 1.514e+03, 1.541e+03, ... % 112
1.569e+03, 1.597e+03, 1.625e+03, 1.654e+03, ...
1.683e+03, 1.712e+03, 1.741e+03, 1.771e+03, ...
1.801e+03, 1.831e+03, 1.861e+03, 1.892e+03, ...
1.923e+03, 1.954e+03, 1.985e+03, 2.017e+03 ]; % 128
s_cnst( o_cnst_1 + i_cnst_mlt ) = 2; % sample rate multiplier
% DJH - string damping curve
pre_curve = 0.3;
min_att = 0.5;
curve = ( logspace( min_att, 1, round( s_cnst( o_cnst_1 + i_cnst_N ) * ( 1 - pre_curve ) ) ) / 10 );
s_cnst( o_cnst_1 + i_cnst_lpf : o_cnst_1 + i_cnst_lpf + s_cnst( o_cnst_1 + i_cnst_N ) - 1 ) = [ ones( 1, s_cnst( o_cnst_1 +
i_cnst_N ) - round( s_cnst( o_cnst_1 + i_cnst_N ) * ( 1 - pre_curve ) ) ) curve( end : -1 : 1 ) ];

%% String 2 (G)
s_cnst( o_cnst_2 + i_cnst_T ) = 116.6; % T
s_cnst( o_cnst_2 + i_cnst_L ) = 0.669; % L
s_cnst( o_cnst_2 + i_cnst_d ) = 1.03e-03; % d
s_cnst( o_cnst_2 + i_cnst_pl ) = 6.85e-03; % pl
s_cnst( o_cnst_2 + i_cnst_E ) = 1.87e+10; % E
s_cnst( o_cnst_2 + i_cnst_mu_s ) = 0.836; % static friction
s_cnst( o_cnst_2 + i_cnst_mu_d ) = 0.415; % dynamic friction
s_cnst( o_cnst_2 + i_cnst_v0 ) = 0.279; % friction curve steepness
s_cnst( o_cnst_2 + i_cnst_cutoff ) = 0.002706; % insignificant string vibrations
s_cnst( o_cnst_2 + i_cnst_N ) = 96; % number of modes N
s_cnst( o_cnst_2 + i_cnst_rn : o_cnst_2 + i_cnst_rn + n_cnst_rn ) = ... % decays: these were calculated by mode-detect/all-modes.py

```

```

[ 3.421e-01, 1.051e+00, 1.521e+00, 2.009e+00, ...
  1.260e+00, 3.250e+00, 4.144e+00, 4.566e+00, ...
  5.290e+00, 6.565e+00, 8.002e+00, 8.420e+00, ...
  7.944e+00, 1.404e+01, 1.947e+01, 1.783e+01, ...
  2.020e+01, 2.272e+01, 2.539e+01, 2.821e+01, ...
  3.119e+01, 3.432e+01, 3.760e+01, 4.104e+01, ...
  4.463e+01, 4.837e+01, 5.226e+01, 5.630e+01, ...
  6.050e+01, 6.485e+01, 6.936e+01, 7.401e+01, ...
  7.882e+01, 8.379e+01, 8.890e+01, 9.417e+01, ...
  9.959e+01, 1.052e+02, 1.109e+02, 1.168e+02, ...
  1.228e+02, 1.290e+02, 1.353e+02, 1.418e+02, ...
  1.484e+02, 1.552e+02, 1.622e+02, 1.693e+02, ...
  1.765e+02, 1.839e+02, 1.915e+02, 1.992e+02, ...
  2.071e+02, 2.151e+02, 2.233e+02, 2.316e+02, ...
  2.400e+02, 2.487e+02, 2.574e+02, 2.664e+02, ...
  2.755e+02, 2.847e+02, 2.941e+02, 3.036e+02, ...
  3.133e+02, 3.232e+02, 3.332e+02, 3.433e+02, ...
  3.536e+02, 3.641e+02, 3.747e+02, 3.855e+02, ...
  3.964e+02, 4.074e+02, 4.187e+02, 4.300e+02, ...
  4.416e+02, 4.533e+02, 4.651e+02, 4.771e+02, ...
  4.892e+02, 5.015e+02, 5.139e+02, 5.265e+02, ...
  5.393e+02, 5.522e+02, 5.652e+02, 5.784e+02, ...
  5.918e+02, 6.053e+02, 6.190e+02, 6.328e+02, ...
  6.468e+02, 6.609e+02, 6.752e+02, 6.896e+02, ...
  7.042e+02, 7.189e+02, 7.338e+02, 7.488e+02, ...
  7.640e+02, 7.794e+02, 7.949e+02, 8.105e+02, ...
  8.263e+02, 8.423e+02, 8.584e+02, 8.746e+02, ...
  8.910e+02, 9.076e+02, 9.243e+02, 9.412e+02, ...
  9.582e+02, 9.754e+02, 9.927e+02, 1.010e+03, ...
  1.028e+03, 1.046e+03, 1.064e+03, 1.082e+03, ...
  1.100e+03, 1.118e+03, 1.137e+03, 1.156e+03, ...
  1.174e+03, 1.193e+03, 1.213e+03, 1.232e+03, ];
s_cnst( o_cnst_2 + i_cnst_mlt) = 2; % sample rate multiplier % TEST ONLY - was 1
% DJH - string damping curve
pre_curve = 0.3;
min_att = 0.5;
curve = ( logspace( min_att, 1, round( s_cnst( o_cnst_2 + i_cnst_N ) * ( 1 - pre_curve ) ) ) / 10 );
s_cnst( o_cnst_2 + i_cnst_lpf : o_cnst_2 + i_cnst_lpf + s_cnst( o_cnst_2 + i_cnst_N ) - 1 ) = [ ones( 1, s_cnst( o_cnst_2 +
i_cnst_N ) - round( s_cnst( o_cnst_2 + i_cnst_N ) * ( 1 - pre_curve ) ) ) curve( end : -1 : 1 ) ];

%% String 3 (D)
s_cnst( o_cnst_3 + i_cnst_T ) = 99.7; % T
s_cnst( o_cnst_3 + i_cnst_L ) = 0.660; % L

```



```

s_cnst( o_cnst_3 + i_cnst_d )      = 8.80e-04;      % d
s_cnst( o_cnst_3 + i_cnst_pl )     = 2.68e-03;      % pl
s_cnst( o_cnst_3 + i_cnst_E )      = 2.50e+10;      % E
s_cnst( o_cnst_3 + i_cnst_mu_s )   = 0.827;         % static friction
s_cnst( o_cnst_3 + i_cnst_mu_d )   = 0.423;         % dynamic friction
s_cnst( o_cnst_3 + i_cnst_v0 )     = 0.284;         % friction curve steepness
s_cnst( o_cnst_3 + i_cnst_cutoff ) = 0.002713;      % insignificant string vibrations
s_cnst( o_cnst_3 + i_cnst_N )      = 76;            % number of modes N
s_cnst( o_cnst_3 + i_cnst_rn : o_cnst_3 + i_cnst_rn + n_cnst_rn ) = ... % decays: these were calculated by mode-detect/all-
modes.py
[
    8.156e-01, 1.836e+00, 3.812e+00, 4.117e+00, ...
    4.889e+00, 5.872e+00, 8.164e+00, 1.280e+01, ...
    1.501e+01, 1.784e+01, 2.033e+01, 2.599e+01, ...
    3.013e+01, 3.504e+01, 4.034e+01, 4.604e+01, ...
    5.213e+01, 5.861e+01, 6.548e+01, 7.275e+01, ...
    8.041e+01, 8.847e+01, 9.691e+01, 1.058e+02, ...
    1.150e+02, 1.246e+02, 1.346e+02, 1.450e+02, ...
    1.558e+02, 1.670e+02, 1.786e+02, 1.906e+02, ...
    2.030e+02, 2.158e+02, 2.289e+02, 2.425e+02, ...
    2.564e+02, 2.708e+02, 2.855e+02, 3.006e+02, ...
    3.161e+02, 3.320e+02, 3.483e+02, 3.650e+02, ...
    3.821e+02, 3.996e+02, 4.175e+02, 4.358e+02, ...
    4.544e+02, 4.735e+02, 4.929e+02, 5.128e+02, ...
    5.330e+02, 5.536e+02, 5.746e+02, 5.961e+02, ...
    6.179e+02, 6.401e+02, 6.626e+02, 6.856e+02, ...
    7.090e+02, 7.328e+02, 7.569e+02, 7.815e+02, ...
    8.064e+02, 8.318e+02, 8.575e+02, 8.836e+02, ...
    9.102e+02, 9.371e+02, 9.644e+02, 9.921e+02, ...
    1.020e+03, 1.049e+03, 1.078e+03, 1.107e+03, ...
    1.136e+03, 1.166e+03, 1.197e+03, 1.228e+03, ...
    1.259e+03, 1.291e+03, 1.323e+03, 1.355e+03, ...
    1.388e+03, 1.421e+03, 1.455e+03, 1.489e+03, ...
    1.523e+03, 1.558e+03, 1.593e+03, 1.629e+03, ...
    1.664e+03, 1.701e+03, 1.738e+03, 1.775e+03, ...
    1.812e+03, 1.850e+03, 1.888e+03, 1.927e+03, ...
    1.966e+03, 2.006e+03, 2.046e+03, 2.086e+03, ...
    2.126e+03, 2.168e+03, 2.209e+03, 2.251e+03, ...
    2.293e+03, 2.336e+03, 2.379e+03, 2.422e+03, ...
    2.466e+03, 2.510e+03, 2.555e+03, 2.600e+03, ...
    2.645e+03, 2.691e+03, 2.737e+03, 2.784e+03, ...
    2.830e+03, 2.878e+03, 2.926e+03, 2.974e+03, ...
    3.022e+03, 3.071e+03, 3.120e+03, 3.170e+03, ];
s_cnst( o_cnst_3 + i_cnst_mlt ) = 2; % sample rate multiplier

```

```

% DJH - string damping curve
pre_curve = 0.3;
min_att    = 0.5;
curve = ( logspace( min_att, 1, round( s_cnst( o_cnst_3 + i_cnst_N ) * ( 1 - pre_curve ) ) ) / 10 );
s_cnst( o_cnst_3 + i_cnst_lpf : o_cnst_3 + i_cnst_lpf + s_cnst( o_cnst_3 + i_cnst_N ) - 1 ) = [ ones( 1,      s_cnst( o_cnst_3 +
i_cnst_N ) - round(      s_cnst( o_cnst_3 + i_cnst_N ) * ( 1 - pre_curve ) ) ) curve( end : -1 : 1 ) ];

%% String 4 (A)
s_cnst( o_cnst_4 + i_cnst_T )      = 139.6;          % T
s_cnst( o_cnst_4 + i_cnst_L )      = 0.658;          % l
s_cnst( o_cnst_4 + i_cnst_d )      = 7.47e-04;       % d
s_cnst( o_cnst_4 + i_cnst_pl )     = 1.67e-03;       % pl
s_cnst( o_cnst_4 + i_cnst_E )      = 2.50e+10;       % E
s_cnst( o_cnst_4 + i_cnst_mu_s )   = 0.873;          % static friction
s_cnst( o_cnst_4 + i_cnst_mu_d )   = 0.428;          % dynamic friction
s_cnst( o_cnst_4 + i_cnst_v0 )     = 0.275;          % friction curve steepness
s_cnst( o_cnst_4 + i_cnst_cutoff ) = 0.00235;        % insignificant string vibrations
s_cnst( o_cnst_4 + i_cnst_N )      = 92;             % number of modes N
s_cnst( o_cnst_4 + i_cnst_rn : o_cnst_4 + i_cnst_rn + n_cnst_rn ) = ... % decays: these were calculated by mode-detect/all-
modes.py
[
    2.481e+00, 2.487e+00, 4.710e+00, 3.964e+00, ...
    6.756e+00, 7.696e+00, 9.814e+00, 1.086e+01, ...
    1.302e+01, 1.572e+01, 2.005e+01, 1.987e+01, ...
    2.036e+01, 2.304e+01, 2.885e+01, 3.250e+01, ...
    3.641e+01, 4.057e+01, 4.498e+01, 4.965e+01, ...
    5.456e+01, 5.973e+01, 6.515e+01, 7.082e+01, ...
    7.675e+01, 8.293e+01, 8.936e+01, 9.604e+01, ...
    1.030e+02, 1.102e+02, 1.176e+02, 1.253e+02, ...
    1.332e+02, 1.414e+02, 1.499e+02, 1.586e+02, ...
    1.675e+02, 1.767e+02, 1.862e+02, 1.959e+02, ...
    2.058e+02, 2.160e+02, 2.265e+02, 2.372e+02, ...
    2.482e+02, 2.594e+02, 2.709e+02, 2.826e+02, ...
    2.946e+02, 3.068e+02, 3.193e+02, 3.320e+02, ...
    3.450e+02, 3.582e+02, 3.717e+02, 3.855e+02, ...
    3.995e+02, 4.137e+02, 4.282e+02, 4.430e+02, ...
    4.580e+02, 4.732e+02, 4.887e+02, 5.045e+02, ...
    5.205e+02, 5.368e+02, 5.533e+02, 5.700e+02, ...
    5.871e+02, 6.043e+02, 6.218e+02, 6.396e+02, ...
    6.576e+02, 6.759e+02, 6.945e+02, 7.132e+02, ...
    7.323e+02, 7.516e+02, 7.711e+02, 7.909e+02, ...
    8.109e+02, 8.312e+02, 8.518e+02, 8.726e+02, ...
    8.936e+02, 9.149e+02, 9.365e+02, 9.583e+02, ...
    9.804e+02, 1.003e+03, 1.025e+03, 1.048e+03, ...

```

```

1.071e+03, 1.094e+03, 1.118e+03, 1.142e+03, ...
1.166e+03, 1.190e+03, 1.215e+03, 1.240e+03, ...
1.265e+03, 1.290e+03, 1.316e+03, 1.342e+03, ...
1.368e+03, 1.394e+03, 1.421e+03, 1.447e+03, ...
1.475e+03, 1.502e+03, 1.529e+03, 1.557e+03, ...
1.585e+03, 1.614e+03, 1.642e+03, 1.671e+03, ...
1.700e+03, 1.730e+03, 1.759e+03, 1.789e+03, ...
1.819e+03, 1.850e+03, 1.880e+03, 1.911e+03, ...
1.942e+03, 1.974e+03, 2.006e+03, 2.037e+03, ];
s_cnst( o_cnst_4 + i_cnst_mlt ) = 2; % sample rate multiplier
% DJH - string damping curve
pre_curve = 0.3;
min_att = 0.5;
curve = ( logspace( min_att, 1, round( s_cnst( o_cnst_4 + i_cnst_N ) * ( 1 - pre_curve ) ) ) / 10 );
s_cnst( o_cnst_4 + i_cnst_lpf : o_cnst_4 + i_cnst_lpf + s_cnst( o_cnst_4 + i_cnst_N ) - 1 ) = [ ones( 1, s_cnst( o_cnst_4 +
i_cnst_N ) - round( s_cnst( o_cnst_4 + i_cnst_N ) * ( 1 - pre_curve ) ) ) curve( end : -1 : 1 ) ];

```

String_h.m

```
% *****
% STRING SIMULATION
%
% Name:   String_h.m
% Date:   02/05/2016
% Author: Dave Humphreys, c1322278
% Desc:   Creates the master string vector (and
%         defines indexes into it) for use within
%         Cello Simulator. Also sets variable sine
%         oscillator parameters (they may be tuned
%         here).
%         String code ported from "Artifastring",
%         (c) 2013 G. Percival.
%         Additional code added by D. Humphreys.
%
% *****

% enums
e_str_OFF      = 1;
e_str_BOW      = 2;
e_str_BOW_ACCEL = 3;
e_str_PLUCK    = 4;
e_str_RELEASE  = 5;

% sizes
n_str_inv_A = 9 - 1;          % matrix is 3x3

% indexes
i_str_fs = 1;                % Sample rate
i_str_dt = 2;                % Time delta
i_str_Fs = 3;                % String force
i_str_Fb = 4;                % Bow force
i_str_Fb_req = 5;            % Bow force requested by the user
i_str_va = 6;                % Bow acceleration, per dt
i_str_vb = 7;                % Bow velocity
i_str_vb_req = 8;            % Bow velocity requested by the user
i_str_vb_target = 9;         % Target bow velocity (used in acceleration)
i_str_div_pc_L = 10;
i_str_sqrt_two_div_L = 11;
i_str_I = 12;
```

```

i_str_X1 = 13; % Displacements
i_str_X2 = i_str_X1 + n_cnst_Nmax + 1; %
i_str_X3 = i_str_X2 + n_cnst_Nmax + 1; %
i_str_Y1 = i_str_X3 + n_cnst_Nmax + 1; % Velocity
i_str_Y2 = i_str_Y1 + n_cnst_Nmax + 1; %
i_str_Y3 = i_str_Y2 + n_cnst_Nmax + 1; %
i_str_G = i_str_Y3 + n_cnst_Nmax + 1; % Bridge
i_str_x0 = i_str_G + n_cnst_Nmax + 1;
i_str_x1 = i_str_x0 + 1;
i_str_x2 = i_str_x1 + 1;
i_str_K0 = i_str_x2 + 1; % "Extra actions"
i_str_K1 = i_str_K0 + 1; %
i_str_K2 = i_str_K1 + 1; %
i_str_R0 = i_str_K2 + 1; %
i_str_R1 = i_str_R0 + 1; %
i_str_R2 = i_str_R1 + 1; %
i_str_D1 = i_str_R2 + 1; % Bow
i_str_D2 = i_str_D1 + 1; %
i_str_D3 = i_str_D2 + 1; %
i_str_D4 = i_str_D3 + 1; %
i_str_D5 = i_str_D4 + 1; % Finger during bowing
i_str_D6 = i_str_D5 + 1; %
i_str_D7 = i_str_D6 + 1; %
i_str_D8 = i_str_D7 + 1; % Pluck release
i_str_D9 = i_str_D8 + 1; %
i_str_D10 = i_str_D9 + 1; %
i_str_D11 = i_str_D10 + 1; %
i_str_inside_phi = i_str_D11 + 1;
i_str_phix0 = i_str_inside_phi + n_cnst_Nmax + 1;
i_str_phix1 = i_str_phix0 + n_cnst_Nmax + 1;
i_str_phix2 = i_str_phix1 + n_cnst_Nmax + 1;
i_str_inv_A = i_str_phix2 + n_cnst_Nmax + 1;
i_str_n = i_str_inv_A + n_str_inv_A + 1;
i_str_slipstate = i_str_n + n_cnst_Nmax + 1; % bow slipping / sticking
i_str_finger_position = i_str_slipstate + 1;
i_str_finger_position_req = i_str_finger_position + 1;
i_str_Kf = i_str_finger_position_req + 1;
i_str_actions = i_str_Kf + 1;
i_str_bow_pluck_position = i_str_actions + 1;
i_str_pluck_samples_remaining = i_str_bow_pluck_position + 1;
i_str_y_pluck = i_str_pluck_samples_remaining + 1; % for pluck displacement
i_str_y_pluck_target = i_str_y_pluck + 1; %
i_str_tick_output_force = i_str_y_pluck_target + 1;

```

```

% engine constants
i_str_FINGER_WIDTH = i_str_tick_output_force + 1;
i_str_PLUCK_SECONDS = i_str_FINGER_WIDTH + 1;
i_str_PLUCK_WIDTH = i_str_PLUCK_SECONDS + 1;
i_str_PLUCK_VELOCITY = i_str_PLUCK_WIDTH + 1;
i_str_PLUCK_DISPLACEMENT = i_str_PLUCK_VELOCITY + 1;
i_str_K_FINGER = i_str_PLUCK_DISPLACEMENT + 1;
i_str_R_FINGER = i_str_K_FINGER + 1;
i_str_K_PLUCK = i_str_R_FINGER + 1;
i_str_R_PLUCK = i_str_K_PLUCK + 1;
% Variations (DJH)
% bow to string friction noise
i_str_A_noise = i_str_R_PLUCK + 1;
i_str_A_rate = i_str_A_noise + 1;
i_str_A_max_dev = i_str_A_rate + 1;
i_str_A_delta = i_str_A_max_dev + 1;
i_str_A_step = i_str_A_delta + 1;
i_str_A_random = i_str_A_step + 1;
i_str_A_state = i_str_A_random + 1;
i_str_No = i_str_A_state + 1;
%
i_str_recache = i_str_No + 1;
% string state arrays
i_str_a = i_str_recache + 1;
i_str_ad = i_str_a + n_cnst_Nmax + 1;
% modes represented in the string state arrays
i_str_w0 = i_str_ad + n_cnst_Nmax + 1;
% WORKING VARIABLES
i_str_ah = i_str_w0 + n_cnst_Nmax + 1;
i_str_adh = i_str_ah + n_cnst_Nmax + 1;
i_str_F0 = i_str_adh + n_cnst_Nmax + 1;
i_str_y0h = i_str_F0 + 1;
i_str_v0h = i_str_y0h + 1;
i_str_y1h = i_str_v0h + 1;
i_str_v1h = i_str_y1h + 1;
% bow velocity noise differences
i_str_vbN_diff = i_str_v1h + 1;
i_str_vbN_diff_prev = i_str_vbN_diff + 1;

% offsets
o_str = i_str_vbN_diff_prev + 1;

```

```

% SINGLE INSTANCE indexes (stored at end of vector)
% bow force noise
i_str_FbN_noise = o_str * n_strings + 1;
i_str_FbN_rate = i_str_FbN_noise + 1;
i_str_FbN_max_dev = i_str_FbN_rate + 1;
i_str_FbN_delta = i_str_FbN_max_dev + 1;
i_str_FbN_step = i_str_FbN_delta + 1;
i_str_FbN_random = i_str_FbN_step + 1;
i_str_FbN_state = i_str_FbN_random + 1;
i_str_FbN_amt = i_str_FbN_state + 1;
% bow velocity noise
i_str_vbN_noise = i_str_FbN_amt + 1;
i_str_vbN_rate = i_str_vbN_noise + 1;
i_str_vbN_max_dev = i_str_vbN_rate + 1;
i_str_vbN_delta = i_str_vbN_max_dev + 1;
i_str_vbN_step = i_str_vbN_delta + 1;
i_str_vbN_random = i_str_vbN_step + 1;
i_str_vbN_state = i_str_vbN_random + 1;
i_str_vbN_amt = i_str_vbN_state + 1;
i_str_vbN_diff = i_str_vbN_amt + 1;
i_str_vbN_diff_prev = i_str_vbN_diff + 1;
% finger force noise
i_str_fpN_noise = i_str_vbN_diff_prev + 1;
i_str_fpN_rate = i_str_fpN_noise + 1;
i_str_fpN_max_dev = i_str_fpN_rate + 1;
i_str_fpN_delta = i_str_fpN_max_dev + 1;
i_str_fpN_step = i_str_fpN_delta + 1;
i_str_fpN_random = i_str_fpN_step + 1;
i_str_fpN_state = i_str_fpN_random + 1;
i_str_fpN_amt = i_str_fpN_state + 1;
% finger position noise
i_str_fpN_noise = i_str_fpN_amt + 1;
i_str_fpN_rate = i_str_fpN_noise + 1;
i_str_fpN_max_dev = i_str_fpN_rate + 1;
i_str_fpN_delta = i_str_fpN_max_dev + 1;
i_str_fpN_step = i_str_fpN_delta + 1;
i_str_fpN_random = i_str_fpN_step + 1;
i_str_fpN_state = i_str_fpN_random + 1;
i_str_fpN_amt = i_str_fpN_state + 1;
% vibrato
i_str_vib_rate = i_str_fpN_amt + 1;
i_str_vib_step = i_str_vib_rate + 1;
i_str_vib_state = i_str_vib_step + 1;
i_str_vib_value = i_str_vib_state + 1;

```

```

i_str_vib_amt    = i_str_vib_value + 1;
%
i_str_curStr = i_str_vib_amt + 1;

% quick multipliers
o_str_1 = 0;           % string 1 variables start
o_str_2 = o_str;        % string 2 variables start
o_str_3 = o_str * 2;    % string 3 variables start
o_str_4 = o_str * 3;    % string 4 variables start

% zero everything
s_str = zeros( 1, o_str * n_strings + 1 );

%% Set initial variable states

% Set same values for all strings
for offset=o_str_1 : o_str : o_str_4
    s_str( offset + i_str_FINGER_WIDTH )      = 0.01;          % metres
    s_str( offset + i_str_PLUCK_SECONDS )      = 0.1;           % seconds
    s_str( offset + i_str_PLUCK_WIDTH )        = 0.012;         % metres
    s_str( offset + i_str_PLUCK_VELOCITY )     = 0.1;           % m/s
    s_str( offset + i_str_PLUCK_DISPLACEMENT ) = 0.005;         % metres
    s_str( offset + i_str_K_FINGER )           = hex2dec('1e5f');
    s_str( offset + i_str_R_FINGER )           = 30;
    s_str( offset + i_str_K_PLUCK )            = hex2dec('1e4f');
    s_str( offset + i_str_R_PLUCK )            = hex2dec('1e1f');
    % Variations (DJH)
    s_str( offset + i_str_A_noise )            = 0.3;           % amount of A_noise to apply (originally, simple A_noise = 0.02)
    s_str( offset + i_str_A_rate )             = 30;           % base rate of change, Hz
    s_str( offset + i_str_A_max_dev )          = 2;            % maximum deviation +/- as a product of rate (or step size)
    s_str( offset + i_str_A_delta )            = 0;            % base change per step, radians
    s_str( offset + i_str_A_step )             = 0;            % actual change per step, radians
    s_str( offset + i_str_A_random )           = 0.05;         % how much of rand()*A_delta to randomly alter A_step by each time
    s_str( offset + i_str_A_state )            = 0;            % current step's value, radians
    s_str( offset + i_str_No )                 = 0;
    %
    s_str( offset + i_str_vbN_diff_prev )      = 0;
end

% bow force noise
s_str( i_str_FbN_rate )          = 0.6;          % base rate of change, Hz
s_str( i_str_FbN_max_dev )       = 1;            % maximum deviation +/-, as a product of rate (or step size)
s_str( i_str_FbN_delta )         = 0;            % base change per step, radians

```



```

s_str( i_str_FbN_step )      = 0;          % actual change per step, radians
s_str( i_str_FbN_random )    = 0.1;        % how much of rand()*A_delta to randomly alter A_step by each time
s_str( i_str_FbN_state )     = 0;          % current step's value, radians
s_str( i_str_FbN_amt )       = 0.2;        % amount to vary bow force by, as a product of requested bow force
% bow velocity noise
s_str( i_str_vbN_rate )      = 0.4;        % base rate of change, Hz
s_str( i_str_vbN_max_dev )    = 1;         % maximum deviation +/-, as a product of rate (or step size)
s_str( i_str_vbN_delta )     = 0;         % base change per step, radians
s_str( i_str_vbN_step )      = 0;         % actual change per step, radians
s_str( i_str_vbN_random )    = 0.5;        % how much of rand()*A_delta to randomly alter A_step by each time
s_str( i_str_vbN_state )     = 0;         % current step's value, radians
s_str( i_str_vbN_amt )       = 0.2;        % amount to vary bow velocity by, as a product of requested bow velocity
% finger position noise
s_str( i_str_fpN_rate )      = 0.1;        % base rate of change, Hz
s_str( i_str_fpN_max_dev )    = 4;         % maximum deviation +/-, as a product of rate (or step size)
s_str( i_str_fpN_delta )     = 0;         % base change per step, radians
s_str( i_str_fpN_step )      = 0;         % actual change per step, radians
s_str( i_str_fpN_random )    = 2.7;        % how much of rand()*A_delta to randomly alter A_step by each time
s_str( i_str_fpN_state )     = 0;         % current step's value, radians
s_str( i_str_fpN_amt )       = 0.0015;    % amount to vary finger position by, as a product of string length
% vibrato
s_str( i_str_vib_rate )      = 3.0;        % rate of change, Hz
s_str( i_str_vib_step )      = 0;         % change per step, radians
s_str( i_str_vib_state )     = 0;         % current step's value, radians
s_str( i_str_vib_amt )       = 0.0045;    % amount to vary finger position by, as a product of string length

```

String_construct.m

```
% *****
% STRING SIMULATION
%
% Name:   String_construct.m
% Date:   02/05/2016
% Author: Dave Humphreys, c1322278
% Desc:   In-line version of a string's constructor,
%         for use within Cello Simulator.
%         Initialises string-specific simulation
%         variables.
%         String code ported from "Artifastring",
%         (c) 2013 G. Percival.
%
% *****

fprintf( 'Constructing string %d...\n', s_str( i_str_curStr ) );

e_str_OFF      = 1;
e_str_BOW      = 2;
e_str_BOW_ACCEL = 3;
e_str_PLUCK     = 4;
e_str_RELEASE  = 5;

% .. assuming fs set
curSOfst = o_str * s_str( i_str_curStr );
curScOfst = o_cnst * s_str( i_str_curStr );
s_str( curSOfst + i_str_fs ) = fs * s_cnst( curScOfst + i_cnst_mlt );
s_str( curSOfst + i_str_dt ) = 1 / s_str( curSOfst + i_str_fs );
% create a linspace vector
s_str( curSOfst + i_str_n : curSOfst + i_str_n + s_cnst( curScOfst + i_cnst_N ) - 1 ) = linspace( 1, s_cnst( curScOfst + i_cnst_N ),
s_cnst( curScOfst + i_cnst_N ) );
% DJH: set variation properties
s_str( curSOfst + i_str_A_delta ) = ( 2 * pi ) / ( s_str( curSOfst + i_str_fs ) / s_str( curSOfst + i_str_A_rate ) );
s_str( curSOfst + i_str_A_step ) = s_str( curSOfst + i_str_A_delta );

String_reset();
```

String_reset.m

```
% *****
% STRING SIMULATION
%
% Name:   String_reset.m
% Date:   02/05/2016
% Author: Dave Humphreys, c1322278
% Desc:   Matlab implementation of Artifastring's
%         reset() method, for use within Cello
%         Simulator. Initialises variables needed
%         by the string simulation code.
%         String code ported from "Artifastring",
%         (c) 2013 G. Percival.
%
% *****

% calculate offsets
curSOfst = o_str * s_str( i_str_curStr );
curScOfst = o_cnst * s_str( i_str_curStr );
%

String_cache_pc_c();

s_str( curSOfst + i_str_x0 ) = 0;
s_str( curSOfst + i_str_x1 ) = 0;
s_str( curSOfst + i_str_x2 ) = 0;
s_str( curSOfst + i_str_y_pluck ) = 0;
s_str( curSOfst + i_str_y_pluck_target ) = 0;
s_str( curSOfst + i_str_Fb_req ) = 0;
s_str( curSOfst + i_str_vb_req ) = 0;
s_str( curSOfst + i_str_vb ) = 0;
s_str( curSOfst + i_str_vb_target ) = 0;
s_str( curSOfst + i_str_va ) = 0;
s_str( curSOfst + i_str_pluck_samples_remaining ) = 0;
s_str( curSOfst + i_str_recache ) = 1; % DJH - recache now 0 or 1 (not false or true)

s_str( curSOfst + i_str_finger_position ) = 0;
s_str( curSOfst + i_str_finger_position_req ) = 0;
s_str( curSOfst + i_str_bow_pluck_position ) = 0;
s_str( curSOfst + i_str_Kf ) = s_str( curSOfst + i_str_K_FINGER );
```

```

s_str( curS0fst + i_str_K0 ) = 0;
s_str( curS0fst + i_str_R0 ) = 0;
s_str( curS0fst + i_str_K2 ) = 0;
s_str( curS0fst + i_str_actions ) = e_str_OFF;
s_str( curS0fst + i_str_slipstate ) = 0;

s_str( curS0fst + i_str_a : curS0fst + i_str_a + s_cnst( curSc0fst + i_cnst_N ) - 1 ) = zeros( 1, s_cnst( curSc0fst + i_cnst_N ) );
s_str( curS0fst + i_str_ad : curS0fst + i_str_ad + s_cnst( curSc0fst + i_cnst_N ) - 1 ) = zeros( 1, s_cnst( curSc0fst + i_cnst_N ) );

```

String_cache_pc.c.m

```
% *****
% STRING SIMULATION
%
% Name:   String_cache_pc.c.m
% Date:   02/05/2016
% Author: Dave Humphreys, c1322278
% Desc:   Matlab implementation of Artifastring's
%         cache_pc_c() method, for use within
%         Cello Simulator. Initialises needed
%         simulation variables.
%         String code ported from "Artifastring",
%         (c) 2013 G. Percival.
%
% *****

% calculate offsets (needed? this is always called from code already setting i_str_curStr?)
curSOfst = o_str * s_str( i_str_curStr );
curScOfst = o_cnst * s_str( i_str_curStr );
%

s_str( curSOfst + i_str_div_pc_L )      = 1 / s_cnst( curScOfst + i_cnst_L );
s_str( curSOfst + i_str_sqrt_two_div_L ) = sqrt( 2 / s_cnst( curScOfst + i_cnst_L ) );
s_str( curSOfst + i_str_I )             = pi * s_cnst( curScOfst + i_cnst_d )^4 / 64;

c_ones = ones( 1, s_cnst( curScOfst + i_cnst_N ) );

s_str( curSOfst + i_str_w0 : curSOfst + i_str_w0 + s_cnst( curScOfst + i_cnst_N ) - 1 ) = ... % generate vector of modes in self.a
    sqrt( ( s_cnst( curScOfst + i_cnst_T ) / s_cnst( curScOfst + i_cnst_pl ) ) * ...
        ( s_str( curSOfst + i_str_n : curSOfst + i_str_n + s_cnst( curScOfst + i_cnst_N ) - 1 ) ...
        * pi * s_str( curSOfst + i_str_div_pc_L ) ).^2 + ...
        ( s_cnst( curScOfst + i_cnst_E ) * s_str( curSOfst + i_str_I ) / s_cnst( curScOfst + i_cnst_pl ) ) * ...
        ( s_str( curSOfst + i_str_n : curSOfst + i_str_n + s_cnst( curScOfst + i_cnst_N ) - 1 ) ...
        * pi * s_str( curSOfst + i_str_div_pc_L ) ).^4 );

highest_freq = s_str( curSOfst + i_str_w0 + ( s_cnst( curScOfst + i_cnst_N ) - 1 ) ) / (2*pi);
if( highest_freq > s_str( curSOfst + i_str_fs ) / 2 )
    fprintf( '%d: BAD FREQ! Highest freq: %f, Nyquist freq: %f\n', s_str( i_str_curStr ) + 1, ...
            highest_freq, s_str( curSOfst + i_str_fs ) / 2 );
end
```

```

w = sqrt( s_str( curSOfst + i_str_w0 : curSOfst + i_str_w0 + s_cnst( curScOfst + i_cnst_N ) - 1 ).^2 ...
- s_cnst( curScOfst + i_cnst_rn : curScOfst + i_cnst_rn + s_cnst( curScOfst + i_cnst_N ) - 1 ).^2 );
s_str( curSOfst + i_str_X1 : curSOfst + i_str_X1+ s_cnst( curScOfst + i_cnst_N ) - 1 ) = ...
( cos( w * s_str( curSOfst + i_str_dt ) ) ...
+ ( s_cnst( curScOfst + i_cnst_rn : curScOfst + i_cnst_rn + s_cnst( curScOfst + i_cnst_N ) - 1 ) / w ) .* ...
sin( w * s_str( curSOfst + i_str_dt ) ) ) .* ...
exp( -s_cnst( curScOfst + i_cnst_rn : curScOfst + i_cnst_rn + s_cnst( curScOfst + i_cnst_N ) - 1 ) ...
* s_str( curSOfst + i_str_dt ) );
s_str( curSOfst + i_str_X2 : curSOfst + i_str_X2+ s_cnst( curScOfst + i_cnst_N ) - 1 ) = ...
( ( c_ones ./ w ) .* sin( w * s_str( curSOfst + i_str_dt ) ) ) .* ...
exp( -s_cnst( curScOfst + i_cnst_rn : curScOfst + i_cnst_rn + s_cnst( curScOfst + i_cnst_N ) - 1 ) * s_str( curSOfst +
i_str_dt ) );
s_str( curSOfst + i_str_X3 : curSOfst + i_str_X3+ s_cnst( curScOfst + i_cnst_N ) - 1 ) = ...
( c_ones - s_str( curSOfst + i_str_X1 : curSOfst + i_str_X1 + s_cnst( curScOfst + i_cnst_N ) - 1 ) ) ./ ...
( s_cnst( curScOfst + i_cnst_pl ) * ...
s_str( curSOfst + i_str_w0 : curSOfst + i_str_w0 + s_cnst( curScOfst + i_cnst_N ) - 1 ).^2 );

s_str( curSOfst + i_str_Y1 : curSOfst + i_str_Y1+ s_cnst( curScOfst + i_cnst_N ) - 1 ) = ...
- ( w + s_cnst( curScOfst + i_cnst_rn : curScOfst + i_cnst_rn + s_cnst( curScOfst + i_cnst_N ) - 1 ).^2 ./ w ) .* ...
sin( w * s_str( curSOfst + i_str_dt ) ) .* ...
exp( -s_cnst( curScOfst + i_cnst_rn : curScOfst + i_cnst_rn + s_cnst( curScOfst + i_cnst_N ) - 1 ) * s_str( curSOfst +
i_str_dt ) );
s_str( curSOfst + i_str_Y2 : curSOfst + i_str_Y2+ s_cnst( curScOfst + i_cnst_N ) - 1 ) = ...
( cos( w * s_str( curSOfst + i_str_dt ) ) - ...
( s_cnst( curScOfst + i_cnst_rn : curScOfst + i_cnst_rn + s_cnst( curScOfst + i_cnst_N ) - 1 ) / w ) .* ...
sin( w * s_str( curSOfst + i_str_dt ) ) ) .* ...
exp( -s_cnst( curScOfst + i_cnst_rn : curScOfst + i_cnst_rn + s_cnst( curScOfst + i_cnst_N ) - 1 ) * s_str( curSOfst +
i_str_dt ) );
s_str( curSOfst + i_str_Y3 : curSOfst + i_str_Y3+ s_cnst( curScOfst + i_cnst_N ) - 1 ) = ...
-s_str( curSOfst + i_str_Y1 : curSOfst + i_str_Y1 + s_cnst( curScOfst + i_cnst_N ) - 1 ) ./ ...
( s_cnst( curScOfst + i_cnst_pl ) * ...
( s_str( curSOfst + i_str_w0 : curSOfst + i_str_w0 + s_cnst( curScOfst + i_cnst_N ) - 1 ).^2 ) );
s_str( curSOfst + i_str_G : curSOfst + i_str_G + s_cnst( curScOfst + i_cnst_N ) - 1 ) = ...
s_str( curSOfst + i_str_sqrt_two_div_L ) * ( s_cnst( curScOfst + i_cnst_T ) * ...
( s_str( curSOfst + i_str_n : curSOfst + i_str_n + s_cnst( curScOfst + i_cnst_N ) - 1 ) ...
* pi * s_str( curSOfst + i_str_div_pc_L ) ) + ...
s_cnst( curScOfst + i_cnst_E ) * s_str( curSOfst + i_str_I ) * ...
( s_str( curSOfst + i_str_n : curSOfst + i_str_n + s_cnst( curScOfst + i_cnst_N ) - 1 ) ...
* pi * s_str( curSOfst + i_str_div_pc_L ) ).^3 );

s_str( curSOfst + i_str_inside_phi : curSOfst + i_str_inside_phi + s_cnst( curScOfst + i_cnst_N ) - 1 ) = ...
s_str( curSOfst + i_str_n : curSOfst + i_str_n + s_cnst( curScOfst + i_cnst_N ) - 1 ) ...
* pi * s_str( curSOfst + i_str_div_pc_L );

```

String_bow.m

```
% *****
% STRING SIMULATION
%
% Name:   String_bow.m
% Date:   02/05/2016
% Author: Dave Humphreys, c1322278
% Desc:   Matlab implementation of Artifastring's
%         bow() method, for use within Cello
%         Simulator. Set bow_ratio_from_bridge,
%         bow_force and bow_velocity before
%         calling.
%         String code ported from "Artifastring",
%         (c) 2013 G. Percival.
% *****

% calculate offsets
curS0fst = o_str * s_str( i_str_curStr );
curSc0fst = o_cnst * s_str( i_str_curStr );
%

fprintf( 'bow(): ratio_from_bridge = %f; force = %f; velocity = %f\n', bow_ratio_from_bridge, bow_force, bow_velocity );
if( bow_force == 0 ) % we're being asked to stop bowing
    % from string_release()
    s_str( curS0fst + i_str_actions ) = e_str_RELEASE;
    % below necessary? "can't use lazy evaluation because not guaranteed to fill_buffer on a 1024-sample boundary"
    String_cache_pa_c();
else
    s_str( curS0fst + i_str_Fb_req ) = bow_force;
    s_str( curS0fst + i_str_vb_req ) = bow_velocity;
    s_str( curS0fst + i_str_vb ) = bow_velocity;
    bow_distance_from_bridge = s_cnst( curSc0fst + i_cnst_L ) * bow_ratio_from_bridge;
    if( abs( s_str( curS0fst + i_str_bow_pluck_position ) - bow_distance_from_bridge ) >= 1e-10 || ...
        s_str( curS0fst + i_str_actions ) ~= e_str_BOW )
        s_str( curS0fst + i_str_bow_pluck_position ) = bow_distance_from_bridge;
        s_str( curS0fst + i_str_recache ) = 1;
    end
    s_str( curS0fst + i_str_actions ) = e_str_BOW;
end
```

String bow accel.m

```
% *****
% STRING SIMULATION
%
% Name:   String_bow_accel.m
% Date:   02/05/2016
% Author: Dave Humphreys, c1322278
% Desc:   Matlab implementation of Artifastring's
%         bow_accel() method, for use within Cello
%         Simulator. Set bow_ratio_from_bridge,
%         bow_force and bow_velocity, bow_accel
%         and bow_velocity_target before calling.
%         String code ported from "Artifastring",
%         (c) 2013 G. Percival.
%
% *****

% calculate offsets (needed? this is always called from code already setting i_str_curStr?)
curS0fst = o_str * s_str( i_str_curStr );
curSc0fst = o_cnst * s_str( i_str_curStr );
%

s_str( curS0fst + i_str_actions ) = e_str_BOW_ACCEL;
s_str( curS0fst + i_str_Fs ) = bow_force;
s_str( curS0fst + i_str_va ) = bow_accel * s_str( curS0fst + i_str_dt );
s_str( curS0fst + i_str_vb_target ) = bow_velocity_target;
bow_distance_from_bridge = s_cnst( curSc0fst + i_cnst_L ) * bow_ratio_from_bridge;

if( abs( s_str( curS0fst + i_str_bow_pluck_position ) - bow_distance_from_bridge ) >= 1e-10 )
    s_str( curS0fst + i_str_bow_pluck_position ) = bow_distance_from_bridge;
    s_str( curS0fst + i_str_recache ) = 1;
end
```


String_finger.m

```
% *****
% STRING SIMULATION
%
% Name:   String_finger.m
% Date:   02/05/2016
% Author: Dave Humphreys, c1322278
% Desc:   Matlab implementation of Artifastring's
%         finger() method, for use within Cello
%         Simulator. Set bow_ratio_from_nut and
%         Kf_get before calling.
%         String code ported from "Artifastring",
%         (c) 2013 G. Percival.
%
% *****

% calculate offsets (needed? this is always called from code already setting i_str_curStr?)
curS0fst = o_str * s_str( i_str_curStr );
curSc0fst = o_cnst * s_str( i_str_curStr );
%

% C assert replacement!
if( ratio_from_nut < 0 )
    ratio_from_nut = 0;
elseif( ratio_from_nut > 1 )
    ratio_from_nut = 1;
end

if( ratio_from_nut == 0 )
    s_str( curS0fst + i_str_finger_position ) = 0;
    s_str( curS0fst + i_str_finger_position_req ) = 0;
else
    s_str( curS0fst + i_str_finger_position_req ) = s_cnst( curSc0fst + i_cnst_L ) * ( 1 - ratio_from_nut );
end

s_str( curS0fst + i_str_Kf ) = Kf_get * s_str( curS0fst + i_str_K_FINGER );
s_str( curS0fst + i_str_recache ) = 1;
```

String_pluck.m

```
% *****
% STRING SIMULATION
%
% Name:   String_pluck.m
% Date:   02/05/2016
% Author: Dave Humphreys, c1322278
% Desc:   Matlab implementation of Artifastring's
%         pluck() method, for use within Cello
%         Simulator. Set ratio_from_bridge and
%         pull_distance before calling.
%         String code ported from "Artifastring",
%         (c) 2013 G. Percival.
%
% *****

% calculate offsets
curS0fst = o_str * s_str( i_str_curStr );
curSc0fst = o_cnst * s_str( i_str_curStr );
%

% C assert replacement
if( ratio_from_bridge < 0 )
    ratio_from_bridge = 0;
elseif( ratio_from_bridge > 1 )
    ratio_from_bridge = 1;
end

s_str( curS0fst + i_str_bow_pluck_position ) = s_cnst( curSc0fst + i_cnst_L ) * ratio_from_bridge;
s_str( curS0fst + i_str_actions ) = e_str_PLUCK;
s_str( curS0fst + i_str_pluck_samples_remaining ) = ...
    s_str( curS0fst + i_str_PLUCK_SECONDS ) * s_str( curS0fst + i_str_fs );
s_str( curS0fst + i_str_y_pluck ) = 0;
s_str( curS0fst + i_str_y_pluck_target ) = pull_distance * s_str( curS0fst + i_str_PLUCK_DISPLACEMENT );

s_str( curS0fst + i_str_recache ) = 1;

% from generate_output()
if( s_str( curS0fst + i_str_pluck_samples_remaining ) > 0 )
    % from update_pluck_actions()
    s_str( curS0fst + i_str_pluck_samples_remaining ) = s_str( curS0fst + i_str_pluck_samples_remaining ) - 1;
```

```

if( s_str( curSOfst + i_str_y_pluck ) < s_str( curSOfst + i_str_y_pluck_target ) )
    s_str( curSOfst + i_str_y_pluck ) = s_str( curSOfst + i_str_y_pluck ) + ...
        ( s_str( curSOfst + i_str_PLUCK_VELOCITY ) * s_str( curSOfst + i_str_dt ) );
end
if( s_str( curSOfst + i_str_pluck_samples_remaining ) == 0 )
    % from string_release()
    s_str( curSOfst + i_str_actions ) = e_str_RELEASE;
    s_str( curSOfst + i_str_y_pluck ) = 0;
    % below necessary? "can't use lazy evaluation because not guaranteed to fill_buffer on a 1024-sample boundary"
    String_cache_pa_c();
end
end

```

String_release.m

```
% *****
% STRING SIMULATION
%
% Name:   String_release.m
% Date:   02/05/2016
% Author: Dave Humphreys, c1322278
% Desc:   Matlab implementation of Artifastring's
%         release() method, for use within Cello
%         Simulator.
%         String code ported from "Artifastring",
%         (c) 2013 G. Percival.
% *****

% calculate offsets
curSOfst = o_str * s_str( i_str_curStr );
curScOfst = o_cnst * s_str( i_str_curStr );
%

s_str( curSOfst + i_str_actions ) = e_str_RELEASE;
s_str( curSOfst + i_str_y_pluck ) = 0;

% Below from cache_pa_c()
s_str( curSOfst + i_str_phix0 : curSOfst + i_str_phix0 + s_cnst( curScOfst + i_cnst_N ) - 1 ) = ...
    s_str( curSOfst + i_str_sqrt_two_div_L ) * sin( s_str( curSOfst + i_str_x0 ) ...
    * s_str( curSOfst + i_str_inside_phi : curSOfst + i_str_inside_phi + s_cnst( curScOfst + i_cnst_N ) - 1 ) );
s_str( curSOfst + i_str_phix1 : curSOfst + i_str_phix1 + s_cnst( curScOfst + i_cnst_N ) - 1 ) = ...
    s_str( curSOfst + i_str_sqrt_two_div_L ) * sin( s_str( curSOfst + i_str_x1 ) ...
    * s_str( curSOfst + i_str_inside_phi : curSOfst + i_str_inside_phi + s_cnst( curScOfst + i_cnst_N ) - 1 ) );
A00 = sum( s_str( curSOfst + i_str_X3 : curSOfst + i_str_X3 + s_cnst( curScOfst + i_cnst_N ) - 1 ) .* ...
    s_str( curSOfst + i_str_phix0 : curSOfst + i_str_phix0 + s_cnst( curScOfst + i_cnst_N ) - 1 ) .* ...
    s_str( curSOfst + i_str_phix0 : curSOfst + i_str_phix0 + s_cnst( curScOfst + i_cnst_N ) - 1 ) );
A01 = sum( s_str( curSOfst + i_str_X3 : curSOfst + i_str_X3 + s_cnst( curScOfst + i_cnst_N ) - 1 ) .* ...
    s_str( curSOfst + i_str_phix0 : curSOfst + i_str_phix0 + s_cnst( curScOfst + i_cnst_N ) - 1 ) .* ...
    s_str( curSOfst + i_str_phix1 : curSOfst + i_str_phix1 + s_cnst( curScOfst + i_cnst_N ) - 1 ) );
A11 = sum( s_str( curSOfst + i_str_X3 : curSOfst + i_str_X3 + s_cnst( curScOfst + i_cnst_N ) - 1 ) .* ...
    s_str( curSOfst + i_str_phix1 : curSOfst + i_str_phix1 + s_cnst( curScOfst + i_cnst_N ) - 1 ) .* ...
    s_str( curSOfst + i_str_phix1 : curSOfst + i_str_phix1 + s_cnst( curScOfst + i_cnst_N ) - 1 ) );
```

```

B00 = sum( s_str( curS0fst + i_str_Y3 : curS0fst + i_str_Y3 + s_cnst( curSc0fst + i_cnst_N ) - 1 ) .* ...
    s_str( curS0fst + i_str_phix0 : curS0fst + i_str_phix0 + s_cnst( curSc0fst + i_cnst_N ) - 1 ) .* ...
    s_str( curS0fst + i_str_phix0 : curS0fst + i_str_phix0 + s_cnst( curSc0fst + i_cnst_N ) - 1 ) );
B01 = sum( s_str( curS0fst + i_str_Y3 : curS0fst + i_str_Y3 + s_cnst( curSc0fst + i_cnst_N ) - 1 ) .* ...
    s_str( curS0fst + i_str_phix0 : curS0fst + i_str_phix0 + s_cnst( curSc0fst + i_cnst_N ) - 1 ) .* ...
    s_str( curS0fst + i_str_phix1 : curS0fst + i_str_phix1 + s_cnst( curSc0fst + i_cnst_N ) - 1 ) );
B11 = sum( s_str( curS0fst + i_str_Y3 : curS0fst + i_str_Y3 + s_cnst( curSc0fst + i_cnst_N ) - 1 ) .* ...
    s_str( curS0fst + i_str_phix1 : curS0fst + i_str_phix1 + s_cnst( curSc0fst + i_cnst_N ) - 1 ) .* ...
    s_str( curS0fst + i_str_phix1 : curS0fst + i_str_phix1 + s_cnst( curSc0fst + i_cnst_N ) - 1 ) );

if( s_str( curS0fst + i_str_actions ) == e_str_BOW )
    % bow coefficients
    L1 = 1 / ( ( B00*B11 - B01*B01 ) * s_str( curS0fst + i_str_R1 ) + ...
        ( A11*B00 - A01*B01 ) * s_str( curS0fst + i_str_K1 ) + B00 );
    s_str( curS0fst + i_str_D1 ) = ( B11 * s_str( curS0fst + i_str_R1 ) + ...
        A11 * s_str( curS0fst + i_str_K1 ) + 1 ) * L1;
    s_str( curS0fst + i_str_D4 ) = 0.5 / s_str( curS0fst + i_str_D1 );
    s_str( curS0fst + i_str_D2 ) = B01 * s_str( curS0fst + i_str_K1 ) * L1;
    s_str( curS0fst + i_str_D3 ) = B01 * s_str( curS0fst + i_str_R1 ) * L1;
    % finger-during-bowing coefficients
    L2 = -1 / ( B11 * s_str( curS0fst + i_str_R1 ) + A11 * s_str( curS0fst + i_str_K1 ) + 1 );
    s_str( curS0fst + i_str_D5 ) = ( B01 * s_str( curS0fst + i_str_R1 ) + ...
        A01 * s_str( curS0fst + i_str_K1 ) ) * L2;
    s_str( curS0fst + i_str_D6 ) = s_str( curS0fst + i_str_R1 ) * L2;
    s_str( curS0fst + i_str_D7 ) = s_str( curS0fst + i_str_K1 ) * L2;
end

if( s_str( curS0fst + i_str_actions ) == e_str_PLUCK )
    s_str( curS0fst + i_str_phix2 : curS0fst + i_str_phix2 + s_cnst( curSc0fst + i_cnst_N ) - 1 ) = ...
        s_str( curS0fst + i_str_sqrt_two_div_L ) .* sin( s_str( curS0fst + i_str_x2 ) ...
            * s_str( curS0fst + i_str_inside_phi : curS0fst + i_str_inside_phi + s_cnst( curSc0fst + i_cnst_N ) - 1 ) );
    A02 = sum( s_str( curS0fst + i_str_X3 : curS0fst + i_str_X3 + s_cnst( curSc0fst + i_cnst_N ) - 1 ) .* ...
        s_str( curS0fst + i_str_phix0 : curS0fst + i_str_phix0 + s_cnst( curSc0fst + i_cnst_N ) - 1 ) .* ...
        s_str( curS0fst + i_str_phix2 : curS0fst + i_str_phix2 + s_cnst( curSc0fst + i_cnst_N ) - 1 ) );
    A12 = sum( s_str( curS0fst + i_str_X3 : curS0fst + i_str_X3 + s_cnst( curSc0fst + i_cnst_N ) - 1 ) .* ...
        s_str( curS0fst + i_str_phix1 : curS0fst + i_str_phix1 + s_cnst( curSc0fst + i_cnst_N ) - 1 ) .* ...
        s_str( curS0fst + i_str_phix2 : curS0fst + i_str_phix2 + s_cnst( curSc0fst + i_cnst_N ) - 1 ) );
    A22 = sum( s_str( curS0fst + i_str_X3 : curS0fst + i_str_X3 + s_cnst( curSc0fst + i_cnst_N ) - 1 ) .* ...
        s_str( curS0fst + i_str_phix2 : curS0fst + i_str_phix2 + s_cnst( curSc0fst + i_cnst_N ) - 1 ) .* ...
        s_str( curS0fst + i_str_phix2 : curS0fst + i_str_phix2 + s_cnst( curSc0fst + i_cnst_N ) - 1 ) );
    B02 = sum( s_str( curS0fst + i_str_Y3 : curS0fst + i_str_Y3 + s_cnst( curSc0fst + i_cnst_N ) - 1 ) .* ...
        s_str( curS0fst + i_str_phix0 : curS0fst + i_str_phix0 + s_cnst( curSc0fst + i_cnst_N ) - 1 ) .* ...
        s_str( curS0fst + i_str_phix2 : curS0fst + i_str_phix2 + s_cnst( curSc0fst + i_cnst_N ) - 1 ) );

```

```

B12 = sum( s_str( curS0fst + i_str_Y3 : curS0fst + i_str_Y3 + s_cnst( curSc0fst + i_cnst_N ) - 1 ) .* ...
           s_str( curS0fst + i_str_phix1 : curS0fst + i_str_phix1 + s_cnst( curSc0fst + i_cnst_N ) - 1 ) .* ...
           s_str( curS0fst + i_str_phix2 : curS0fst + i_str_phix2 + s_cnst( curSc0fst + i_cnst_N ) - 1 ) );
B22 = sum( s_str( curS0fst + i_str_Y3 : curS0fst + i_str_Y3 + s_cnst( curSc0fst + i_cnst_N ) - 1 ) .* ...
           s_str( curS0fst + i_str_phix2 : curS0fst + i_str_phix2 + s_cnst( curSc0fst + i_cnst_N ) - 1 ) .* ...
           s_str( curS0fst + i_str_phix2 : curS0fst + i_str_phix2 + s_cnst( curSc0fst + i_cnst_N ) - 1 ) );

matrix_A = [ B00 * s_str( curS0fst + i_str_R0 ) + A00 * s_str( curS0fst + i_str_K0 ) + 1, ...
             B01 * s_str( curS0fst + i_str_R0 ) + A01 * s_str( curS0fst + i_str_K0 ), ...
             B02 * s_str( curS0fst + i_str_R0 ) + A02 * s_str( curS0fst + i_str_K0 ); ...

             B01 * s_str( curS0fst + i_str_R1 ) + A01 * s_str( curS0fst + i_str_K1 ), ...
             B11 * s_str( curS0fst + i_str_R1 ) + A11 * s_str( curS0fst + i_str_K1 ) + 1, ...
             B12 * s_str( curS0fst + i_str_R1 ) + A12 * s_str( curS0fst + i_str_K1 ); ...

             B02 * s_str( curS0fst + i_str_R2 ) + A02 * s_str( curS0fst + i_str_K2 ), ...
             B12 * s_str( curS0fst + i_str_R2 ) + A12 * s_str( curS0fst + i_str_K2 ), ...
             B22 * s_str( curS0fst + i_str_R2 ) + A22 * s_str( curS0fst + i_str_K2 ) + 1; ];
inv_A = inv( matrix_A );
s_str( curS0fst + i_str_inv_A : curS0fst + i_str_inv_A + n_str_inv_A ) = [ inv_A(1,:) inv_A(2,:) inv_A(3,:) ];
end

if( s_str( curS0fst + i_str_actions ) == e_str_RELEASE )
M00 = A00 * s_str( curS0fst + i_str_K1 ) + B00 * s_str( curS0fst + i_str_R1 ) + 1;
M01 = A01 * s_str( curS0fst + i_str_K1 ) + B01 * s_str( curS0fst + i_str_R1 );
M11 = A11 * s_str( curS0fst + i_str_K1 ) + B11 * s_str( curS0fst + i_str_R1 ) + 1;

L3 = -1 / ( M00 * M11 - M01 * M01 );
s_str( curS0fst + i_str_D8 ) = -1 / M00;
s_str( curS0fst + i_str_D9 ) = M01 * s_str( curS0fst + i_str_D8 );
s_str( curS0fst + i_str_D10 ) = -M01 * L3;
s_str( curS0fst + i_str_D11 ) = M00 * L3;

% "extra inv_A for any remaining tick_pluck() which occurs before a new buffer is called (i.e. the switch/case in fill_buffer)"
matrix_A = [ B00 * s_str( curS0fst + i_str_R0 ) + A00 * s_str( curS0fst + i_str_K0 ) + 1, ...
             B01 * s_str( curS0fst + i_str_R0 ) + A01 * s_str( curS0fst + i_str_K0 ), ...
             0; ...

             B01 * s_str( curS0fst + i_str_R1 ) + A01 * s_str( curS0fst + i_str_K1 ), ...
             B11 * s_str( curS0fst + i_str_R1 ) + A11 * s_str( curS0fst + i_str_K1 ) + 1, ...
             0; ...

```

```

        0, ...
        0, ...
        1; ];
    inv_A = inv( matrix_A );
    s_str( curS0fst + i_str_inv_A : curS0fst + i_str_inv_A + n_str_inv_A ) = [ inv_A(1,:) inv_A(2,:) inv_A(3,:) ];
end

% now that the recache is done, reset the flag
s_str( curS0fst + i_str_recache ) = 0;

```

Generate inputSim.m

```
% *****
% GENERATE RAW STRING SAMPLES
%
% Name:    Generate_inputSim.m
% Date:    02/05/2016
% Author:  Dave Humphreys, c1322278
% Desc:    Standalone script which generates a set
%           of raw string outputs, for use with
%           CelloSim.m (convolution vectors are
%           calculated from these and the target
%           real cello sounds). Requires the
%           CelloSim scripts.
%
% *****

% Clear any existing data from the workspace
clear;
% Load simulation variables
CelloSim_h();
% override simulation time
sim_time = 3;
% disable the stopping of bowing
stop_bowing = false;
% deactivate all feedback (to obtain raw strings)
str_gain    = 0;
body_gain   = 0;
% disable 'realism' features
bow_noise_active = false;
vibrato_active   = false;
vary_bow_force   = false;
vary_bow_velocity = false;
vary_finger_pos   = false;

% Allow level change to alter bowing velocity
velo_floor = 2;      % floor from which to begin velocity alteration

fprintf( 'Generating inputSim files:\n' );
```



```

for active_string=0:num_strings - 1
    for gen_n=1:num_files
        active_pos = 0;
        if( gen_n == 2 )
            active_pos = 4;
        elseif( gen_n == 3 )
            active_pos = 7;
        elseif( gen_n == 4 )
            active_pos = 10;
        end

        for gen_l=1:num_levels
            fprintf( '\n>> Generating sim_%d_%d_%d.wav... <<\n', active_string + 1, gen_n, gen_l );

            %% Setup and Run Simulation
            % Initialise all strings
            Strings_init(); % indexed 0-3
            % Initialise convolution vector extractor
            IR_construct(); % indexed 1-4, for now
            % Initialise body convolution code
            Convolve_construct();
            % Initialise FFT velocity feedback code
            FFTv_construct();

            % Set initial state
            CelloSim_initial_state();

            % Set custom bow parameters
            bow_velocity = ( bow_velocity / velo_floor ) + ( ( ( bow_velocity / velo_floor ) / num_levels ) * gen_l );
            String_bow();

            % RUN SIMULATION
            CelloSim_engine();

            % FINALISE
            CelloSim_finalise();
            %%

```

```

% Edit and copy output to inputSim directory
[ x, fs ] = audioread( strcat( 'output/string', num2str( active_string + 1 ), '.wav' ) );
x = x( fs * 1 : end );
filename = strcat( 'inputSim/sim_', num2str( active_string + 1 ), '_', num2str( gen_n ), '_', num2str( gen_l ), '.wav' );
fprintf( 'Writing %s... ', filename );
audiowrite( filename, x, fs );
fprintf( 'done.\n' );
end
end
end
fprintf( '\ninputSim files generated.\n' );

```

compareCQT.m

```
% *****
% COMPARISON FUNCTION
%
% Name:    compareCQT.m
% Date:    02/05/2016
% Author:  Dave Humphreys, c1322278
% Desc:    Compares two audio samples <a> and <b>,
%           at sample rate <fs>, by calculating the
%           difference between their constant-Q
%           transforms. Returns a difference value
%           in dBW. A value of 0 = samples are
%           identical.
% *****

function overall_diff = compareCQT( a, b, fs )
    % Make sure sample dimensions match, and fade start and end sections
    s = min( size( a, 1 ), size( b, 1 ) );
    % Fade sample start and ends proportionally to the sample frequency
    fade_se_t = 0.005; % (s)
    fade_se = [ linspace( 0, 1, round( fade_se_t * fs ) ) ...
                ones( 1, s - 2 * round( fade_se_t * fs ) ) ...
                linspace( 1, 0, round( fade_se_t * fs ) ) ]';
    a = a( 1:s ) .* fade_se;
    b = b( 1:s ) .* fade_se;

    % Test for erroneous audio
    overall_diff = flintmax;
    threshold_level = 0.1;           % full-scale proportion below which audio should be considered in error
    threshold_length = 0.5;          % amount of sample which should contain audio above threshold

    error_level_1 = max( sum( abs( a ) < threshold_level ) / length( a ) > threshold_length, ...
                        sum( abs( b ) < threshold_level ) / length( b ) > threshold_length );
    error_level_2 = max( abs( ( sum( a > 0 ) - sum( a < 0 ) ) / length( a ) ) > threshold_length, ...
                        abs( ( sum( b > 0 ) - sum( b < 0 ) ) / length( b ) ) > threshold_length );
    error_mode = min( abs( mode( a ) ), abs( mode( b ) ) ) == 0;
```

```

if( error_level_1 || error_level_2 || error_mode )
    fprintf( 'compareCQT: ERROR - invalid audio encountered.\n' );
else
    % FFT
    n = 1024;           % bins per octave
    fs_min = 10;        % lowest frequency to analyse
    ya = cqt( a, n, fs, fs_min, fs/2, 'rasterize', 'full', 'gamma', 0, 'normalise', 'sine' );
    yb = cqt( b, n, fs, fs_min, fs/2, 'rasterize', 'full', 'gamma', 0, 'normalise', 'sine' );

    % Normalise output
    ya = abs( real( ya.c ) / max( abs( real( ya.c ) ) ) );
    yb = abs( real( yb.c ) / max( abs( real( yb.c ) ) ) );

    % Create difference matrix
    diff = log( ( ya - yb ) + ( ( ya - yb ) == 0 .* eps ) );

    % Create difference by freq. and overall
    overall_diff = mean( mean( abs( diff ) ) ); % difference in freq. and time domain
end
end

```

Optimise_main.m

```
% *****
% OPTIMISER
%
% Name:    Optimise_main.m
% Date:    02/05/2016
% Author:  Dave Humphreys, c1322278
% Desc:    Standalone script which executes the
%           optimiser once for the sample with
%           filename target_audio. String damping
%           (optim_lpf) and string diameter,
%           linear density and Young's modulus
%           (d, pl and E) may be optimised together
%           or separately by setting optim_lpf and
%           optim_dplE.
%
% *****

function Optimise_main
    tic

    optim_lpf    = true;
    optim_dplE   = true;

    target_audio = 'target_1_1_2.wav';
    string_no    = 1;
    finger_pos   = 0;                % i.e. 0 = open string, 7 = seventh semitone
    offset       = 0.45;             % seconds
    sim_time     = 0.75;             % seconds

    num_taps     = 8;                % number of bands representing the LPF
    T_max        = 2000;             % maximum tension to allow during d, pl and E optimisation

    os = OptimiseS( target_audio, 'results.txt', 'optimise', offset, sim_time, string_no, finger_pos );

    if( optim_lpf )
        os.optimise_lpf( num_taps );
    end
```

```
if( optim_dp1E )
    parameters      = os.set_parameters( 0.00146, 0.0212, 2.0600e+10 );
    parameters_min  = os.set_parameters( 0.0001, 0.005, 6.0000e+8 );
    parameters_max  = os.set_parameters( 0.0015, 0.05, 4.0600e+10 );
    os.optimise_dp1E( parameters, parameters_min, parameters_max, T_max );
end

os.close();

toc
end
```

Optimise_all.m

```
% *****
% OPTIMISER
%
% Name:    Optimise_all.m
% Date:    02/05/2016
% Author:  Dave Humphreys, c1322278
% Desc:    Standalone script which executes the
%           optimiser once for each note specified,
%           storing the outputs generated in
%           different folders, and the results in
%           different files. Can be made to continue
%           from a given point by setting optim_lpf,
%           optim_dplE, start_string and start_pos.
% *****

function Optimise_all
    optim_lpf    = true;
    optim_dplE   = true;

    offset       = 0.45;           % seconds
    sim_time     = 0.75;           % seconds

    num_taps     = 8;               % number of bands representing the string's damping vector
    T_max        = 2000;            % maximum tension to allow during d, pl and E optimisation

    start_string = 1;
    start_pos    = 1;

    for string_no=start_string:4
        for fp=start_pos:2:3

            folder_name = strcat( 'optimise_', num2str( string_no ), '_', num2str( fp ) );
            results_name = strcat( 'results_', num2str( string_no ), '_', num2str( fp ), '.txt' );
            target_audio = strcat( 'target_', num2str( string_no ), '_', num2str( fp ), '_2.wav' ); % choose level 2 for these runs

            finger_pos = 0;
            if( fp == 3 )
                finger_pos = 7;           % i.e. 0 = open string, 7 = seventh semitone
            end
        end
    end
```

```

tic

os = OptimiseS( target_audio, results_name, folder_name, offset, sim_time, string_no, finger_pos );
fprintf( 'Using target %s...\n', target_audio );

if( optim_lpf )
    os.optimise_lpf( num_taps );
end
if( optim_dplE )
    parameters      = os.set_parameters( 0.00146, 0.0212, 2.0600e+10 );
    parameters_min  = os.set_parameters( 0.0001, 0.005, 6.0000e+8 );
    parameters_max  = os.set_parameters( 0.0015, 0.05, 4.0600e+10 );
    os.optimise_dplE( parameters, parameters_min, parameters_max, T_max );
end

os.close();

toc

start_pos = 1;
optim_lpf = true;
optim_dplE = true;
end
end
end

```


OptimiseS.m

```
% *****
% OPTIMISER
%
% Name:    OptimiseS.m
% Date:    02/05/2016
% Author:  Dave Humphreys, c1322278
% Desc:    Optimiser class. Operates on the Cello
%           Simulation using MCS and fminsearchbnd
%           to find the minimum difference between
%           the constant-Q transforms of generated
%           audio and a given sample.
% *****
```

```
classdef OptimiseS
    properties
        output_dir = '';           % directory to store/fetch generated audio from
        rID         = 0;           % result output text file handle
        fs          = 0;           % sample rate (Hz)
        sim_time    = 0;           % time to run simulation for (s)
        offset      = 0;           % time to begin comparison (s)
        s_range     = [];          % range of samples to analyze
        s           = 0;           % string number to optimize for
        finger_pos  = 0;           % finger position (notes up from open) for this run
        values_ofst = 20;          % offset for optimiser vector's input values

        x_input     = [];          % target sound, given as input

        curveObj    = [];          % the curve generator object - takes values vector, maps to larger vector
        curveMlt    = 2;           % width per band of the curveObj to affect (in bands)
        lpfOrig     = [];          % original state of LPF vector before optimisation
        lpfIDs      = {};          % list mapping each calculate output to an output file number
        lpfNextID   = 0;           % next output file number
        lpfIDs_filename = '';      % location of the saved lpfIDs file (mapping list)

        % indexes to parameters[] (optimiser values)
        i_d = 1;
        i_pl = 2;
        i_E = 3;
    end
end
```

```

fund          = 0;          % fundamental frequency of the current string
T_max         = 0;          % maximum allowed tension during optimisation testing
end

methods
%% CONSTRUCTOR
function self = Optimises( input_wav, result_file, output_dir, offset, sim_time, string, finger_pos )
    self.output_dir = strcat( output_dir, '/' );
    if( exist( strcat( 'output/', self.output_dir ) ) ~= 7 )
        mkdir( 'output/', self.output_dir );
    end

    self.s = string - 1;
    self.sim_time = sim_time;
    self.offset = offset;
    self.finger_pos = finger_pos;

    % Add path for optimiser and mcs
    addpath( 'mcs', 'minq5', 'gls', 'parseargs' );

    % Open the result output file
    self.rID = fopen( result_file, 'w' );

    % Load target sound
    [ self.x_input, self.fs ] = audioread( strcat( 'inputTarget/', input_wav ) );
    self.s_range = round( self.fs*self.offset ) : round( self.fs*self.sim_time - 1 ); % calculate start to end sample range
    self.x_input = self.x_input( self.s_range ); % compare only this section
end

% Cleanly close the open results file
function close( self )
    fclose( self.rID );
end

%% Get / Set functions
function parameters = set_parameters( self, d, pl, E )
    parameters = [];
    parameters = self.set_d( parameters, d );
    parameters = self.set_pl( parameters, pl );
    parameters = self.set_E( parameters, E );
end

```

```

function parameters = set_d( self, parameters, d )
    parameters( self.i_d ) = d;
end

function parameters = set_pl( self, parameters, pl )
    parameters( self.i_pl ) = pl;
end

function parameters = set_E( self, parameters, E )
    parameters( self.i_E ) = E;
end

function d = get_d( self, parameters )
    d = parameters( self.i_d );
end

function pl = get_pl( self, parameters )
    pl = parameters( self.i_pl );
end

function E = get_E( self, parameters )
    E = parameters( self.i_E );
end

%% Optimisation functions
% string damping coefficients (formerly called 'lpf') using fminsearchbnd
function lpf = optimise_lpf( self, num_taps )
    % Load String constants
    String_constants();

    % Store default state of damping vector, create a curve object and set up optimiser offset and boundaries
    n = s_cnst( o_cnst * self.s + i_cnst_N );
    self.curveObj = curveGen( n, num_taps, self.curveMlt );
    self.lpfOrig = s_cnst( o_cnst * self.s + i_cnst_lpf : o_cnst * self.s + i_cnst_lpf + n - 1 );
    values_min = zeros( 1, num_taps );
    values_max = ones( 1, num_taps );
    values_init = values_min;

    [ values, FVAL, EXITFLAG ] = fminsearchbnd( @self.run_lpf_test, values_init + self.values_ofst, ...
        values_min + self.values_ofst, values_max + self.values_ofst );

```

```

% Display / write the optimiser return codes
fprintf( '\n\n*** optimise_lpf() RESULT: fval=%d, exitflag=%d ***\n', FVAL, EXITFLAG );
fprintf( self.rID, '\n\n*** optimise_lpf() RESULT: fval=%d, exitflag=%d ***\n', FVAL, EXITFLAG );

% Generate a curve object from the result, and write it
[ self.curveObj, curve ] = self.curveObj.calcCurve( values' - self.values_ofst );
% Prevent the principal frequencies from becoming suppressed
curve( 1 : 5 ) = 0;
curve( 11 : 15 ) = 0;
lpf = ( 1 - curve ) .* ( ( 1 - curve ) > 0 );
self.write_lpf( FVAL, lpf );
plot( lpf );
end

% d, pl and E, using MCS
function parameters = optimise_dpLE( self, parameters, parameters_min, parameters_max, T_max )
    self.T_max = T_max;

    % Load String constants
    String_constants();

    % Set sought fundamental
    % Load the CelloSim default values
    CelloSim_h();
    % Initialise all strings
    Strings_init(); % indexed 0-3
    % * From String_fundamental * %
    % calculate offsets
    s_str( i_str_curStr ) = self.s;
    curS0fst = o_str * s_str( i_str_curStr );
    curSc0fst = o_cnst * s_str( i_str_curStr );

    % Get current fundamental frequency
    self.fund = s_str( curS0fst + i_str_w0 ) / (2*pi);
    if( s_str( curS0fst + i_str_finger_position ) > 0 )
        self.fund = self.fund * ( 1 / ( s_str( curS0fst + i_str_finger_position ) / s_cnst( curSc0fst + i_cnst_L ) ) );
    end

    [ parameters, FVAL, EXITFLAG ] = optimise( @self.run_dpLE_test, ...
        'method', 'mcs', ...
        'x0', parameters, ...
        'minx', parameters_min, ...
        'maxx', parameters_max );

```

```

% Insert the new parameters into the simulation
s_cnst( o_cnst * self.s + i_cnst_d ) = self.get_d( parameters );
s_cnst( o_cnst * self.s + i_cnst_pl ) = self.get_pl( parameters );
s_cnst( o_cnst * self.s + i_cnst_E ) = self.get_E( parameters );

% Make it recalculate its internal values
s_str( i_str_curStr ) = self.s;
String_reset();

% Obtain the matching tension (T) for parameters[]
T = self.calculate_T( s_cnst( o_cnst * self.s + i_cnst_E ), ...
                    s_str( o_str * self.s + i_str_I ), ...
                    s_cnst( o_cnst * self.s + i_cnst_pl ), ...
                    s_str( o_str * self.s + i_str_div_pc_L ) );

fprintf( '\n\n*** optimise_dplE() RESULT: fval=%d, exitflag=%d ***\n', FVAL, EXITFLAG );
fprintf( '\tfor d=%f, pl=%f, E=%f, T=%f\n', self.get_d( parameters ), self.get_pl( parameters ), self.get_E( parameters ),
T );

fprintf( self.rID, '\nresult = %f;\n', FVAL );
fprintf( self.rID, 'd = %f;\tpl = %f;\tE = %f;\tT = %f;\n', self.get_d( parameters ), self.get_pl( parameters ),
self.get_E( parameters ), T );
end

%% Single optimiser iteration functions
% Single damping vector optimiser iterations
function result = run_lpf_test( self, values )
% Load the CelloSim default values
CelloSim_h();
% Initialise all strings
Strings_init(); % indexed 0-3
% Initialise convolution vector extractor
IR_construct(); % indexed 1-4, for now
% Initialise body convolution code
Convolve_construct();
% Initialise FFT velocity feedback code
FFTV_construct();

% Generate a curve from this iteration's values
[ self.curveObj, curve ] = self.curveObj.calcCurve( values' - self.values_ofst );
% Prevent the principal frequencies from becoming suppressed
curve( 1 : 5 ) = 0; % for open string

```

```

curve( 11 : 15 ) = 0;          % for +7 semitones
% Apply it to the string's damping (LPF) vector
s_cnst( o_cnst * self.s + i_cnst_lpf : o_cnst * self.s + i_cnst_lpf + s_cnst( o_cnst * self.s + i_cnst_N ) - 1 ) ...
    = ( 1 - curve ) .* ( ( 1 - curve ) > 0 );

% Display the current LPF curve under test
fprintf( 'Plotting LPF state...\n' );
plot( s_cnst( o_cnst * self.s + i_cnst_lpf : o_cnst * self.s + i_cnst_lpf + s_cnst( o_cnst * self.s + i_cnst_N ) - 1 ) );
axis( [ 0 100 0 1 ] );
pause( 0.000001 );

% FETCH A FILE or RUN THE SIM (and get the relevant segment of its output)
% Write out the state of LPF (and display the difference from the default vector) for this test
fprintf( 'Computing for diff %s\n', sprintf( '%f, ', ...
    s_cnst( o_cnst * self.s + i_cnst_lpf : o_cnst * self.s + i_cnst_lpf + s_cnst( o_cnst * self.s + i_cnst_N ) - 1 ) ...
    - self.lpfOrig ) );

% Load the known lpfIDs, so we can see if audio generated from these parameters already exists
self = self.load_lpfIDs();

% Check for its existence
compute = true;
filename = self.generate_lpf_filename( s_cnst( o_cnst * self.s + i_cnst_N ), ...
    s_cnst( o_cnst * self.s + i_cnst_lpf : o_cnst * self.s + i_cnst_lpf + s_cnst( o_cnst * self.s + i_cnst_N )
- 1 ) );
idx = find( ismember( self.lpfIDs, filename ) );
% If it exists, load it!
if( ~isempty( idx ) )
    [ x_sim, fs_load ] = audioread( self.generate_full_filename( strcat( self.output_dir, num2str( idx ), '_' ) ) );
    if( fs_load == self.fs )
        compute = false;
        fprintf( '(Loaded existing file "%s"...)\n', filename );
    end
% If it doesn't exist, set function variables to allow its creation
else
    idx = 1;
    if( length( self.lpfIDs{ 1 } ) > 1 )
        idx = length( self.lpfIDs ) + 1;
    end
end

% If there's something to compute, do it!
if( compute == true )
    fprintf( '(Generating audio file "%s"...)\n', filename );

```

```

lpfIDfilename = strcat( self.output_dir, num2str( idx ), '_' );
self.lpfIDs{ idx } = [ filename ];

% > RUN SIMULATION with new LPF <
% override simulation time
sim_time = self.sim_time;
% disable the stopping of bowing
stop_bowing = false;
% basic bowing / fingering settings
active_string = self.s;
active_pos     = self.finger_pos;
% disable 'realism' settings
bow_noise_active = false;
vary_bow_force   = false;
vary_bow_velocity = false;
vary_finger_pos  = false;
% disable special functions
active_melody = 0;

% Set initial state
CelloSim_initial_state();

%% RUN SIMULATION
CelloSim_engine();

%% FINALISE
CelloSim_finalise();
% > End of simulation <

x_sim = audioread( self.generate_full_filename( '' ) );
x_sim = x_sim( self.s_range );
% save the output to the optimiser store
audiowrite( self.generate_full_filename( lpfIDfilename ), x_sim, self.fs );
end

% Compare the output to our target sound
result = compareCQT( self.x_input, x_sim, self.fs );
fprintf( '** Result: %f **\n\n', result );
self.write_lpf( result, s_cnst( o_cnst * self.s + i_cnst_lpf : o_cnst * self.s + i_cnst_lpf + s_cnst( o_cnst * self.s +
i_cnst_N ) - 1 ) );

```

```

% Save the lpfIDs list, including this addition
self.save_lpfIDs();
end

% Single d, pl and E optimiser iteration
function result = run_dplE_test( self, parameters )
% If we fail to run, return accordingly
result = flintmax;

% Load the CelloSim default values
CelloSim_h();
% Initialise all strings
Strings_init(); % indexed 0-3
% Initialise convolution vector extractor
IR_construct(); % indexed 1-4, for now
% Initialise body convolution code
Convolv_construct();
% Initialise FFT velocity feedback code
FFTV_construct();

% Set the simulation to the new parameters
s_cnst( o_cnst * self.s + i_cnst_d ) = self.get_d( parameters );
s_cnst( o_cnst * self.s + i_cnst_pl ) = self.get_pl( parameters );
s_cnst( o_cnst * self.s + i_cnst_E ) = self.get_E( parameters );

% Make it recalculate its internal values
s_str( i_str_curStr ) = self.s;
String_reset();

% Compute a new tension figure (to retain original pitch)
T = self.calculate_T( s_cnst( o_cnst * self.s + i_cnst_E ), ...
                    s_str( o_str * self.s + i_str_I ), ...
                    s_cnst( o_cnst * self.s + i_cnst_pl ), ...
                    s_str( o_str * self.s + i_str_div_pc_L ) );

% Only compute if we have a value tension figure
if( T >= 0 && T <= self.T_max )
% Set for the new tension
s_cnst( o_cnst * self.s + i_cnst_T ) = T;
s_str( i_str_curStr ) = self.s;
String_reset();

```



```

% FETCH A FILE / RUN THE SIM (and get the relevant segment of its output)
fprintf( 'Computing for d=%f, pl=%f, E=%f, T=%f ...\n', ...
    s_cnst( o_cnst * self.s + i_cnst_d ), s_cnst( o_cnst * self.s + i_cnst_pl ), ...
    s_cnst( o_cnst * self.s + i_cnst_E ), s_cnst( o_cnst * self.s + i_cnst_T ) );
fprintf( self.rID, 'd=%f; pl=%f; E=%f; T=%f;\n', ...
    s_cnst( o_cnst * self.s + i_cnst_d ), s_cnst( o_cnst * self.s + i_cnst_pl ), ...
    s_cnst( o_cnst * self.s + i_cnst_E ), s_cnst( o_cnst * self.s + i_cnst_T ) );
compute = true;
filename = self.generate_dplE_filename( s_cnst( o_cnst * self.s + i_cnst_d ), ...
    s_cnst( o_cnst * self.s + i_cnst_pl ), ...
    s_cnst( o_cnst * self.s + i_cnst_E ) );
ffilename = self.generate_full_filename( filename );

if( exist( ffilename, 'file' ) )
    [ x_sim, fs_load ] = audioread( ffilename );
    if( fs_load == self.fs )
        compute = false;
        fprintf( '(Loaded existing file "%s"...)\n', ffilename );
    end
end
if( compute == true )
    fprintf( '(Generating audio file "%s"...)\n', ffilename );

    % > RUN SIMULATION with new d, pl and E <
    % override simulation time
    sim_time = self.sim_time;
    % disable the stopping of bowing
    stop_bowing = false;
    % basic bowing / fingering settings
    active_string = self.s;
    active_pos = self.finger_pos;
    % disable 'realism' settings
    bow_noise_active = false;
    vary_bow_force = false;
    vary_bow_velocity = false;
    vary_finger_pos = false;
    % disable special functions
    active_melody = 0;

    % Set initial state
    CelloSim_initial_state();

```

```

%% RUN SIMULATION
CelloSim_engine();

%% FINALISE
CelloSim_finalise();
% > End of simulation <

x_sim = audioread( self.generate_full_filename( ' ' ) );
x_sim = x_sim( self.s_range' );
% save the output to the optimiser store
audiowrite( ffilename, x_sim, self.fs );
end

% Compare the output to our target sound
result = compareCQT( self.x_input, x_sim, self.fs );
else
fprintf( 'Skipping computation (d=%f, pl=%f, E=%f, T=%f).\n', ...
    s_cnst( o_cnst * self.s + i_cnst_d ), s_cnst( o_cnst * self.s + i_cnst_pl ), ...
    s_cnst( o_cnst * self.s + i_cnst_E ), T );
end

fprintf( 'Result (d=%f, pl=%f, E=%f, T=%f):\t%f\n', ...
    s_cnst( o_cnst * self.s + i_cnst_d ), s_cnst( o_cnst * self.s + i_cnst_pl ), ...
    s_cnst( o_cnst * self.s + i_cnst_E ), s_cnst( o_cnst * self.s + i_cnst_T ), result );
fprintf( self.rID, 'Result (d=%f, pl=%f, E=%f, T=%f):\t%f\n', ...
    s_cnst( o_cnst * self.s + i_cnst_d ), s_cnst( o_cnst * self.s + i_cnst_pl ), ...
    s_cnst( o_cnst * self.s + i_cnst_E ), s_cnst( o_cnst * self.s + i_cnst_T ), result );
end

%% Utility functions
% Create a (long!) unique filename from the LPF vector
function filename = generate_lpf_filename( self, N, lpf )
filename = strcat( 's', num2str( self.s ), '_lpf-' );
for ii=1:N
    filename = strcat( filename, num2str( lpf(ii), 10 ), '_' );
end
end

```

```

% Create a unique filename from the d, pl and E parameters
function filename = generate_dplE_filename( self, d, pl, E )
    filename = strcat( self.output_dir, 's', num2str( self.s ), ...
        '_d-', num2str( d ), '_pl-', num2str( pl ), '_e-', num2str( E ), '_' );
end

% Take a filename prefix, and return a full path to its output.wav
function filename = generate_full_filename( self, fn )
    filename = strcat( 'output/', fn, 'output.wav' );
end

% Load the list of lpfIDs into memory, or initialise the index if no list exists
function self = load_lpfIDs( self )
    % Initialise the list
    self.lpfNextID = 1;
    self.lpfIDs = { [ '' ] };
    self.lpfIDs_filename = strcat( 'output/', self.output_dir, 'lpfIDs.txt' );

    % Do we have an existing list?
    if( exist( self.lpfIDs_filename, 'file' ) )
        % If so, load it!
        f_lpfIDs = fopen( self.lpfIDs_filename, 'r' );
        data = fgetl( f_lpfIDs );
        while( length( data ) > 1 )
            self.lpfIDs{ self.lpfNextID } = data;
            self.lpfNextID = self.lpfNextID + 1;
            data = fgetl( f_lpfIDs );
        end
        fclose( f_lpfIDs );
    end
end

% Save the list of lpfIDs to local storage
function save_lpfIDs( self )
    f_lpfIDs = fopen( self.lpfIDs_filename, 'w' );
    for ii=1:length( self.lpfIDs )
        fprintf( f_lpfIDs, '%s\n', self.lpfIDs{ ii } );
    end
    fclose( f_lpfIDs );
end

```

```

% Write out the vector lpf to the results file
function write_lpf( self, result, lpf )
    fprintf( self.rID, 'lpf = [ ' );
    for ii=1:length( lpf )
        fprintf( self.rID, '%f, ', lpf( ii ) );
        if( mod( ii, 8 ) == 0 && ii < length( lpf ) )
            fprintf( self.rID, '...\n\t' );
        end
    end
    fprintf( self.rID, '];\n' );
    fprintf( self.rID, 'result = %f;\n\n', result );
end

function T = calculate_T( self, E, I, pl, str_div_pc_L )
    T = ( ( -( E * I / pl ) * ...
        ( pi * str_div_pc_L )^4 + (self.fund*(2*pi))^2 ) / ...
        ( pi * str_div_pc_L )^2 ) * pl; % find a new T for our fundamental
end

end
end

```

curveGen.m

```
% *****
% CURVE GENERATOR
%
% Name:    curveGen.m
% Date:    02/05/2016
% Author:  Dave Humphreys, c1322278
% Desc:    This class generates a vector <num_points>
%           in length from a controlling vector
%           <num_taps> in length, approximately
%           representing a curve. The bandwidth of
%           each section can be set using
%           <tap_width_multiplier>. Its purpose is
%           to map a (potentially large) vector
%           space to a selectively smaller one.
% *****

classdef curveGen
    properties
        num_points = 0;
        num_taps = 0;
        tap_width_multiplier = 0;
        tap_spacing = 0;
        tap_width = 0;
        tap = [];
        tap_centres = [];
        tap_values = [];
    end

    methods
        function self = curveGen( num_points, num_taps, tap_width_multiplier )
            % Input parameters
            self.num_points = num_points;
            self.num_taps = num_taps;
            self.tap_width_multiplier = tap_width_multiplier;

            % Make a tap curve
            self.tap_spacing = self.num_points / ( self.num_taps - 1 );
            self.tap_width = round( self.tap_spacing * self.tap_width_multiplier );
            self.tap = self.genTap();
        end
    end
end
```

```

% Make array and initial values
self.tap_centres = ones( 1, num_taps );
self.tap_values = ones( 1, num_taps ); % initially all at maximum
current_centre = 1;
for i=2:num_taps - 1
    current_centre = current_centre + self.tap_spacing;
    self.tap_centres( i ) = round( current_centre );
end
self.tap_centres( self.num_taps ) = self.num_points;
end

function [ self, a ] = calcCurve( self, tap_values )
self.tap_values = tap_values .* ( tap_values > 0 );

a = zeros( 1, self.num_points );

for i=1:self.num_taps
    thisTap = self.tap * self.tap_values( i );
    start = self.tap_centres( i ) - floor( self.tap_width / 2 ); % Index into array
    stop = self.tap_centres( i ) + ceil( self.tap_width / 2 ) - 1; %
    tapStart = 1; % Index into tap itself (small)
    tapStop = self.tap_width; %
    if( start < 1 )
        tapStart = tapStart - start + 1;
        start = 1;
    end
    if( stop > self.num_points )
        tapStop = tapStop - ( stop - self.num_points );
        stop = self.num_points;
    end

    a( start : stop ) = max( a( start : stop ), thisTap( tapStart : tapStop ) );
end
end

```

```

function tap = genTap( self )
    % Generate a tap_width length sqrt half-sine curve
    rad_step = pi / ( self.tap_width + 1 );

    r = rad_step;
    tap = zeros( 1, self.tap_width );
    for i=1:ceil( self.tap_width )
        tap(i) = sin( r );
        r = r + rad_step;
    end

    tap = sqrt( tap );
end
end
end

```

Appendix B: Software used to create the subjective evaluation tests

```
import pygame
from random import random, shuffle
from time import asctime, sleep

# Setup
pygame.init()

prompt = " > "
results = "% test begins\n"

num_plays = 16                      # Must be divisible by 2
duplicates = 0.33                   # How many of the tests are duplicates
num_strings = 4
num_notes_per_string = 4
level = 2
modes = ( "real-mine_", "real-kontakt_", "sim-csimmine_", "sim-csimkontakt_", "artifastring_" )
combis = ( (0,2), (1,3), (0,4), (1,4), (2,4), (3,4) )      # i.e. real (mine) and simulated (mine), real (kontakt) and simulated
(kontakt) etc.
real = 1
sim = 2
arti = 4

# Show welcome message
print( "\033[2J\033[0;0H" )
print( " Welcome to my cello research software!\n (Written by Dave Humphreys, Cardiff University c1322278)\n\n" )
print( " The purpose of this test is to discover whether you are\n listening to sounds from a real cello, or a virtual one.\n\n" )
print( " You will hear " + str( num_plays * 3 ) + " sounds in the headphones.\n" )
print( " Please listen carefully, then type:\n\n 'R' if you want to re-play the sound,\n 'Y' if you heard a real cello sound, or" )
print( " 'N' if you did not hear a real cello sound\n\t(perhaps you heard a virtual one).\n" )
print( " Once you have typed your response, press the <return> key.\n\n" )
print( " Firstly, please type 'Y' if you are a musician / musically minded,\n\tor 'N' if not, and press the <return> key.\n" )
string = raw_input( " Are you musical? (Y/N)" + prompt )
print( " Thank you. When you are ready to start, simply press the <return> key.\n" )
raw_input( prompt )

if( string.upper() == 'Y' ):
    results += "asctime='" + asctime() + "'; musical=1;\n";
else:
    results += "asctime='" + asctime() + "'; musical=0;\n";
results += "% test: identify the real sound\n"
```



```

# Run both "sounds real" tests
for t_num in range( 2 ):

    # Decide on test order
    vscore_count = 0
    ascore_count = 0
    rscore_count = int( num_plays / 2 )
    max_count = int( round( num_plays / 4 ) ) + 1          # limit any one mode from becoming dominant
    tests = []
    num_unique = int( round( ( round( ( 1 - duplicates ) * num_plays ) / 2 ) ) * 2 )
    num_duplicates = num_plays - num_unique
    for n in range( num_unique / 2 ):
        allow = False
        while( allow == False ):
            combi = combis[ int( round( random() * ( len( combis ) - 3 ) ) ) ]
            allow = True
            # Is this combination disallowed?
            if( combi[0] == arti or combi[1] == arti ):
                if( ascore_count >= max_count ):
                    allow = False
                else:
                    ascore_count += 1
            else:
                if( vscore_count >= max_count ):
                    allow = False
                else:
                    vscore_count += 1

        string = int( round( ( random() * ( num_strings - 1 ) ) + 1 ) )
        note = int( round( ( random() * ( num_notes_per_string - 1 ) ) + 1 ) )
        fnpart = str( string ) + "_" + str( note ) + "_" + str( level ) + ".wav"
        tests.append( ( combi[0], modes[ combi[0] ] + fnpart ) )
        tests.append( ( combi[1], modes[ combi[1] ] + fnpart ) )
        if n < ( num_duplicates / 2 ):
            tests.append( ( combi[0], modes[ combi[0] ] + fnpart ) )
            tests.append( ( combi[1], modes[ combi[1] ] + fnpart ) )
    shuffle( tests )

# Run this test set
for n in range( num_plays ):
    print( "\033[2J\033[0;0H" )
    print( " Sound " + str( ( n + 1 ) + ( t_num * num_plays ) ) + " of " + str( num_plays * 3 ) + ":\n\n" )
    print( " Press the <return> key to hear the sound\n\n\n" )
    raw_input( " " )

```

```

#print( "\n (DEBUG): Playing " + str( t_num ) + "_" + tests[n][1] + "... \n\n\n" )
print( "\n Playing the sound...\n\n\n" )
play = True
while( play == True ):
    pygame.mixer.Sound( str( t_num ) + "_" + tests[n][1] ).play();
    while( pygame.mixer.get_busy() ):
        pass
    play = False

    print( " Was that a real cello? Type 'Y' for Yes, or 'N' for No,\n\tthen press the <return> key.\n\n" )
    print( " (To hear the sound again, type 'R' and press the <return> key.)\n\n" )
    string = raw_input( prompt )
    if( string.upper() == 'Y' ):
        if( tests[n][0] < sim ):
            results += "type=" + str( tests[n][0] ) + "; result=1; file='" + tests[n][1] + "';\n";
        else:
            results += "type=" + str( tests[n][0] ) + "; result=0; file='" + tests[n][1] + "';\n";
    elif( string.upper() == 'N' ):
        if( tests[n][0] > real ):
            results += "type=" + str( tests[n][0] ) + "; result=1; file='" + tests[n][1] + "';\n";
        else:
            results += "type=" + str( tests[n][0] ) + "; result=0; file='" + tests[n][1] + "';\n";
    else:
        play = True
        print( "\033[2J\033[0;0H" )
        print( " Okay - playing the sound again...\n\n\n" )

# Show test change message
print( "\033[2J\033[0;0H" )
print( " Thank you. You will now hear " + str( num_plays / 2 ) + " pairs of sounds in the headphones.\n" )
print( " Please listen carefully, then type:\n\n 'R' if you want to re-play the sounds," )
print( " '1' if you felt the first sound was most realistic, or\n '2' if you felt the second sound was most realistic\n" )
print( " Once you have typed your response, press the <return> key.\n\n" )
print( " When you are ready to continue, simply press the <return> key.\n" )
raw_input( prompt )

results += "% test: select the most realistic sound\n"

# Generate the simulated sound comparison tests
tests = []

```

```

num_unique = int( round( ( round( ( 1 - duplicates ) * num_plays ) / 2 ) ) * 2 )
num_duplicates = num_plays - num_unique
for n in range( num_unique / 2 ):
    combin = int( round( random() ) ) + ( len( combis ) - 2 )
    string = int( round( ( random() * ( num_strings - 1 ) ) + 1 ) )
    note = int( round( ( random() * ( num_notes_per_string - 1 ) ) + 1 ) )
    fnpart = str( string ) + "_" + str( note ) + "_" + str( level ) + ".wav"
    tests.append( ( combin, fnpart ) )
    if n < ( num_duplicates / 2 ):
        tests.append( ( combin, fnpart ) )
shuffle( tests )

# Run the simulated sound comparison tests
for n in range( num_plays / 2 ):
    print( "\033[2J\033[0;0H" )
    print( " Sounds " + str( num_plays * 2 + ( n * 2 + 1 ) ) + " and " + str( num_plays * 2 + ( n * 2 + 2 ) ) + " of " +
str( num_plays * 3 ) + ":\n\n" )
    print( " Press the <return> key to hear the two sounds\n\n\n" )
    raw_input( " " )

    test = list( combis[ tests[n][0] ] );
    shuffle( test )

    #print( "\n (DEBUG): Playing 1_" + modes[ test[0] ] + tests[n][1] + "..." )
    #print( " (DEBUG): Playing 1_" + modes[ test[1] ] + tests[n][1] + "... \n\n\n\n" )

    play = True
    while( play == True ):
        print( " Playing sound 1..." )
        pygame.mixer.Sound( "1_" + modes[ test[0] ] + tests[n][1] ).play();
        while( pygame.mixer.get_busy() ):
            pass
        sleep( 0.5 )
        print( " Playing sound 2..." )
        pygame.mixer.Sound( "1_" + modes[ test[1] ] + tests[n][1] ).play();
        while( pygame.mixer.get_busy() ):
            pass
        sleep( 0.5 )
        play = False

    print( "\n\n\n Which was most realistic?\n Type '1' for the first sound, or '2' for the second,\n\tthen press the <return>
key.\n\n" )
    print( " (To hear the sounds again, type 'R' and press the <return> key.)\n\n" )

```

[illegible]

Appendix C: Optimiser output

Optimised string parameter settings

Recordings I:

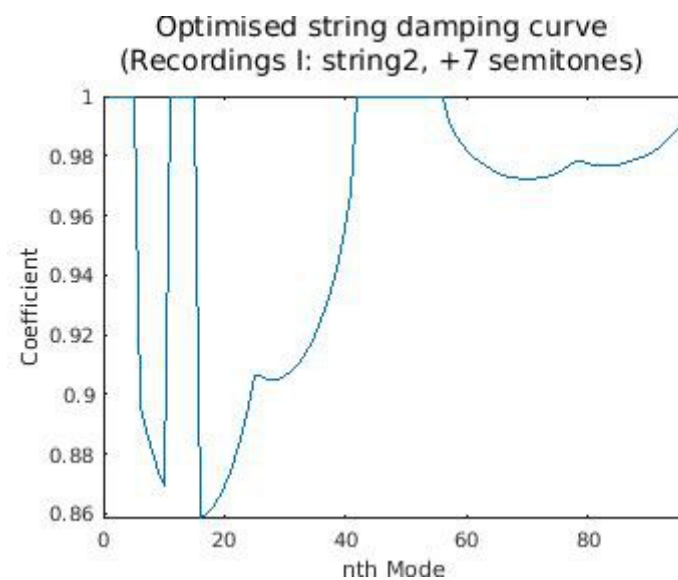
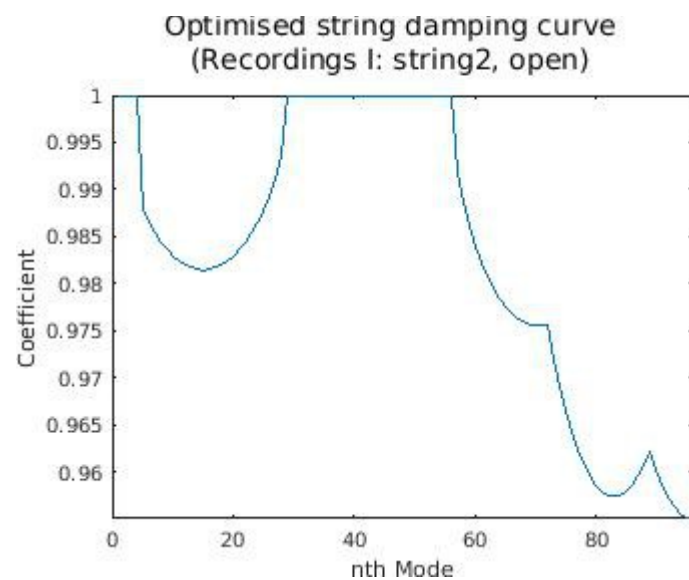
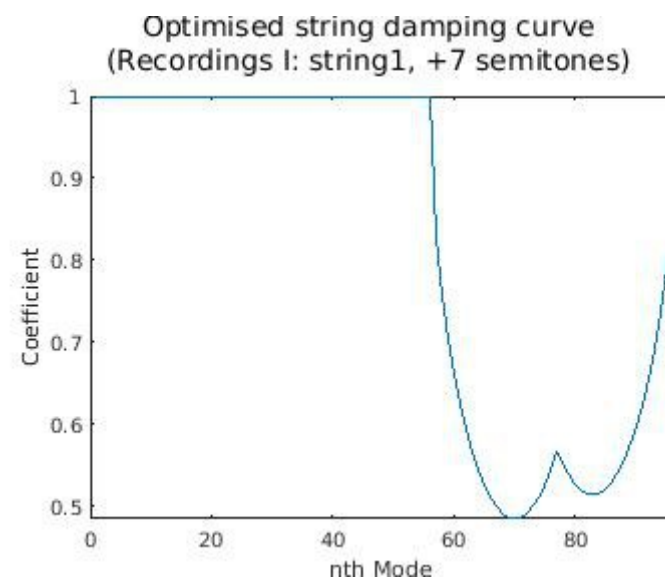
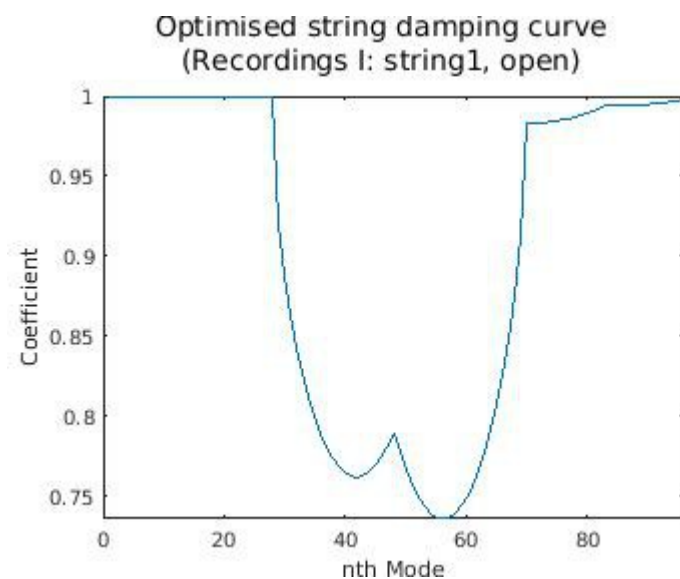
String No.	Note	Diameter (m)	Linear density (kg/m)	Young's modulus	Tension (N)
1	Open	0.0001	0.038075	22872528309.804886	290.229319
	+7 semitones	0.0015	0.05	32417981844.878048	380.956811
2	Open	0.005	0.05	48522333771.452156	84.860634
	+7 semitones	0.001476	0.05	40600000000.0	84.866488
3	Open	0.0008	0.005	15299433520.835085	186.378748
	+7 semitones	0.0015	0.005	34241975307.432041	185.845772
4	Open	0.001148	0.005	46375826040.326843	417.900111
	+7 semitones	0.001126	0.015131	29296542866.379913	1264.887602

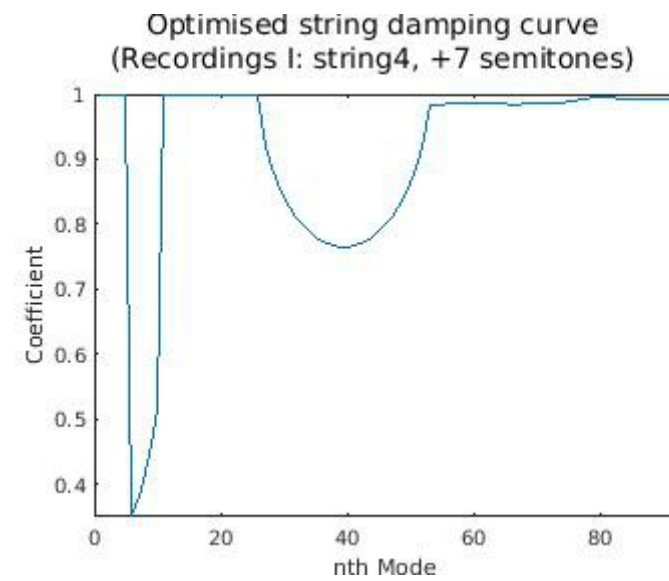
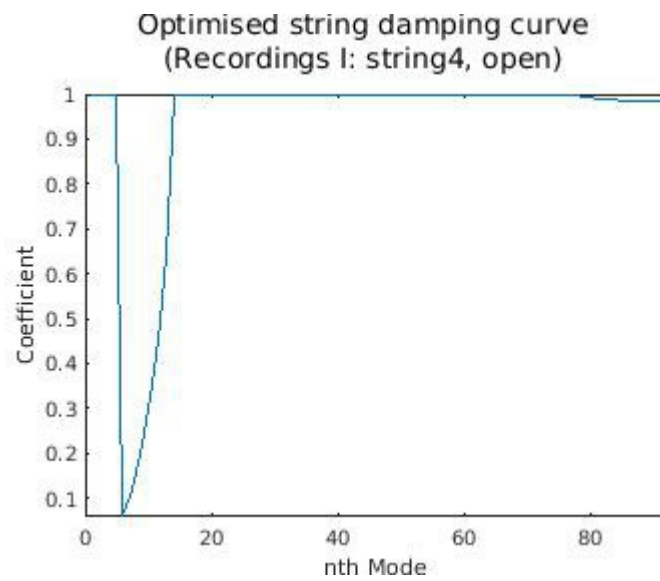
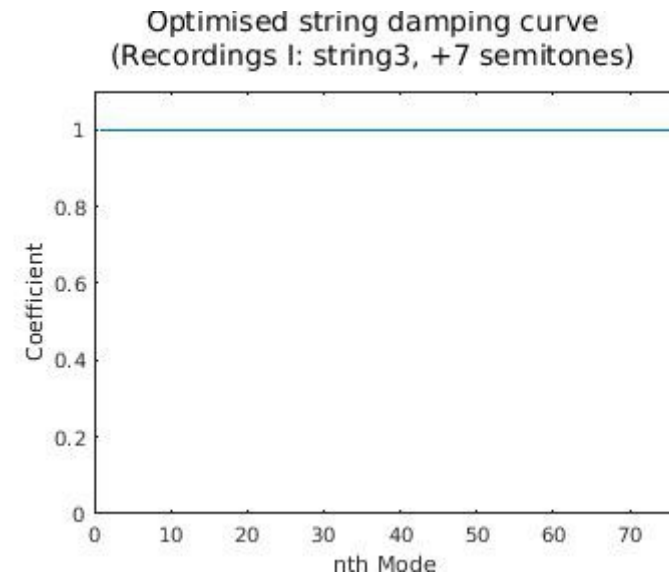
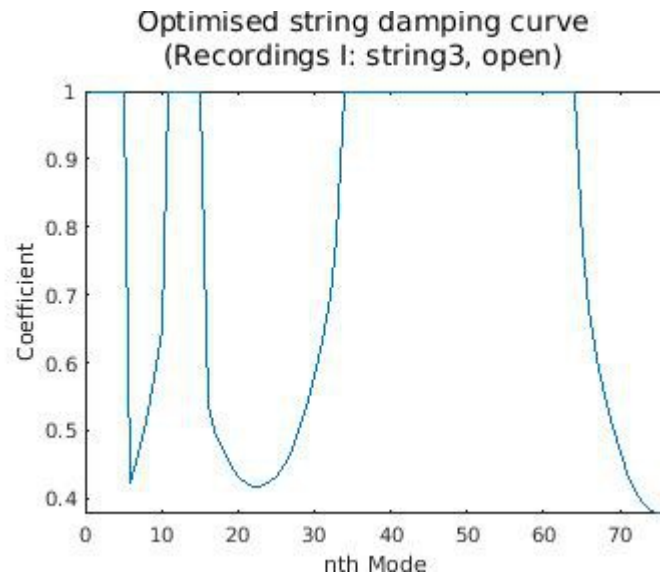
Recordings II:

String No.	Note	Diameter (m)	Linear density (kg/m)	Young's modulus	Tension (N)
1	Open	0.000405	0.025625	600007266.545343	195.326530
	+7 semitones	0.0015	0.0275	40600000000.0	209.402944
2	Open	0.0015	0.005	34303507928.684322	84.938135
	+7 semitones	0.0015	0.005	20749136131.066719	85.012413
3	Open	0.0008	0.009186	40600000000.0	341.775038
	+7 semitones	0.0015	0.005	33077997347.880699	185.852326
4	Open	0.000278	0.005	26849646486.285278	417.993244
	+7 semitones	0.0015	0.005	24211551735.286995	417.852998

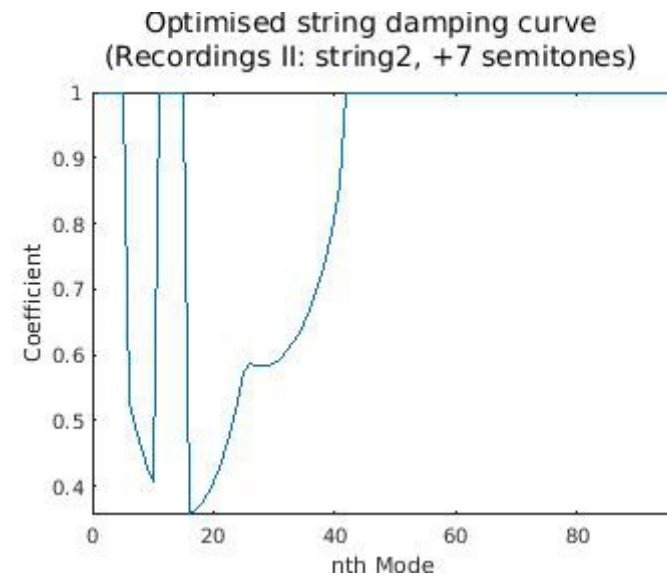
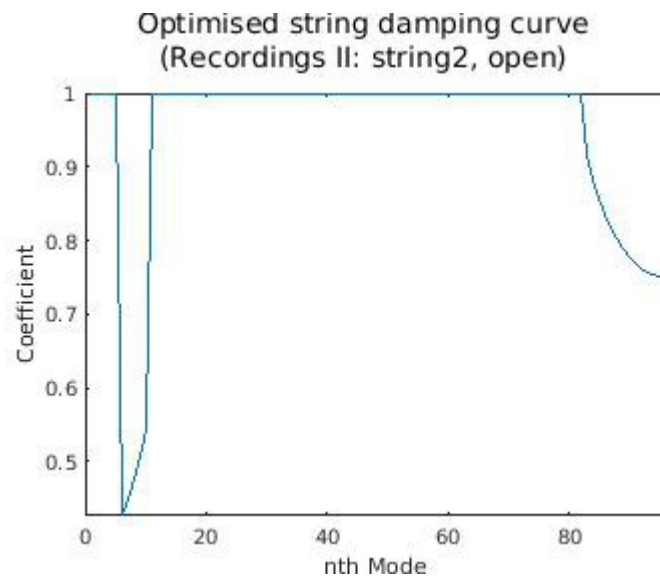
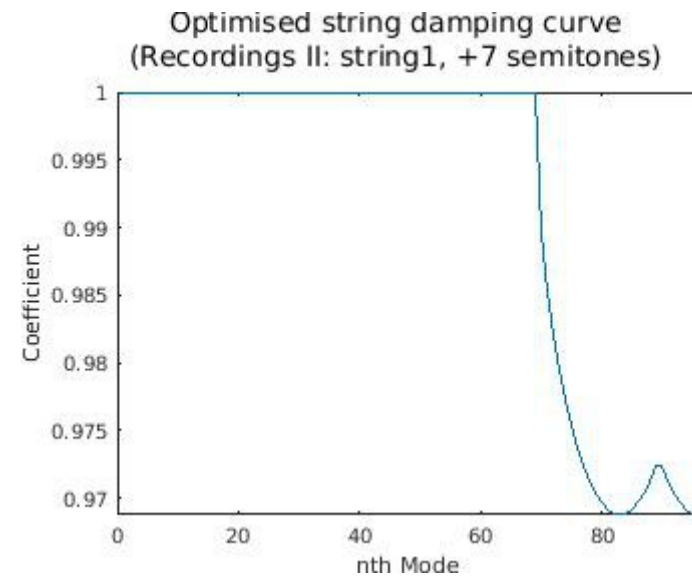
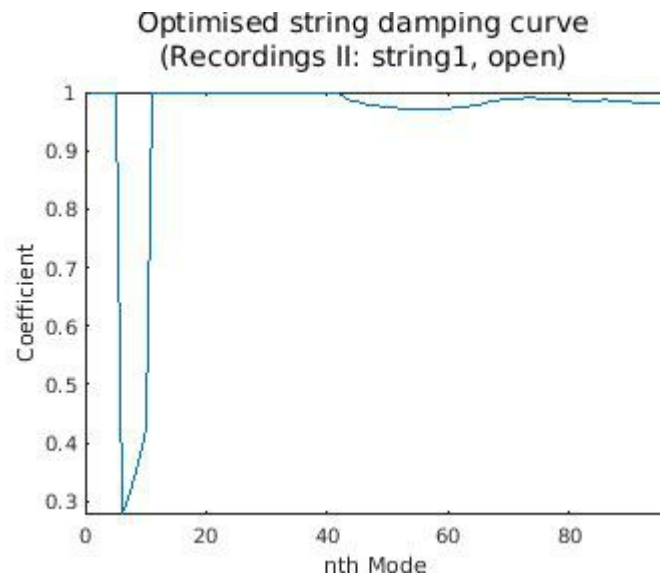
Optimised string damping coefficients

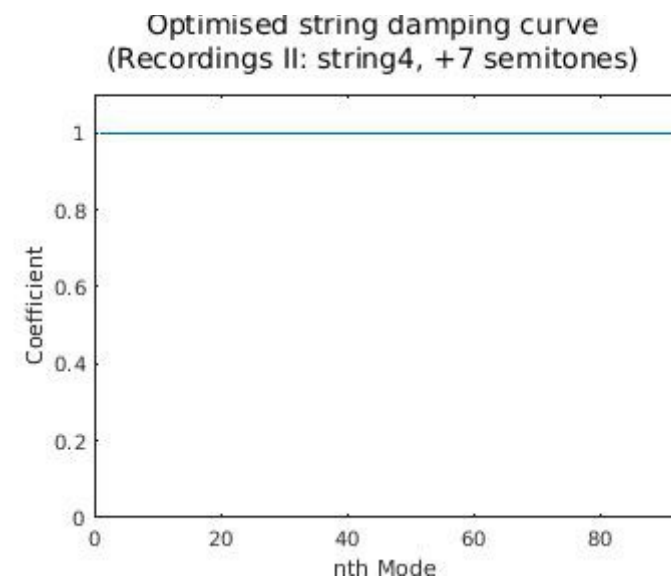
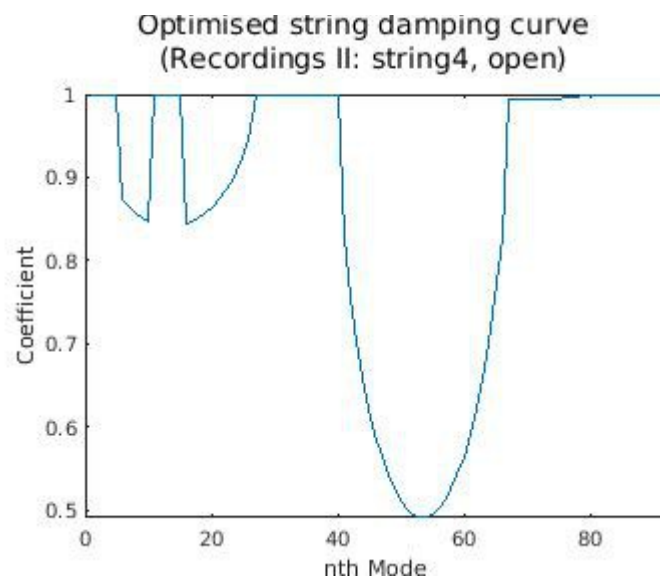
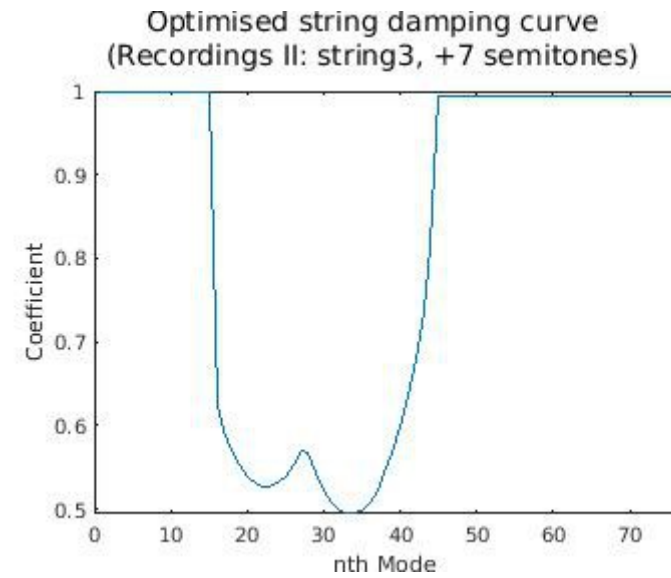
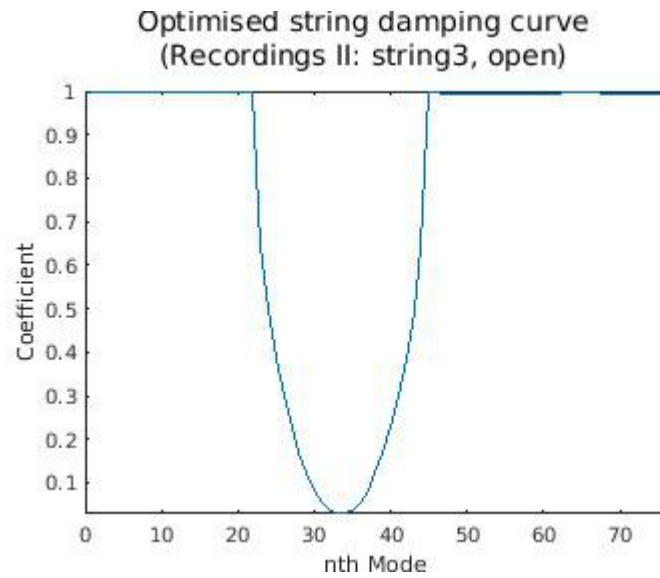
Recordings I:





Recordings II:





Example of search space traversal (for String 1, open):

String damping:

```
lpf = [ 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, ...
        1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, ...
        1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, ...
        1.000000, 1.000000, 1.000000, 1.000000, 0.665389, 0.528279, 0.425301, 0.341301, ...
        0.270595, 0.210386, 0.159104, 0.115788, 0.079824, 0.050806, 0.028463, 0.012616, ...
        0.003149, 0.000000, 0.003149, 0.012616, 0.028463, 0.050806, 0.079824, 0.115788, ...
        0.159104, 0.210386, 0.270595, 0.341301, 0.425301, 0.528279, 0.665389, 1.000000, ...
        1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, ...
        1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, ...
        1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, ...
        1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, ];
```

```
result = 9.101749;
```

```
lpf = [ 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, ...
        1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, ...
        1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, ...
        1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, ...
        1.000000, 1.000000, 0.665389, 0.528279, 0.425301, 0.341301, 0.270595, 0.210386, ...
        0.159104, 0.115788, 0.079824, 0.050806, 0.028463, 0.012616, 0.003149, 0.000000, ...
        0.003149, 0.012616, 0.028463, 0.050806, 0.079824, 0.115788, 0.159104, 0.210386, ...
        0.270595, 0.341301, 0.425301, 0.528279, 0.665389, 1.000000, 1.000000, 1.000000, ...
        1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, ...
        1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, ...
        1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, ];
```

```
result = 8.960984;
```

```
lpf = [ 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, ...
        1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, ...
        1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, ...
        1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, ...
        1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, ...
        1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, ...
        0.665389, 0.528279, 0.425301, 0.341301, 0.270595, 0.210386, 0.159104, 0.115788, ...
        0.079824, 0.050806, 0.028463, 0.012616, 0.003149, 0.000000, 0.003149, 0.012616, ...
        0.028463, 0.050806, 0.079824, 0.115788, 0.159104, 0.210386, 0.270595, 0.341301, ...
        0.425301, 0.528279, 0.665389, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, ...
```

```

    1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, ];
result = 8.801019;

...
lpf = [ 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, ...
        1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, ...
        1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, ...
        1.000000, 1.000000, 1.000000, 1.000000, 0.920153, 0.887435, 0.862861, 0.842817, ...
        0.825944, 0.811577, 0.799339, 0.789003, 0.780421, 0.773497, 0.768165, 0.764383, ...
        0.762124, 0.761373, 0.762124, 0.764383, 0.768165, 0.773497, 0.780421, 0.789003, ...
        0.777095, 0.765612, 0.756079, 0.748387, 0.742464, 0.738263, 0.735754, 0.734919, ...
        0.735754, 0.738263, 0.742464, 0.748387, 0.756079, 0.765612, 0.777095, 0.790689, ...
        0.806649, 0.825392, 0.847658, 0.874956, 0.911301, 0.986345, 0.986388, 0.986517, ...
        0.986733, 0.987038, 0.987435, 0.987926, 0.988517, 0.989218, 0.990040, 0.991005, ...
        0.992152, 0.992770, 0.992747, 0.992770, 0.992838, 0.992953, 0.993115, 0.993326, ...
        0.993587, 0.993901, 0.994273, 0.994710, 0.995222, 0.995832, 0.996579, 0.997573, ];
result = 9.018985;

lpf = [ 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, ...
        1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, ...
        1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, ...
        1.000000, 1.000000, 1.000000, 1.000000, 0.920255, 0.887579, 0.863037, 0.843018, ...
        0.826167, 0.811818, 0.799596, 0.789273, 0.780702, 0.773786, 0.768462, 0.764685, ...
        0.762429, 0.761678, 0.762429, 0.764685, 0.768462, 0.773786, 0.780702, 0.789273, ...
        0.777759, 0.766311, 0.756806, 0.749137, 0.743232, 0.739043, 0.736541, 0.735709, ...
        0.736541, 0.739043, 0.743232, 0.749137, 0.756806, 0.766311, 0.777759, 0.791312, ...
        0.807225, 0.825912, 0.848112, 0.875328, 0.911565, 0.983078, 0.983132, 0.983292, ...
        0.983560, 0.983938, 0.984429, 0.985038, 0.985771, 0.986639, 0.987657, 0.988854, ...
        0.990275, 0.992018, 0.993811, 0.993830, 0.993889, 0.993987, 0.994125, 0.994305, ...
        0.994527, 0.994796, 0.995113, 0.995486, 0.995923, 0.996443, 0.997080, 0.997929, ];
result = 8.352619;

lpf = [ 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, ...
        1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, ...
        1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, ...
        1.000000, 1.000000, 1.000000, 1.000000, 0.919883, 0.887055, 0.862399, 0.842287, ...
        0.825357, 0.810941, 0.798663, 0.788291, 0.779680, 0.772733, 0.767383, 0.763589, ...
        0.761322, 0.760568, 0.761322, 0.763589, 0.767383, 0.772733, 0.779680, 0.788291, ...
        0.778065, 0.766633, 0.757141, 0.749483, 0.743586, 0.739403, 0.736905, 0.736074, ...
        0.736905, 0.739403, 0.743586, 0.749483, 0.757141, 0.766633, 0.778065, 0.791600, ...
        0.807491, 0.826152, 0.848322, 0.875500, 0.911687, 0.982426, 0.982481, 0.982647, ...
        0.982926, 0.983318, 0.983828, 0.984460, 0.985222, 0.986123, 0.987181, 0.988424, ...
        0.989900, 0.991710, 0.993584, 0.993604, 0.993665, 0.993766, 0.993910, 0.994096, ...
        0.994327, 0.994605, 0.994934, 0.995320, 0.995774, 0.996313, 0.996973, 0.997853, ];
result = 8.527386;

```

```

...
lpf = [ 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, ...
        1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, ...
        1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, ...
        1.000000, 1.000000, 1.000000, 1.000000, 0.920292, 0.887631, 0.863100, 0.843091, ...
        0.826248, 0.811905, 0.799689, 0.789371, 0.780804, 0.773892, 0.768569, 0.764794, ...
        0.762539, 0.761789, 0.762539, 0.764794, 0.768569, 0.773892, 0.780804, 0.789371, ...
        0.777653, 0.766200, 0.756690, 0.749018, 0.743110, 0.738919, 0.736416, 0.735584, ...
        0.736416, 0.738919, 0.743110, 0.749018, 0.756690, 0.766200, 0.777653, 0.791213, ...
        0.807133, 0.825829, 0.848040, 0.875269, 0.911523, 0.983211, 0.983264, 0.983423, ...
        0.983689, 0.984064, 0.984551, 0.985155, 0.985882, 0.986743, 0.987754, 0.988941, ...
        0.990351, 0.992080, 0.993767, 0.993787, 0.993846, 0.993945, 0.994084, 0.994265, ...
        0.994489, 0.994759, 0.995079, 0.995454, 0.995895, 0.996418, 0.997060, 0.997915, ];
result = 8.545297;

```

```

lpf = [ 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, ...
        1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, ...
        1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, ...
        1.000000, 1.000000, 1.000000, 1.000000, 0.920259, 0.887584, 0.863043, 0.843025, ...
        0.826175, 0.811826, 0.799605, 0.789282, 0.780712, 0.773797, 0.768472, 0.764695, ...
        0.762439, 0.761689, 0.762439, 0.764695, 0.768472, 0.773797, 0.780712, 0.789282, ...
        0.777701, 0.766250, 0.756743, 0.749072, 0.743165, 0.738976, 0.736473, 0.735641, ...
        0.736473, 0.738976, 0.743165, 0.749072, 0.756743, 0.766250, 0.777701, 0.791258, ...
        0.807175, 0.825867, 0.848073, 0.875296, 0.911543, 0.983026, 0.983080, 0.983241, ...
        0.983510, 0.983889, 0.984381, 0.984992, 0.985727, 0.986597, 0.987619, 0.988820, ...
        0.990245, 0.991993, 0.993841, 0.993860, 0.993918, 0.994016, 0.994153, 0.994332, ...
        0.994554, 0.994821, 0.995136, 0.995507, 0.995943, 0.996460, 0.997094, 0.997939, ];
result = 8.440499;

```

```

*** optimise_lpf() RESULT: fval=8.352619e+00, exitflag=70 ***
lpf = [ 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, ...
        1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, ...
        1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000, ...
        1.000000, 1.000000, 1.000000, 1.000000, 0.920255, 0.887579, 0.863037, 0.843018, ...
        0.826167, 0.811818, 0.799596, 0.789273, 0.780702, 0.773786, 0.768462, 0.764685, ...
        0.762429, 0.761678, 0.762429, 0.764685, 0.768462, 0.773786, 0.780702, 0.789273, ...
        0.777759, 0.766311, 0.756806, 0.749137, 0.743232, 0.739043, 0.736541, 0.735709, ...
        0.736541, 0.739043, 0.743232, 0.749137, 0.756806, 0.766311, 0.777759, 0.791312, ...
        0.807225, 0.825912, 0.848112, 0.875328, 0.911565, 0.983078, 0.983132, 0.983292, ...
        0.983560, 0.983938, 0.984429, 0.985038, 0.985771, 0.986639, 0.987657, 0.988854, ...
        0.990275, 0.992018, 0.993811, 0.993830, 0.993889, 0.993987, 0.994125, 0.994305, ...
        0.994527, 0.994796, 0.995113, 0.995486, 0.995923, 0.996443, 0.997080, 0.997929, ];
result = 8.352619;

```

String diameter, linear density and Young's modulus:

```
d=0.000800; pl=0.027500; E=25600000000.000000; T=209.611580;
Result (d=0.000800, pl=0.027500, E=25600000000.000000, T=209.611580): 9.142339
d=0.000100; pl=0.027500; E=25600000000.000000; T=209.622794;
Result (d=0.000100, pl=0.027500, E=25600000000.000000, T=209.622794): 8.919646
d=0.001500; pl=0.027500; E=25600000000.000000; T=209.484170;
Result (d=0.001500, pl=0.027500, E=25600000000.000000, T=209.484170): 9.048157
d=0.000100; pl=0.038055; E=25600000000.000000; T=290.077637;
Result (d=0.000100, pl=0.038055, E=25600000000.000000, T=290.077637): 8.779778
d=0.000388; pl=0.038055; E=25600000000.000000; T=290.077017;
Result (d=0.000388, pl=0.038055, E=25600000000.000000, T=290.077017): 8.745179
d=0.000388; pl=0.038055; E=15299433520.835085; T=290.077267;
Result (d=0.000388, pl=0.038055, E=15299433520.835085, T=290.077267): 8.744314
d=0.000301; pl=0.038055; E=15299433520.835085; T=290.077506;
Result (d=0.000301, pl=0.038055, E=15299433520.835085, T=290.077506): 8.807399
d=0.000388; pl=0.038055; E=20122990821.622372; T=290.077150;
Result (d=0.000388, pl=0.038055, E=20122990821.622372, T=290.077150): 8.729008
d=0.000388; pl=0.038055; E=20379104798.615269; T=290.077144;
Result (d=0.000388, pl=0.038055, E=20379104798.615269, T=290.077144): 8.692198
...
d=0.000100; pl=0.038073; E=22871655920.568848; T=290.213798;
Result (d=0.000100, pl=0.038073, E=22871655920.568848, T=290.213798): 8.780247
d=0.000100; pl=0.038074; E=22872237513.392872; T=290.224145;
Result (d=0.000100, pl=0.038074, E=22872237513.392872, T=290.224145): 8.828911
d=0.000100; pl=0.038075; E=22872528309.804886; T=290.229319;
Result (d=0.000100, pl=0.038075, E=22872528309.804886, T=290.229319): 8.618805
d=0.000148; pl=0.038118; E=23150342592.856976; T=290.556356;
Result (d=0.000148, pl=0.038118, E=23150342592.856976, T=290.556356): 8.651263
d=0.000195; pl=0.038160; E=23428156875.909069; T=290.883378;
Result (d=0.000195, pl=0.038160, E=23428156875.909069, T=290.883378): 8.695761
d=0.000194; pl=0.038062; E=23175909503.323128; T=290.133784;
Result (d=0.000194, pl=0.038062, E=23175909503.323128, T=290.133784): 8.728753
d=0.000190; pl=0.038050; E=22034065308.416557; T=290.043904;
Result (d=0.000190, pl=0.038050, E=22034065308.416557, T=290.043904): 8.751880
d=0.000161; pl=0.038081; E=21148198041.077965; T=290.274653;
Result (d=0.000161, pl=0.038081, E=21148198041.077965, T=290.274653): 8.755485
d=0.000130; pl=0.038217; E=28754701018.274357; T=291.311574;
Result (d=0.000130, pl=0.038217, E=28754701018.274357, T=291.311574): 8.808079
d=0.000100; pl=0.039185; E=32114916737.871490; T=298.693389;
Result (d=0.000100, pl=0.039185, E=32114916737.871490, T=298.693389): 8.774944

result = 8.618805;
d = 0.000100; pl = 0.038075; E = 22872528309.804886; T = 290.229319;
```

Appendix D: Development diary

- 01/11/15 - 25/01/16** Conducted preliminary research into physical simulations and sample-based virtual instruments by checking research documents and existing commercial products online. Obtained academic license details for Matlab, and installed it on the development machine.
- 26/01/16** Supervisor meeting at 10:00, discussed initial plan. Located and downloaded “Vibrating String Simulator” for Matlab (<http://uk.mathworks.com/matlabcentral/fileexchange/35746-vibrating-string-simulator>)
- 27/01/16** Produced the initial plan document. Examined research conducted at NESS, and documents detailing the progression of research regarding virtual instruments.
- 28/01/16** Researched Fourier components, read Bilbao and Desvages paper on the two-polarisation model, and contacted Professor Bilbao to ask if a publicly available implementation existed.
- 29/01/16** Received a reply from Bilbao – no implementation available currently. However, he provided links to Matlab examples relevant to his system, and a link to a co-authored book on numerical sound synthesis. Downloaded and began reading this. Messaged supervisor, and contacted Dr. Bernard Richardson to enquire about the use of an anechoic chamber and string bar.
- 30/01/16** Examined various mathematical tools from Numerical Recipes, in particular in relation to derivatives and integrals, and creating finite difference models.
- 31/01/16** Made changes to the initial report, and discussed this with supervisor. Became unwell.
- 01/02/16** Investigated addition of bowing motion to Vibrating String Simulator, and possible solutions to Bilbao and Desvages' two-polarisation model.
- 02/02/16** Added basic bowing motion to Vibrating String Simulator, had limited success! Unsure what current scale is for y values and forces.
- 03/02/16** Constructed basic body simulation from bank of DSP filters, coding errors prevent it from functioning.
- 04/02/16** Solved coding errors, but results not promising – something better is needed. Investigated using Freeverb to add room ambience.
- 05/02/16** Located Artifastring model and discussed its relationship to the two-polarisation model with supervisor. Responded to a message from Dr. Richardson, and began constructing a functional diagram for the project's simulation. Tried applying convolution to Vibrating String Simulator output – results seem quite promising!
- 06/02/16** Attempted to implement the two-polarisation model; not successful. Oscillation would not occur until parameters were clearly incorrect, and then motion did not follow the equations - I must have misunderstood the implementation.

- 07/02/16** Attempted to correct these errors, and construct my own model, but again was not successful. I would need to invest a great deal of time in understanding the equations and learning new mathematical concepts, and not enough time remains. Compiled and ran Artifastring, output is excellent! Discussed its use as a basis for the string simulation,
- 08/02/16** Decided to use Artifastring's string simulation code, began implementing it in Matlab. Added everything but its tick() methods.
- 09/02/16** Added tick() methods and tested – implementation works! Found a way to mimic Artifastring's use of the Eigen class in Matlab, and completed coding the core string simulation.
- 10/02/16** Converted code to an object, and tested – code works, and now multiple string instances are possible.
- 11/02/16** Created an implementation using multiple strings, and created a convolution class to allow use of an impulse response – successful.
- 12/02/16** Coupled body simulation (using convolution) to strings using a feedback mechanism. Tested that strings are correctly excited by resonance within the body. The feature works, but appears to have little effect on the overall sound to my ear, surprisingly.
- 13/02/16** Discovered I had misunderstood the a and ad vectors within Artifastring (ad represents the first derivative). Re-wrote code so that the frequency content of audio resulting from the body can be extracted at the correct frequencies for the string's modes, and applied to them as modal velocities. Result is better, but still not quite right.
- 14/02/16** Did CM3202 coursework. Checked coupling code – a bug was causing direct feedback between strings and not between strings and body. Corrected this, and now coupling seems to be operating correctly.
- 15/02/16** Did CM3202 coursework. Added coupling between strings by analysis of the frequencies present in their output.
- 16/02/16** Began setting up a GitLab server to create a code repository. Plotted some string bowing parameters including velocity, slip state and friction between string and bow. Added noise at various points to see if mechanical nature of output can be altered, preliminarily this seems possible.
- 17/02/16** Attended supervisor meeting between 15:30 and 17:00. Discussed the progress so far and decided on direction of development. An optimiser would be constructed to operate on the d , pl and E parameters of the strings, and a solution for a system of linear equations representing transforms from raw string output to desired target sounds sought. Addressed some GitLab issues, some remain.
- 18/02/16** Tried adding limited digital noise to bow velocity and force directly; seems quite good. Converted the cello (multiple string) code into an object, and completed installation of GitLab – created a code repository.
- 19/02/16** Work suspended; family time.

20/02/16 Did CM3202 coursework.

21/02/16 Did CM3202 coursework.

22/02/16 Completed CM3202 coursework. Created comparison measure between two audio samples, using the average of differences between two constant-Q transforms.

23/02/16 Cleaned up comparison code, and checked it for correct operation. Constructed a function around this, and committed the change to GitLab.

24/02/16 Discovered better constant-Q transform library, and investigated it. Decided to convert my existing comparison code for its use, which worked well. Results now seem more accurate and various parameters can be customised (such as bins per octave etc.)

25/02/16 Developed the equations needed to maintain pitch (by altering tension) when changing a string's diameter, linear density or Young's modulus. Created first prototype optimisation routine, operating on a string's linear density. It functions to a degree, but is incorrect for some reason.

26/02/16 Discovered bug, and obtained fundamental of string correctly (ω_0 vector element 1 $/ (2\pi)$). Extended optimiser to operate on string's diameter and Young's modulus parameters also.

27/02/16 Set code to function using `fminsearch()`; run-time is extreme due to number of iterations and slowness of Matlab implementation. Attempted use of optimiser on the current body convolution vector – search space is too great. Tried using `fminsearchbnd()` to restrict space, left running overnight.

28/02/16 Tried using de-convolution to extract body audio data from real samples, not very successful – use of recovered data on simulator results in clearly erroneous output. Should investigate frequency domain instead? Optimiser run-time is better using `fminsearchbnd()`.

29/02/16 Used Matlab profiler to investigate run-time issues. Abandoned de-convolution in favour of a frequency domain approach to extraction of body and ambient information from real cello samples, then converting to time domain using `IFFT()` - seems to work well! Need to construct code to generate impulse responses from a set of these samples.

01/03/16 Nasty resonance is heard in output using this code, thought perhaps caused by lack of overall response in extracted frequency data. Tried interpolating between points, but no change. Constructed test code to examine this, and reduced ringing by manipulating the impulse response vector; but change did not work when added to simulator.

02/03/16 Discussed methods to extract impulse responses, and possible use of machine learning to solve body response. Optimiser returned a result! But clearly run-time is too great. Sent samples of current output to supervisor. Decided distortion present in output audio must be

addressed.

- 03/03/16** Discovered bug, causing ringing in impulse response; fixed. Created code to extract set of responses from a set of real cello sounds and raw string output, and code to allow blending of these responses when intermediate pitches are played. Seems very promising!
- 04/03/16** Implemented this code for the current simulation. Investigated the source of the distortion, and discovered it's not due to a sudden change in bow force or friction. Checked the original Artifastring code, discovered distortion is present within that model also, and the use of convolution and low-pass filtering appears to address is to a significant degree.
- 05/03/16** Tried using frequency domain transforms instead of time domain convolution – very slow execution, due to the need to match frequencies between string modes at each step. Will need to improve this code later. Added original convolution code to model to aid in step-wise convolution, and tried applying damping as a low-pass filter to strings. Works well to reduce distortion!
- 06/03/16** Developed equation for calculating pitch at a given finger position, and started added individual string damping vectors to each string. Tried using Matlab's parfor to parallelise code and speed up execution – did not work, execution was much slower!
- 07/03/16** Completed adding string damping vectors, and added sample frequency multiplier (so frequencies in excess of the Nyquist frequency could still be modelled, as in the original Artifastring). Generated new outputs as tests – results seem good!
- 08/03/16** Experiments with string damping showed a wide range of tonal variation was possible. Created a new optimiser routine working directly on the string's mode decay vectors. Works, but run-time is tremendous due to large search space. Cello simulation run-time must be addressed quickly.
- 09/03/16** Obtained second set of cello recordings to use as audio targets. Began vectorisation of string simulation code.
- 10/03/16** Continued vectorisation of string simulation code; approximately 60% done, initialisation code mostly converted.
- 11/03/16** Completed vectorisation of string simulation code. A speed-up of 6x was obtained! Considering generating a Mex executable from this code for a greater improvement.
- 12/03/16** Began vectorising other core code, and tested multiple simultaneous string simulations – unfortunately execution speed is now less than original (class-based) version! Perhaps further vectorisation is needed? Or the use of functions or global variables are impacting performance greatly?
- 13/03/16** Trialled several coding styles – the use of in-line code hugely increased performance, and interestingly inserting even a call to a function with no operation caused a tremendous degradation. Selected in-line style as best for project, rewrote convolution code and added disk buffering to support this.

- 14/03/16** Rewrote frequency domain analysis code (used in sharing energy between body and strings) to match the new style. Speed is much better! Now 1s of output consumes only 40s of computation time. Factorisation and in-lining of core code is complete.
- 15/03/16** Tested new code; convolution is not functioning correctly, and large changes in the tone of convolved audio is noticeable in comparison to the previous version. Investigation showed the fade times of source audio prior to convolution vector creation is critical to the response returned – have developed a workaround (fade time = cycle time), but clearly more work is needed here.
- 16/03/16** Continued work on optimiser, and discussed finalisation of project direction and sub-goals with supervisor. Modified the use of `fminsearch`, so that bigger steps may be taken initially (by adding a relatively large offset to `fminsearch()`'s parameters).
- 17/03/16** Tidied up existing code, and tried adding for loops within string engine code where appropriate (since Matlab's optimisation of for constructs appears to be very effective in this situation). It did not improve performance. GitLab failed, and required re-configuration! Optimiser is able to produce a reasonable number of iterations in a given time period now, but perhaps changes to its function are needed.
- 18/03/16** Added pseudo-natural variations for finger position, bow force and velocity using sine oscillators. Changed optimiser routine for more effective operating on string's mode decay vector. Early results seem odd, but left optimiser to run.
- 19/03/16** Results from optimiser not correct – found bug (wrong string's vector was being manipulated). Fixed this, and tested optimiser. Possible tonal range doesn't seem as great as I had hoped, so changed optimiser to operate on string damping vector – output seemed greatly improved! Uploaded changes to repository and left to run.
- 20/03/16** Discovered bug in how pre-computed files were indexed (value was truncated). Changed number of generations to <1500. Created a set of finger positions for quick access. Optimiser seems to be working well, albeit slowly! Discussed progress with supervisor.
- 21/03/16** Optimiser work continues – primary indications are that it's finding a good result, but very slowly, despite improved simulator performance. This could be a problem.
- 22/03/16** Attended supervisor meeting, and discussed other optimisation and performance improvement methods, and correction of frequency profile of convolved output. Started planning to alter code to operate simultaneously on all four strings via matrices. Optimiser still running; still very slow.
- 23/03/16** Began re-writing core code to use multi-string matrices, approximately 50% completed. Noticed `compute_bow()` has repeated instructions, so re-factored it to some $\frac{1}{4}$ of its original length!
- 24/03/16** Finished re-writing main parts of core code, began testing and debugging. A method is necessary to prevent routines from changing values linked to strings that are not altered under specific conditions; tried `repmat()` to resize a mask – preliminary tests suggest this might be too inefficient.

- 25/03/16** Completed masking using `repmat()` - performance is worse than original (class-based) code! This may be due to 1) every statement running (regardless of whether one or many strings use it), 2) use of `repmat()` results in large masks (4x96 in the case of the mode derivatives matrix). Also, NaN values 'slip' through the mask, since $\text{NaN} * 0 = \text{NaN}$. Reverted the change, so I can focus on the frequency correction issue, identify whether bottlenecks can be averted and try different optimiser methods.
- 26/03/16** Tried selecting only the centre portion of the results of FFTs within the code (to avoid artefacts caused by sudden transitions at sample endpoints) – seems to produce better results when tested within individual code sections. Also tried using longer samples, for greater FFT accuracy. Again, results look better, but performance of code using the resulting vectors is greatly degraded.
- 27/03/16** The new use of FFT did not improve the output of the engine in practice, and the accuracy of the transform was not improved – the match to the target samples was no closer than the previous (quicker) method. Reverted the changes, and continued running the optimiser.
- 28/03/16** Began adding a third dimension to the convolution vector generator, to represent the response of the body simulation to changes in amplitude (playing intensity) as well as changes in pitch. Investigated the use of `bsxfun()` instead of `repmat()` to enable masking for a matrix-based implementation.
- 29/03/16** Completed converting the convolution vector code for three-dimensions – the result seems good! Tried `bsxfun()`, performance was worse than when using `repmat()`. Also tried Mex version of masking code, but call overhead again degraded performance unacceptably, even when no operations were contained within the Mex executable! Added boundaries to the sine oscillators for smoother operation.
- 30/03/16** System hard-drive died on development machine! Began re-installing Gentoo and Matlab...
- 31/03/16** Completed restoring operation of development machine. Corrected issues with the comparison function (ignore phase (imaginary) component, took the logarithmic difference), results seem better. Attempted to correct the output of the body simulator in the frequency domain – the result was poor, either obliterating time domain effects or not matching the desired response at all.
- 01/04/16** Investigated a method for performing frequency correction in the time domain – added a new buffer to operate on. But correcting in a step-wise fashion using this buffer produces nasty artefacts, since a different correction is necessary at each step, and calculating one causes loss of time domain effects when it is applied! Turned attention to optimiser, created a curve generator (in the style of a graphic equaliser) to reduce the search space. Re-structured d, pl and E optimisation routine to use MCS optimiser, and began testing.
- 02/04/16** Discovered some bugs in my use of MCS (for example, input vectors needed transposing to match its scheme); fixed them. Left optimiser running, and investigated various audio manipulation techniques mentioned in DAFX. Tried manipulating phase and amplitude separately, or individually, but could not quickly find workable method of frequency correction. Decided a precise equalisation algorithm would work, but would be too complex computationally due to the resolution of the equalisation curve.

- 03/04/16** Discovered ringing in output of string 4. Tried developing code to detect this and remove the problem component from the frequency transform.
- 04/04/16** Solution only partially worked, and I discovered ringing at different frequencies, so instead investigated the source of the confirmed problem. Issue was discovered to be a slight difference in pitch between the audio samples used in the calculation of the frequency domain transform. Correcting this solved the problem, but it highlights how tenuous this method of transforming one sound to another is – more research is necessary.
- 05/04/16** Continued optimiser runs, and began producing simulation output ready for testing. Laptop power connector failed, so work was suspended whilst I replaced it. Researched statistical validation tests (chi-squared, binomial etc.), and began creating a mock-up of the subjective tests.
- 06/04/16** Designed subjective tests, and constructed software in Python to automate them. Added a basic vibrato oscillator to the simulation and edited available output samples ready for use by the test software.
- 07/04/16** Added a limit to the transitions between convolution vectors, so that sudden changes in amplitude do not cause changes in body response that occur quicker than possible in the real instrument body. Continued generating and preparing test samples, and added a second section to the test software so that participants can register their preference for the 'most real' audio. Set test software to retain all entered data.
- 08/04/16** Completed preparation of the test software, including all sounds and conversion of the script into an executable (for deployment on various systems). Arranged to meet with testers, aiming for at least 25 participants.
- 09/04/16** First day of subjective testing. 9 participants.
- 10/04/16** Second day of subjective testing. 6 participants
- 11/04/16** Third day of subjective testing. 7 participants.
- 12/04/16** Fourth day of subjective testing. 7 participants.
- 13/04/16** Final day of subjective testing. 4 participants. Constructed scripts to extract results and perform chi-squared and binomial distribution tests. Began analysing results, and planning for the final report.
- 14/04/16** Obtained results, and added glissando function to simulation as a demonstration. Began writing the final report.

Bibliography

- [1] Analogue Catalogue. 2016. *Analogue Catalogue Studio*. Available at: <http://www.analoguecat.com/rates-and-info/4586823156> [Accessed: 13/04/16].
- [2] Bilbao S. & Desvages C. 2014. *Physical Modeling of Nonlinear Player-String Interactions in Bowed String Sound Synthesis Using Finite Difference Methods*. Available at: <http://www.ness-music.eu/wp-content/uploads/2014/05/desvages.pdf> [Accessed: 27/01/16].
- [3] Brown J.C. 1991. *Calculation of a constant Q spectral transform*. Available at: <http://academics.wellesley.edu/Physics/brown/pubs/cq1stPaper.pdf> [Accessed: 16/04/16].
- [4] Cannon J. T. & Dostrovsky S. 1983. *The Evolution of Dynamics, Vibration Theory from 1687 to 1742*. Available at: <http://homes.chass.utoronto.ca/~cfraser/vibration.pdf> [Accessed: 13/04/16].
- [5] D'Errico J. 2012. *fminsearchbnd, fminsearchcon*. Available at: <http://www.mathworks.com/matlabcentral/fileexchange/8277-fminsearchbnd--fminsearchcon> [Accessed: 13/04/16].
- [6] Demoucron M. 2008. *On the control of virtual violins - Physical modelling and control of bowed string instruments*. Available at: <https://hal.archives-ouvertes.fr/tel-00349920/document> [Accessed: 13/04/16].
- [7] Fletcher N.H. & Rossing T.D. 1998. *The Physics of Musical Instruments*, Springer-Verlag, New York.
- [8] Hanselman D. & Littlefield B. 2005. *Mastering MATLAB 7*. Upper Saddle River, N.J.: Pearson Prentice Hall.
- [9] Helmholtz H. 1895. *On the sensations of Tone as a physiological basis for the Theory of Music*, Longmans, Green & co., London, New York.
- [10] Image Line 2016. *Drumaxx*. Available at: <http://www.image-line.com/plugins/Synths/drumaxx/> [Accessed: 13/04/16].
- [11] Klapuri A. & Schörkhuber C. 2010. *Constant-Q Transform Toolbox for Music Processing*. Available at: http://iem.kug.ac.at/fileadmin/media/iem/projects/2010/smc10_schoerkhuber.pdf [Accessed: 13/04/16].
- [12] Lewin W. 2004. *Lec 11: Fourier Analysis, Time Evolution of Pulses*. Available at: <http://youtube.com/watch?v=U58c-BRfwqg> [Accessed: 05/02/16].
- [13] London Cellist, The. 2016. Available at: <http://www.warble-entertainment.com/the-london-cellist> [Accessed: 13/04/16].
- [14] Manuel 2014. *Cello impulse responses*. Available at: <http://www.fiddleforum.com/fiddleforum/index.php?topic=36535.0> [Accessed: 13/04/16].
- [15] Mathworks 2016. *Matlab 2016a: Techniques to Improve Performance*. Available at: http://uk.mathworks.com/help/matlab/matlab_prog/techniques-for-improving-performance.html [Accessed: 13/04/16].

- [16] McIntyre M.E. & Woodhouse J 1979. 'On the fundamentals of Bowed-String Dynamics', *Acustica*, vol. 34, no. 2, pp. 93-108.
- [17] Native Instruments 2011. *Komplete: Samplers: Kontakt 5*. Available at: <http://www.native-instruments.com/en/products/komplete/samplers/kontakt-5/> [Accessed: 13/04/16].
- [18] Nelson M. 2009. *myBinomTest(s,n,p,Sided)*. Available at: <http://www.mathworks.com/matlabcentral/fileexchange/24813-mybinomtest-s-n-p-sided-> [Accessed: 04/05/16].
- [19] Neumaier A. 2000. *MCS: Global Optimization by Multilevel Coordinate Search*. Available at: <http://www.mat.univie.ac.at/~neum/software/mcs/> [Accessed: 16/04/16].
- [20] Percival G. 2013. *Artifastring: a highly optimized physical simulation of a violin for sound synthesis*. Available at: <http://percival-music.ca/artifastring/> [Accessed: 13/04/16].
- [21] Percival G. et. al 2011. *Physical Modelling meets Machine Learning: Teaching Bow Control to a Virtual Violinist*. Available at: <http://percival-music.ca/research/percival-smc2011.pdf> [Accessed: 13/04/16].
- [22] Pianoteq 2016. *Pianoteq – Pianoteq 5*. Available at: <https://www.pianoteq.com/pianoteq5> [Accessed: 13/04/16].
- [23] Press H. et al. 2007. *Numerical recipes: the art of scientific computing*. 3rd ed. Cambridge; New York: Cambridge University Press.
- [24] Sample Modelling 2016. *Sample Modelling: SWAM platform*. Available at: http://www.samplemodeling.com/en/swam_saxophones.php [Accessed: 13/04/16].
- [25] Schwarzer & Mantik GmbH 2016. *The Berlin Series: Berlin Strings*. Available at: http://www.orchestraltools.com/libraries/berlin_strings.php [Accessed: 13/04/16].
- [26] Smith III J.O. 2010. *Virtual Acoustic Musical Instruments: Review and Update*. Available at: <http://ccrma.stanford.edu/~jos/jnmr/jnmr.pdf> [Accessed: 13/04/16]
- [27] Vedenyov M. 2012. *Vibrating String Simulator*. Available at: <http://www.mathworks.com/matlabcentral/fileexchange/35746-vibrating-string-simulator> [Accessed: 13/04/16].
- [28] Vienna Symphonic Library 2016. *Instruments: Strings Complete: Solo Strings Bundle*. Available at: https://vsl.co.at/en/Strings_Complete/Solo_Strings_Bundle [Accessed: 16/04/16].
- [29] Vienna Symphonic Library 2016. *Software: Vienna Instruments Sample Player Software*. Available at: https://vsl.co.at/en/Software/Vienna_Instruments [Accessed: 16/04/16].
- [30] Wallander Instruments 2016. *Wallander Instruments – Modelled Virtual Instruments in VST & AU Format*. Available at: <http://www.wallanderinstruments.com/?mode=products&lang=en> [Accessed: 16/04/16].
- [31] Zölzer U. 2011. *DAFX: digital audio effects*. 2nd ed. Chichester: Wiley.