ECS732 Final Assignment: Textile FM Synthesis

Chris Yeoward

Introduction

Whilst the range of sonic capability in electronic instruments far outreaches that of any acoustic instrument, they have long suffered from a difficulty in expression, with the keyboard on many synthesisers only providing control in 3 dimensions: pitch, amplitude, and a customisable modulation wheel. Whilst this is adequate for some situations, this can often be a limitation when faced with the vast array of parameters that electronic instruments offer. One such form of overwhelmingly parameterised synthesis is frequency modulation, with the original instruments, such as Yamaha's DX7, being notoriously complex to program.

As such, various forms of multi-dimensional interfaces have found application in the musical domain. In [1], a passive pickup matrix was used to control physical modelling synthesis, and in [5], a generic eTextile was created for use in musical performance.

In this assignment I created a single-touch expressive frequency modulated synthesiser, using two 3D matrix sensors to control the pitch and amplitude of the carrier and modulator waves. Subsequently, I added a ping pong delay to the system to increase the width of the sound generated.

For the matrix sensor implementation, I followed a method similar to that of [5].

Implementation

This project uses the Bela board [2] to process the sensor input and compute the sound output. The board features 8 analog pins, so without the use of multiplexers the system is limited to grids of 4x4 for each sensor. The sensors use piezo-resistive foam, sandwiched between two orthogonal sets of conductive strips, with one set of electrodes providing the current using digital output pins, and the other reading it into the board via the analog pins. The foam becomes less resistive as it is compressed, permitting a continual increase in current. As described in [5], the method for finding the displacement along the input axis (in this case the vertical, y, axis) involves triggering a different digital output pin each frame, reading the analog values, and gradually constructing a representation for the sensor state. In this way the sample rate for a *matrix frame* is lower than both digital and analog sample rates, by a factor equivalent to the dimensions of the matrix.

Matrix Sensors

Before constructing the first sensor, it was necessary to determine the optimum spacing between the strips, which is derived from the characteristics of the resistive foam. How close a touch would need to be to an input strip to produce a current was measured by placing an input copper tape as an anode beneath the foam, and positioning another above for the cathode, perpendicularly to the first.

```
bool setup(BelaContext *context, void *userData)
{
  pinMode(context, 0, P8_07, OUTPUT);
  gAudioFramesPerAnalogFrame = context->audioFrames
      / context->analogFrames;
  return true;
}

void render(BelaContext *context, void *userData) {
  digitalWrite(context, 0, P8_07, GPIO_HIGH);
  for(unsigned int n = 0; n < context->audioFrames;
      n++) {
    if(n % gAudioFramesPerAnalogFrame == 0) {
      int analogFrame =
          n/gAudioFramesPerAnalogFrame;
      float input = analogRead(context,
          analogFrame, 0);
      rt_printf("%f", input);
    }
  }
}
```

Calculating `gAudioFramesPerAnalogFrame` ensures that the correct analog frame is used, irrespective of the analog or audio sampling rates.

Measurements of the input current as a function of displacement from the centre of the bottom anode were taken, moving along the cathode strip. As shown in I, no current is transmitted until approximately 23.5mm from the centre of the strip, corresponding to a distance of 15mm from the edge.

Distance (mm)	Value
Inf	0.05
28.5	0.05
23.5	0.3
18.5	0.45
13.5	0.57
8.5	0.6
0	0.7

TABLE I: Resistive foam calibration values.

In choosing a spacing between the strips, there exists a balance between responsiveness and size of matrix. Ideally, they would be as far apart as possible to enable the greatest control, but not large enough to produce dips in the amplitude, or sections of continually constant amplitude. As such, the inter-strip distance was chosen as 5mm, theoretically wide enough to produce a current in neighbouring strips when pressing directly down on any given electrode. Full construction of the sensors is detailed in appendix A. The circuit diagram is shown in figure 1.

Once the first matrix sensor had been constructed, the first stage was to determine the position of a touch along an axis. Starting with the simpler case of the analog input (x) axis, the first two anodes were connected to the first two analog input

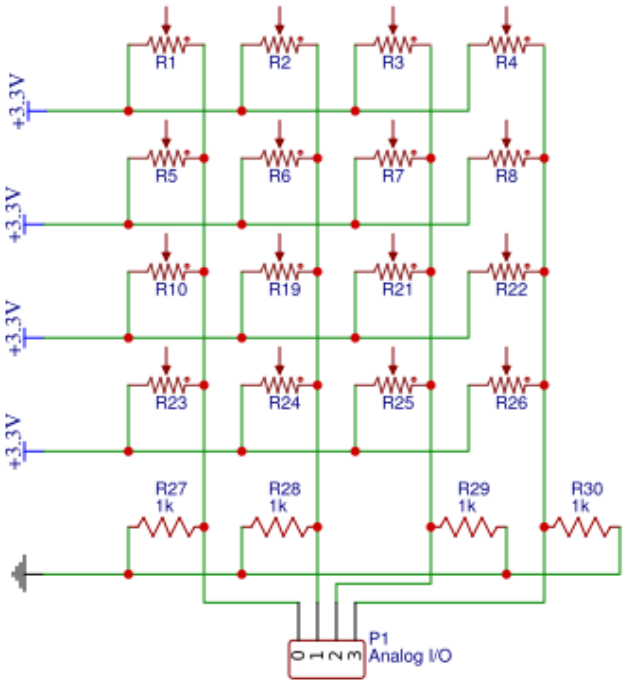

Fig. 1: Circuit diagram for a matrix sensor. Each row provides a current triggered by a GPIO output pin. The row that is active changes with each analog frame, such that only one row is active at a time. The current is received by a set of anodes that read into the analog pins on the cape.

pins, and used the same cathode strip as before to produce a constant current:

```
#define M_DIMENSION 2

bool setup(BelaContext *context, void *userData) {
  ...
}
void render(BelaContext *context, void *userData) {
  ...
  float input[M_DIMENSION];
  input[0] = analogRead(context, analogFrame, 0);
  input[1] = analogRead(context, analogFrame, 1);
  ...
}
```

`setup()` stays the same, but input is replaced with an array. Quadratic interpolation was used to approximate the position of a touch between the two sensors. Often used to provide increased frequency resolution between spectral bins [6], quadratic interpolation uses neighbouring values to fit a curve around a maximal peak. The formula for determining the displacement x_p from the nearest peak value x_{max} is given by [6]:

$$x_p = \frac{1}{2}\frac{\alpha - \gamma}{\alpha - 2\beta + \gamma} \tag{1}$$

$$x_p \in [-0.5, 0.5]$$

where $\alpha = y(x_{max} - 1)$, $\beta = y(x_{max})$, and $\gamma = y(x_{max} + 1)$. The interpolated position of the peak is then $x_{max} + x_p$.

Giving consideration to the matrix boundaries, the value of a virtual node existing beyond the matrix was fixed to 0 for any node situated along an edge. In this way, the method for determining the position between two anodes was to find the greatest value in matrix, find the values for the neighbouring nodes on either side if they existed, or set to 0 otherwise:

```
float interpolatePosition(float alpha, float beta,
    float gamma) {
  return (alpha - gamma) / (2 * (gamma - 2*beta +
      alpha));
}

void render(BelaContext *context, void *userData) {
  ...
  int maxNode = 0;
  if(input[1] > input[0]) maxNode = 1;
  float alpha, beta, gamma;
  alpha = maxNode == 0 ? 0.0 : input[maxNode - 1];
  beta = input[maxNode];
  gamma = maxNode >= (M_DIMENSION - 1) ? 0.0 :
      input[maxNode + 1];
  float x = (float) maxNode +
      interpolatePosition(alpha, beta, gamma);
  ...
}
```

This provided a continuous calculation of the position of a touch when moving from one anode to the other.

The next step was to expand the 2x1 grid to 2x2, and track the position along the y axis. Following the method of [5], for each analog frame a different cathode is triggered by a digital output pin. The analog values are read in, and the frame is iteratively constructed until all the digital outputs have been triggered. In the simplest case of a 2x2, this involves merely switching between the two digital pins, turning them on as required. For this, the Bela function `digitalWriteOnce()`, that sets the value of the specified digital pin for one digital frame only, was most suitable. From the Bela documentation [3], this is a faster operation than `digitalWrite()`, and is better suited to this use case as each digital pin should stay active for one analog frame, at most.

```
...

float gMatrixInput[M_DIMENSION][M_DIMENSION] = {};
int gDigitalOutputPins[M_DIMENSION] = {P8_07,
    P8_08};

void render(BelaContext *context, void *userData){
  for(unsigned int n = 0; n < context->audioFrames;
      n++) {
    digitalWriteOnce(context, n,
        gDigitalOutputPins[gCurrentActiveDigitalOutputRow],
        GPIO_HIGH);
    if(n % gAudioFramesPerAnalogFrame == 0) {
      for (int column = 0; column < M_DIMENSION;
          column++) {
        int row = analogInputPins[column] == 6 ||
            analogInputPins[column] == 7 ?
            (currentRow + 1) % M_DIMENSION :
            currentRow;
        gMatrixInput[column][gCurrentActiveDigitalOutputRow]
            = analogRead(context, analogFrame,
            column);
        if(gMatrixInput[column][gCurrentActiveDigitalOutputRo
            > gCurrentPressureInFrame) {
          gCurrentPressureInFrame =
              gMatrixInput[column][gCurrentActiveDigitalOutpu
          gActiveNodeX = column;
          gActiveNodeX = row;
        }
      }
      gCurrentActiveDigitalOutputRow++;
```

```
      if(gCurrentActiveDigitalOutputRow >=
          M_DIMENSION) {
        // process gMatrixInput
        gCurrentPressureInFrame = 0;
        gCurrentActiveDigitalOutputRow = 0;
      }
    }
  }
}
```

The variables `gCurrentPressureInFrame`, `gActiveNodeX` and `gActiveNodeX` were introduced to retain information as to which node in x and y holds the current maximum value (`gCurrentPressureInFrame`). By making the variables global, then the matrix dimension and frame size is not coupled to the render block size, should it ever change.

Note that placing the command `digitalWriteOnce()` outside of the analog frame processing block, in line 9, is essential. Whilst the command is fast, if writing to the digital pin was left until the required analog frame, then there is no guarantee that sufficient current would be produced and read by the analog inputs in the duration of one digital frame, and undefined behaviour will occur. In this way, the analog value is read, and `gCurrentActiveDigitalOutputRow` is incremented. `digitalWriteOnce()` then activates the next GPIO pin in the following two digital frames, the second coinciding with the next analog frame (figure 2).

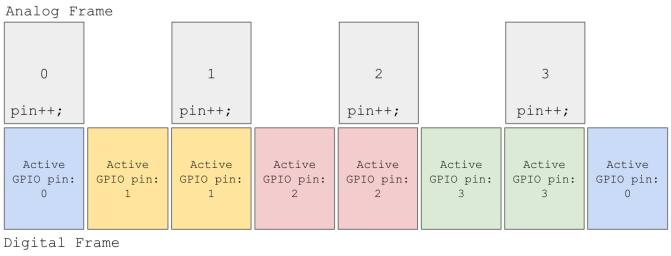

Fig. 2: Sequence of GPIO pins, given an analog sampling rate half that of the digital.

The global variable `gCurrentActiveDigitalOutputRow` retains the state between frames of which matrix index to write to. Once all the samples for the matrix have been collected, then the matrix frame is processed, and the matrix sampling starts again. In implementing this, a bug was discovered: whilst the second cathode, or row, worked as expected, pressing down on the first, would produce a reading in both. Further investigation determined that the root cause of this was that the first cathode was never being set to 0, instead permanently producing a current. Thus the problem was solved by explicitly initialising each of the digital output rows at the start of each `render()` cycle:

```
void render(BelaContext *context, void *userData){
for (int pin = 0; pin < M_DIMENSION; pin++) {
  digitalWrite(context, 0, gDigitalOutputPins[pin],
      GPIO_LOW);
}
...
}
```

A likely cause of this is that the call to `digitalWriteOnce()` in the final digital frame iteration (i.e. when `n = context->audioFrames`) was actually setting the relevant digital pin permanently, similar to `digitalWrite()`, rather than for that final frame only.

With the two cathodes reading values correctly, and independently, an equivalent interpolation is applied to the y axis to find the position between electrodes, such that the position of any touch on the 2x2 matrix would be registered.

From this implementation, scaling up to 4x4 merely required connecting up the remaining electrodes to the appropriate digital out and analog input pins. The matrix position calculation was abstracted into a `MatrixSensor` class that is initialised with array specifying the analog input pins to use, and holds a reference to that array in the class instance. It incrementally takes samples of these pins, before rendering a point in normalised 3D space using an ancillary `Point` class, such that:

$$x, y, z \in [0, 1]$$

where Z corresponds to the depth or pressure of the touch, in this implementation simply the maximum value in the matrix, although quadratic interpolation can also be used to estimate the true peak value. As the algorithm relies on identifying the matrix position receiving the greatest current to determine the x,y position of the touch, the calculated x,y position can fluctuate while $Z \ll 1$. To reduce the effect of noise on the rendered position, a lowpass filter with a cutoff of 50Hz was applied to each of the dimensions. This cutoff was high enough to be reactive to any change in touch, but low enough to remove the majority of the high frequency noise.

The lowpass filter itself is a second order Butterworth filter, with coefficients calculated on `setup()`.

Carrier Oscillator

Adding an oscillator class created the first half of the synthesis engine, and enabled testing of the matrix sensor:

```
//Oscillator.cpp
#include <Oscillator.h>

void Oscillator::setup(float audioSampleRate) {
  sampleRate = audioSampleRate;
};

float Oscillator::getNextSample() {
  float output = amplitude * sinf(phase);
  phase += 2.0 * M_PI * frequency / sampleRate;
  if(phase > 2.0 * M_PI) phase -= 2.0 * M_PI;
  return output;
};

void Oscillator::setFrequency(float frequency) {
  this->frequency = frequency;
};

void Oscillator::setAmplitude(float amplitude) {
  this->amplitude = amplitude;
};
```

The class instance holds the oscillator's phase value, rendering the next sample based on the phase increment expected for the given sample rate and frequency. This provided a simple interface with which to interact in `render()`, using the x and y coordinates of the matrix sensor to modulate the frequency and amplitude respectively:

```
float gCarrierAmplitude = 0.0;
float gCarrierTuning = 220.0;
float gCarrierFrequency = gCarrierTuning;

LowPassFilter carrierAmplitudeEnvFilter;

void render(BelaContext *context, void *userData){
...

Point carrierMatrixPosition =
    carrierMatrix.readPosition();

if(carrierMatrixPosition.z > PRESSURE_THRESHOLD) {
  gCarrierAmplitude = carrierMatrixPosition.y;
  gCarrierFrequency = gCarrierTuning * (pow(2,
      carrierMatrixPosition.x));
} else {
  gCarrierAmplitude = 0;
}
gCarrierAmplitude =
    carrierAmplitudeEnvFilter.applyTo(gCarrierAmplitude);

carrierOsc.setAmplitude(gCarrierAmplitude);
carrierOsc.setFrequency(gCarrierFrequency);

float synthOut = carrierOsc.getNextSample();

for(unsigned int channel = 0; channel <
    context->audioOutChannels; channel++) {
audioWrite(context, n, channel, synthOut);
};
}
```

For personal preference, the synth's range was set to an octave, from a base tuning $F_{carrier}$ of 220Hz, which required increasing the tuning frequency by a factor of 2, such that:

$$f_{carrier}(x) = F_{carrier}2^x$$

As mentioned previously, the x and y coordinates fluctuate when no touch is applied to the sensor. A threshold pressure was therefore applied (of 0.1, based on observed values) above which to trigger the sound. The note onset and offset generated an audible click, as expected as the attack and release times of the amplitude envelope were effectively 0, thus necessitating the presence of high frequency content. An envelope was desired that would feature an attack and release quick enough not to render an audible click, with a constant sustain, such that the synth was responsive to the touch but sonically free of artefacts. With these characteristics in mind, the simplest solution was to apply another Low Pass filter to the oscillator amplitude, with a cutoff low enough to produce the desired attack and release behaviour. Conversely, the frequency did not require a lowpass as no click would be present due to using phase to calculate the oscillator output.

Modulator sensor

With the carrier wave controlled by the first sensor, the next step was to modulate the source signal with a second oscillator, controlled by another textile sensor. Given the MatrixSensor abstraction, the code addition was theoretically trivial once the sensor had been connected to the required pins: 4,5,6,7 for the analog in, and P8_27, P8_28, P8_29 and P8_30 for the digital out. After connecting the sensor, two things were noted from the input readings: firstly, the matrix values read for pins 6 and 7 appeared to be delayed by an analog frame and out of sync with the digital outputs, such that a touch on the second cathode registered as a touch on the first. Secondly, the value for analog input 6 seemed to be notably lower than other analog input pins.

Regarding the first issue, the Bela docs [3] imply that the samples for analog inputs 6 and 7 are two samples older than the current frame. Whilst not ideal, the solution was therefore was to offset the matrix values by one when reading analog 6 or 7:

```
void MatrixSensor::takeNextSample(int analogFrame,
    int currentRow) {
  for (int column = 0; column < M_DIMENSION;
      column++) {
    int row = analogInputPins[column] == 6 ||
        analogInputPins[column] == 7 ? (currentRow
        + 1) % M_DIMENSION : currentRow;
    matrixValues[column][row] =
        map(analogRead(belaContext, analogFrame,
        analogInputPins[column]),
        inputMinimums[column],
        inputMaximums[column], 0, 1);
    if(matrixValues[column][row] >
        currentPressureInFrame) {
      currentPressureInFrame =
          matrixValues[column][row];
      activeNodeX = column;
      activeNodeY = row;
    }
  }
};
```

To solve the second issue, a simple calibration was implemented, that also gave more consistent behaviour across the electrodes on both sensors. The algorithm measures the minimal values of the sensor at rest first, by measuring the maximum values of each of the inputs. Once the given time has elapsed, in this instance 5s is given for both stages, a firm touch is applied directly to each cathode and the maxima are measured by again reading the peak values for each of the inputs. Once the calibration had finished, the values for each input were mapped from their minimum and maximum to be between 0 and 1. The minima finding method that starts values at 1, then decrements, is not suitable in this case, as for the first render cycle the analog input values are 0, thus also setting all the minima to 0, rather than their inactive values.

```
void MatrixSensor::calibrate(int analogFrame){
  switch(calState) {
    case kInit:
    rt_printf("Calibrating minima, do nothing...
        \n");
    calState = kMin;

    case kMin:
    for (int column = 0; column < M_DIMENSION;
        column++) {
      float input;
      input = analogRead(belaContext, analogFrame,
          analogInputPins[column]);
      if (input > inputMinimums[column]) {
        inputMinimums[column] = input;
      }
    }
    calibrationCount++;
    if(calibrationCount >= calibrationDuration) {
      rt_printf("Finished calibrating minima, now
          push down to calibrate maxima...\n");
      calState = kMax;
      calibrationCount = 0;
    }
```

```
    break;

    case kMax:
    for (int column = 0; column < M_DIMENSION;
        column++) {
      float input;
      input = analogRead(belaContext, analogFrame,
          analogInputPins[column]);
      if (input > inputMaximums[column]) {
        inputMaximums[column] = input;
      }
    }
    calibrationCount++;
    if(calibrationCount >= calibrationDuration) {
      rt_printf("Finished calibrating!\n");
      calState = kFinished;
    }
    break;

    case kFinished:
    break;
  }
};

void MatrixSensor::takeNextSample(int analogFrame,
    int currentRow) {
  ...
  matrixValues[column][row] =
      map(analogRead(belaContext, analogFrame,
      analogInputPins[column]),
      inputMinimums[column], inputMaximums[column],
      0, 1);
  ...
};
```

The calibration implementation then required that the matrix class hold calibration state, such that the instance could be queried about whether or not it had finishing calibrating. It follows a simple state machine sequence, shown in figure 3.

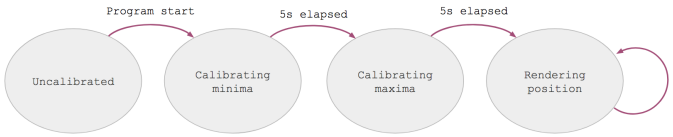

Fig. 3: Calibration state sequence of a matrix sensor

This calibration routine is not the most advanced, and could be improved by taking maximum and minimum values for each of the nodes in the matrix. This would, however, require an operator to push down on each node of the matrix in turn during calibration. For the sake of a faster development cycle, the calibration reading was limited to only the first row, applying the mapping globally to all other rows, rather than calibrating each node individually.

When implementing the calibration of the two sensors in `render()`, the routine followed a similar state sequence to the matrix calibration, first calibrating the carrier matrix, followed by the modulator, before entering the permanent state of rendering the touch positions.

Frequency Modulation

With the second matrix sensor reading values correctly, the task remained to use the sensor to modulate the carrier wave. Using the oscillator class, a second signal generator was created, much like the first, that modulated the frequency of the carrier wave, such that:

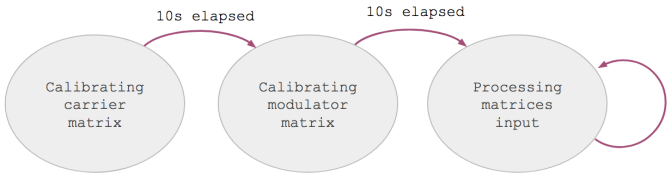

Fig. 4: Calibration state sequence of the two matrix system

$$y(n) = A_c sin(\omega_c n + A_m sin(\omega_m n))$$

where A_c and ω_c are the carrier amplitude and frequency, and A_m and ω_m that of the modulator.

Frequency modulation was chosen, rather than amplitude modulation, as the resultant spectrum is richer in harmonics, and able to give a notably brighter sound from a sine carrier wave [8]. Indeed, frequency modulation creates an infinite number of sidebands, the magnitude of which depends on the modulation index β:

$$\beta = \frac{A_m}{A_c}$$

Thus, increasing the modulation index increases the amplitude of the sidebands, while the modulation frequency determines their spacing[8].

Again, the x axis of the sensor controls the frequency of the modulator oscillator. In this case however, the frequency is a ratio of the carrier, such that when the frequency of the carrier changes, the modulator frequency changes also. In this way, the modulator can be seen to effectively control the timbre of the note played. The frequency range was scaled, such that at the centre point of the x axis the modulator is the same frequency as the carrier:

```
void render(BelaContext *context, void *userData){
...
Point modulatorMatrixPosition =
    modulatorMatrix.readPosition();
gModulatorFrequency = gCarrierFrequency *
    (modulatorMatrixPosition.x * 2);
gModulatorAmplitude = (gCarrierFrequency/2) *
    modulatorMatrixPosition.z;
modulatorOsc.setAmplitude(gModulatorAmplitude);
modulatorOsc.setFrequency(gModulatorFrequency);
gCarrierFrequency = gCarrierFrequency +
    modulatorOsc.getNextSample();
carrierOsc.setAmplitude(gCarrierAmplitude);
carrierOsc.setFrequency(gCarrierFrequency);
...
}
```

Delay

To widen the sound, a ping pong delay was added to the output. Ping pong is a common stereo audio effect, in which the echoed signal alternates between the left and right channels, and is a simple way to widen the sound[7]. The implementation requires a circular buffer to hold the delayed samples, one for each channel. The buffers are given an arbitrary length of 44100 samples, corresponding to one second of audio for the default sampling rate.

```
void Delay::setup(float audioSampleRate, float
    delayLengthS) {
  delayLength = delayLengthS * audioSampleRate;
  readPointer[L] = ((writePointer[L] +
      DELAY_BUF_LEN) - delayLength) % DELAY_BUF_LEN;
  readPointer[R] = ((writePointer[R] +
      DELAY_BUF_LEN) - delayLength) % DELAY_BUF_LEN;
};

float Delay::render(int channel, float input) {
  switch(channel) {
    case L:
    return renderLeft(input);
    break;

    case R:
    return renderRight();
    break;

    default:
    return 0.0;
  }
}

float Delay::renderLeft(float inputSample) {
  float delayedSample = buffer[L][readPointer[L]];
  buffer[L][writePointer[L]] = inputSample;
  buffer[R][writePointer[R]] = delayedSample;
  readPointer[L]++;
  readPointer[L] %= DELAY_BUF_LEN;
  writePointer[R]++;
  writePointer[R] %= DELAY_BUF_LEN;
  return delayedSample;
};

float Delay::renderRight() {
  float delayedSample = buffer[R][readPointer[R]];
  buffer[L][writePointer[L]] += feedbackGain *
      delayedSample;
  readPointer[R]++;
  readPointer[R] %= DELAY_BUF_LEN;
  writePointer[L]++;
  writePointer[L] %= DELAY_BUF_LEN;
  return delayedSample;
};
```

The implementation uses read and write pointers that render the next sample, and store the current sample respectively. The read pointer is delayed by the number of samples corresponding to the delay time, such that for any given sample stored at the write pointer position, the read pointer will reach it once the once the delay time has passed, in samples. In the implementation, a sample is firstly read from the left buffer, at the position of the read pointer. This sample is used as the output of the delayed sample, but is also placed in the buffer of the right channel, at the position of the right channel's write pointer. This ensures that once the delay time has passed again, the same sample will be read out to the right channel. When reading the samples in the right channel, the output is similarly placed back into the left buffer, though this time with a feedback factor. If this is set to 1, then the delay will never decay, and the system will distort. Set to 0, and the delay will only occur once. The write pointers are only incremented once both channels are finished using them at their current position.

To control the delay, and to provide and more natural interface with the sensors, the amplitude of the oscillations was modified to be controlled instead by the z axis, or touch pressure, and the dry/wet mix of the delay controlled by the y axis of the carrier matrix. In this implementation, the dry/wet control is closer to a send channel, or audio bus, rather than true dry/wet. This allows the delay to continue decaying even when the wet mix is set to 0.

Evaluation

As the system is composed of several components, analysis can be completed in three stages: firstly testing the matrix sensors, then the signal generation and frequency modulation, and finally the ping pong delay.

Matrix sensors

Figure 5 shows the matrix values for a firm touch located at position $(3, 3)$ on the modulator matrix. Whilst the matrix registers the maximum at the correct touch point, the surrounding matrix values can be seen to vary. The point at $(2, 3)$ is considerably lower than $(3, 2)$, reading a value of -0.024358 compared to 0.226570. Equally, at 0.127263 $(3, 4)$ is greater than $(4, 3)$ which reads -0.012807. This could be caused by calibration error, as previously stated the calibration is performed using only the first row $(1y)$, and applied to all other rows. More likely, however, is that the main contribution to this is caused by the difference in materials and spacing between the top and bottom electrodes. The conductive fabric used for the top layer has thinner electrodes, spaced further apart, than the copper tape electrodes on the bottom. The touch positioning algorithm would benefit from positive readings appearing on each neighbouring node, such that any change in the touch's position affects the calculation.

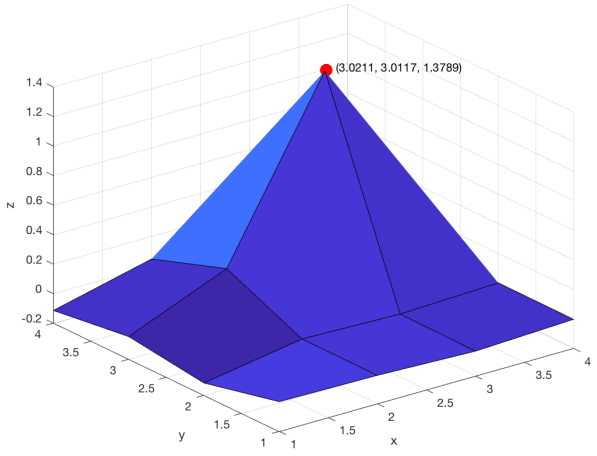

Fig. 5: Matrix values of a firm touch applied at 3,3.

Figures 6 and 7 show the calculated position along each axis for the two matrices. The measurements were taken 2mm apart, starting at the first edge of the first electrode, along the length of the axis. Whilst there are variations between the axes, the notable characteristic of all 4 data sets is the stepped appearance, where for considerable distances the calculated position does not change much, only to leap to the next node. Indeed, interaction with the sensor confirms this. While continuous frequency modulation is possible with one finger,

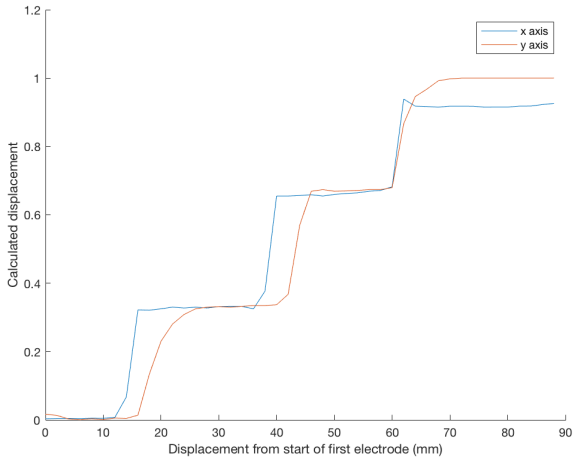

Fig. 6: Measure of the calculated position for the carrier matrix, as a function of distance from the start of the first electrode. X axis readings were taken along $y = 2$, y along $x = 3$.

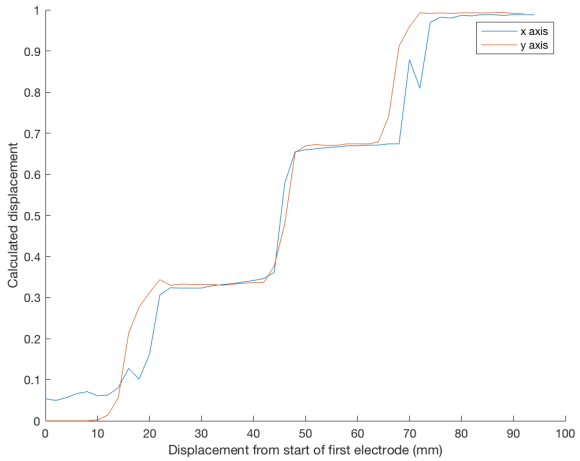

Fig. 7: Measure of the calculated position for the modulator matrix, as a function of distance from the start of the first electrode. X axis readings were taken along $y = 3$, y along $x = 2$.

it requires care and tentatively small movements. It is easier to use two fingers, controlling two electrodes directly, however this then makes operation more cumbersome. Thus the jumps are not discontinuous, merely very steep, and vary for the different axes. For example, the y axis of the modulator matrix exhibits a more gradual transition between steps. Given the differences in construction between the two axis, this could be expected, although the difference is not so pronounced in the modulator matrix. Additionally, the x axis of the carrier matrix never reaches 1, indicating that the there is permanently a value being read for the third anode.

The underlying issue regarding the stepped calculation could result from two possible factors. Firstly, as described above, the construction of the matrices could be at fault. Thinner copper tape on the bottom plane could be used, in conjunction with a closer spacing between the electrodes. This would ensure that the distance over which the activated electrode and its neighbours read a constant voltage would be minimised, though at the cost of reducing the size of the playable area.

The other cause could be the interpolation method. Several optimisations for quadratic interpolation exist, particularly in the domain of spectral processing[10]. Aside from using quadratic interpolation, blob detection in image processing frequently uses centroid calculation[11] to determine the centre of a set of pixels and could be used as an alternative.

Signal generation and frequency modulation

Figures 8 and 9 show the spectra for a single sine wave at 220 Hz, and for when the frequency is modulated at the 220 Hz with varying modulation indices.

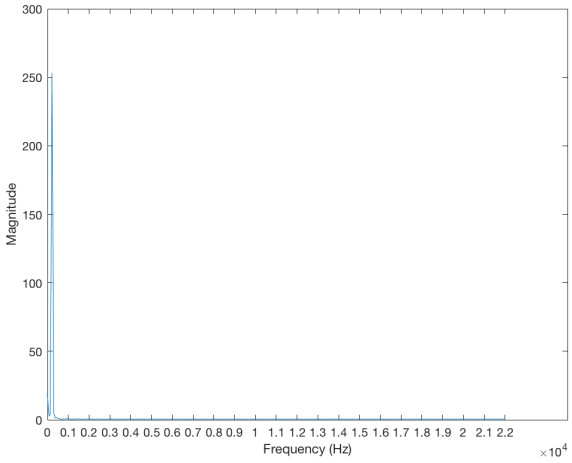

Fig. 8: Spectrum of a 220Hz sine wave

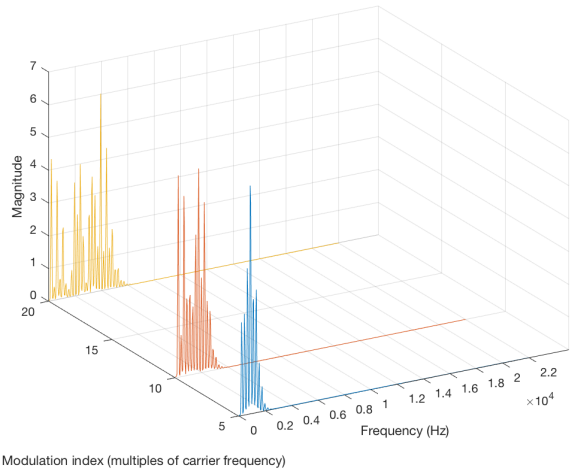

Fig. 9: Spectrum of a 220Hz sine wave, frequency modulated with different modulation indices. The modulation index is a function of the carrier frequency, such that it remains constant whilst the carrier frequency changes.

In FM synthesis sidebands appear in pairs, either side of the carrier wave, such that the first pair will appear at $f_c \pm f_m$[8]. When modulating the carrier wave by a signal that has the same frequency, these sidebands quickly move into the negative frequency range, and are then reflected back into the positive frequency range with phase inversion (as $sin(-\theta) = -sin(\theta)$)[12], causing interference. In analog synthesis, this can be countered by applying a high-pass filter to the incoming signal.

Increasing the modulation index can clearly be seen to increase the amplitudes of the sidebands, and indeed the presence of sidebands at higher frequencies, thus adding brightness. The sidebands are spaced at distances equal to the modulator frequency, in this case the same as the carrier frequency, with peaks of different amplitudes. As is characteristic of FM synthesis, the carrier frequency loses its presence as β increases [12], with the amplitudes of each sidebands defined by Bessel functions [9]. A general rule is that there will be approximately $\beta + 1$ sidebands audible [12], and the spectra roughly agree with this. 10 shows how the peak spacing spacing increases with increasing f_m, as expected.

Another artifact is that of aliasing, more visible in figures 10 and 11, which demonstrate how the spectrum changes for a carrier of 10kHz. However, both figures exhibit sidebands at multiple points, and not only at the expected frequencies of 0Hz and 20kHz. This is due to the higher order partials existing above the Nyquist rate, aliasing, and reflecting down into the audible spectrum. Similarly to the negative frequencies, a lowpass filter is often applied to analog signals before sampling. In the digital domain, however, this is often countered by limiting the maximum modulation index value [12].

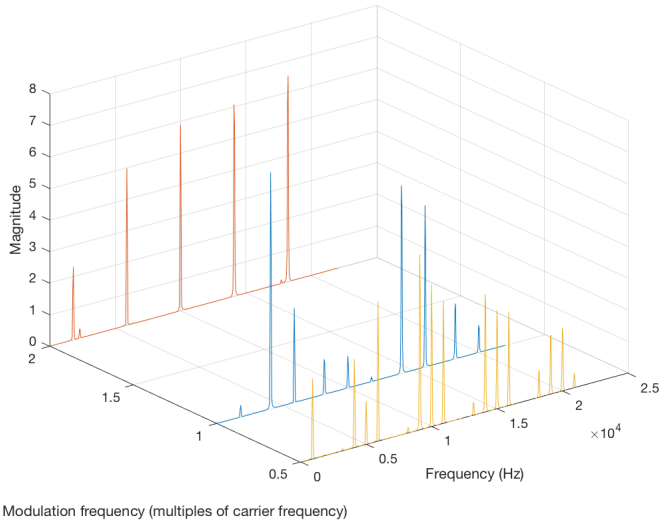

Fig. 10: Spectrum of a 10000Hz sine wave, frequency modulated with different modulation frequencies.

The infinite number of sidebands can create unwanted artefacts in the signal. Either through reflection from above the Nyquist frequency, or in the opposite direction, as sidebands move into negative frequency. These can both be undesirable in the resultant signal, and so filters were applied at either end

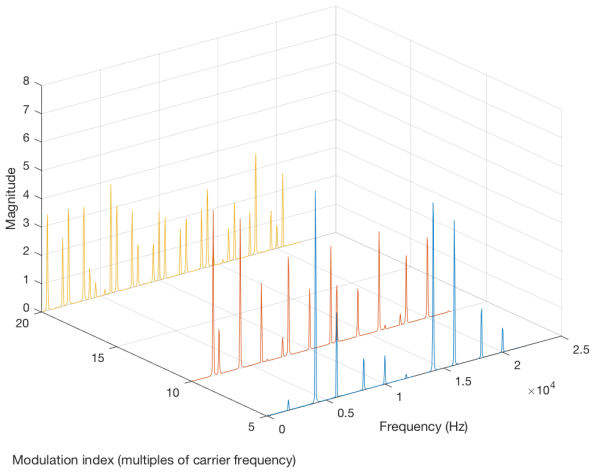

Fig. 11: Spectrum of a 10000Hz sine wave, frequency modulated with different modulation indices.

of the spectrum: lowpass at the hearing threshold of 20 kHz, and a high pass to reduce the sub-audible components below 20Hz.

Delay

Figure 12 shows the impulse response of the ping pong delay. The initial input can be seen at roughly 0.2s in both channels, before alternating between each channel, with delay of 0.5s. The magnitude difference between the first and second echos in the left channel is 0.5, matching that of the feedback factor, as expected.

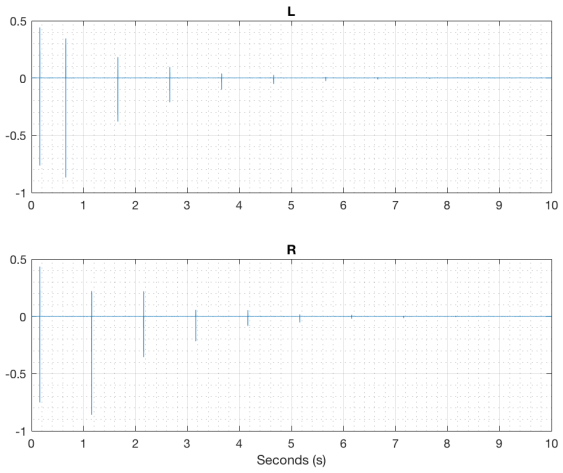

Fig. 12: Two channel impulse response of a ping pong delay, with feedback set to 0.5 and a 0.5s delay time.

Discussion

This report demonstrates the implementation of a single-touch textile FM synthesiser. One matrix is used to control the amplitude, pitch, and amount of delay to add to the signal, whilst the other controls the frequency modulation

relative pitch and amplitude. As such, there remains one unused dimension of the carrier matrix, that could be utilised to further change the timbral character of the carrier wave. The positioning process of the system requires refinement, through either changes to the construction, or to the algorithm. The matrix could be extended to read multiple concurrent touches, providing polyphony in the case of the carrier matrix, though what multi-touch would acheive in the modulator matrix is undecided undecided.

References

[1] Jones, R.E. (2008) *Intimate Control for Physical Modeling Synthesis* University of Victoria

[2] Andrew McPherson, Astrid Bin, Liam Donovan, Christian Heinrichs, Robert Jack, *Bela* https://bela.io/

[3] Andrew McPherson, Astrid Bin, Liam Donovan, Christian Heinrichs, Robert Jack, *Bela* http://docs.bela.io/

[4] Rebecca Stewart (2018) *Real Time DSP*

[5] Donneaud, M., Honnet C., and Strohmeier P. (2018) *Designing a Multi-Touch eTextile for Music Performances* NIME

[6] Smith III, J. O. (2011) *Spectral Audio Signal Processing* W3K Publishing

[7] Reiss, J. and McPherson A (2014) *Audio Effects: Theory, Implementation and Application* CRC Press

[8] Dixon, S. *ECS731: Music Analysis and Synthesis* Queen Mary, University of London

[9] Roads, C. (1995) *The Computer Music Tutorial* MIT Press

[10] Kurt James Werner (2015) *The XQIFFT: Increasing the Accuracy of Quadratic Interpolation of Spectral Peaks via Exponential Magnitude Spectrum Weighting* ICMC

[11] Owens, R. (1997) *Analysis of binary images* University of Edinburgh

[12] Hass, J. (2013) *Introduction to Computer Music: Volume One* Indiana University

Appendix A
Sensor Construction

The sensors were constructed on a 30x30x8cm corkboard base. Conductive fabric was attached to non-elastic, non-conductive material using fabric glue 13, gluing only the non-conductive strips. Strips of 19mm copper tape were cut such that they were sufficiently long enough to cover all 4 conductive rows of the fabric, plus the offset from the edge of the fabric. The copper tape was affixed to paper using the adhesive on the back of the copper tape, and separated by 5mm 14. The piezoresistive foam, 150x150mm was placed on top of the copper tape electrodes, and the fabric pinned on top to hold all layers in place 15. The electrodes were connected to the Bela board using crocodile clips, taped to the board to hold them in place.

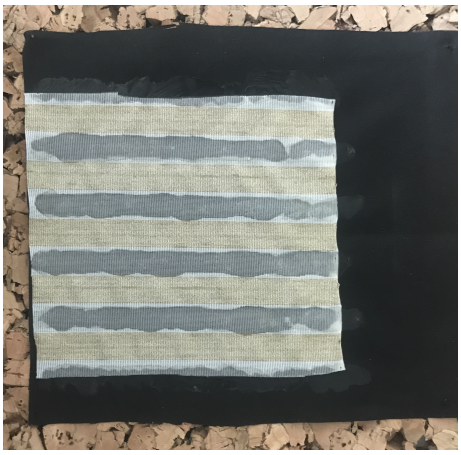

Fig. 13: Fabric with conductive strips (gold), glued to a non-conductive backing. Conductive strips are 10mm wide, 11mm apart.

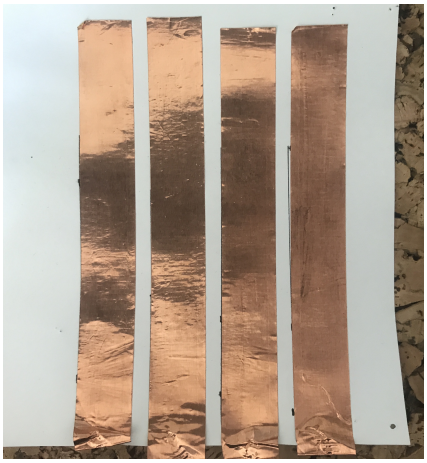

Fig. 14: Copper tape, fixed to paper base. Tape strips is 19mm wide, 150mm long and 5mm apart.

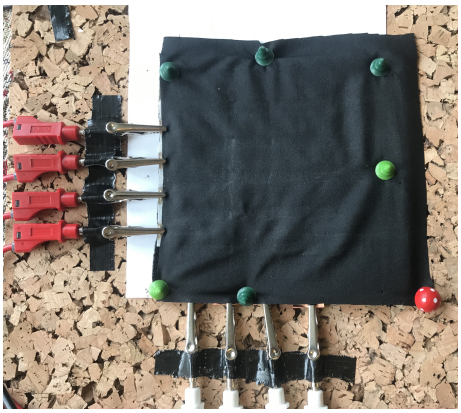

Fig. 15: Complete matrix. The top layer, with the conductive fabric, is pinned in place on top of the resistive foam and copper tape.