Creative Computing II

Christophe Rhodes
c.rhodes@gold.ac.uk

Autumn 2010, Wednesdays:
10:00–12:00: RHB307 & 14:00–16:00: WB316
Winter 2011, TBC

Octave

Introduction

Octave: high-level language for numerical computations:

- part of the GNU project;
- dealing with *signals*:
  - construction;
  - manipulation;
  - visualization.
- interactive interface.

Octave: high-level language for numerical computations:

- part of the GNU project;
- dealing with *signals*:
  - construction;
  - manipulation;
  - visualization.
- interactive interface.

For our purposes:

- visualisation of signals;
- rapid investigation of audio;
- (next term) linear systems and filters.

Octave

Scalars

Two types of scalar:

- numbers:
  - `3`
  - `-6`
  - `3.1416`
- strings:
  - `'a string'`

Octave

Scalar Operations

Both *operators* and *functions* operate on scalars:

- basic mathematical operators: `+`, `-`, `*`, `/`, `^`
- statistical functions: `min()`, `max()`, `mean()`
- trigonometric functions: `sin()`, `cos()`, `tan()`

Octave

Scalar Operations

Both *operators* and *functions* operate on scalars:

- basic mathematical operators: `+`, `-`, `*`, `/`, `^`
- statistical functions: `min()`, `max()`, `mean()`
- trigonometric functions: `sin()`, `cos()`, `tan()`

Examples:

- `4.5 + 9.6 * 2`
- `(4.5 + 9.6) * 2`
- `(4.5+9.6) ^ 2 / 2`

Octave

Variables

Names for values:

- variables are untyped;
- values have type;
- = is assignment.

Octave

Variables

Names for values:

- variables are untyped;
- values have type;
- = is assignment.

Examples:

- `a = -100`
- `aLongVariableName = 10`

Octave

Variables

Names for values:

- variables are untyped;
- values have type;
- = is assignment.

Examples:

- `a = -100`
- `aLongVariableName = 10`

Some predefined constants:

- `pi`
- `e`

Octave

Other programming constructs

Relational operators:

- <, >, <=, >=
- ==, !=

Relational operators:

- `<`, `>`, `<=`, `>=`
- `==`, `!=`

Conditionals:

- `if(...)   ...   endif`
- `switch ...   case {...} ...   otherwise ...   endswitch`

Octave

Other programming constructs

Relational operators:

- `<`, `>`, `<=`, `>=`
- `==`, `!=`

Conditionals:

- `if(...) ... endif`
- `switch ... case {...} ... otherwise ... endswitch`

Loops:

- `for(...) ... endfor`
- `while(...) ... endwhile`

Octave

Other programming constructs

Relational operators:

- `<`, `>`, `<=`, `>=`
- `==`, `!=`

Conditionals:

- `if(...) ... endif`
- `switch ... case {...} ... otherwise ... endswitch`

Loops:

- `for(...) ... endfor`
- `while(...) ... endwhile`

Look, ma, no braces!

We can define *scripts* and *functions* to perform calculations:

- script: no arguments, no results: just side-effects;
- function: multiple arguments, multiple results, side-effects.

Octave

Scripts and Functions

We can define *scripts* and *functions* to perform calculations:

- script: no arguments, no results: just side-effects;
- function: multiple arguments, multiple results, side-effects.

```
function f = add(x, y)
  f = x + y;
endfunction

function [a, b] = fun(c, d)
  a = c + 1;
  b = sin(d);
endfunction
```

Octave

Vectors

Vectors are sequences of scalars:

- `[0 1 2 3 4 5]`
- `[0:5]`
- `[0:1:5]`

Octave

Vectors

Vectors are sequences of scalars:

- `[0 1 2 3 4 5]`
- `[0:5]`
- `[0:1:5]`

Many of the same operations work on vectors as scalars:

- `1 + [0:5]`
- `[0:5] - 2`
- `3 * [0:5]`
- `sin([0:5])`

Vectors are sequences of scalars:

- `[0 1 2 3 4 5]`
- `[0:5]`
- `[0:1:5]`

Many of the same operations work on vectors as scalars:

- `1 + [0:5]`
- `[0:5] - 2`
- `3 * [0:5]`
- `sin([0:5])`

but not division (/), exponentiation(^) or multiplication of vectors by vectors.

Octave

Vector indexing

Elements of a vector can be retrieved by *indexing*:

- `[0:5](3)`
- `sin([0:5])(3)`

Octave

Vector indexing

Elements of a vector can be retrieved by *indexing*:

- `[0:5](3)`
- `sin([0:5])(3)`

Note: although Java and *Processing* arrays are indexed starting from `0`, *Octave* arrays are indexed starting from `1`.

Octave

Vector multiplication

Vectors can be multiplied in two ways:

- **element-by-element**: the two vectors must have the same dimensions:
  - `[0:5] .* [5:-1:0]`
  - error: `[0:5] .* [1:5]`

  (also called the Hadamard product)

Octave

Vector multiplication

Vectors can be multiplied in two ways:

- **element-by-element**: the two vectors must have the same dimensions:
  - `[0:5] .* [5:-1:0]`
  - error: `[0:5] .* [1:5]`

  (also called the Hadamard product)
- **matrix**: the two vectors must have the same length and one must be *transposed*:
  - `[0:5] * [5:-1:0]'`
  - `[0:5]' * [5:-1:0]`
  - error: `[0:5] * [5:-1:0]`

Signals

What is a **signal**?

- a time-varying quantity;

Signals

What is a **signal**?

- a time-varying quantity;
- any quantity varying over space or time.
- (maths: *function*; physics: *field*)

Signals

What is a **signal**?

- a time-varying quantity;
- any quantity varying over space or time.
- (maths: *function*; physics: *field*)

Examples:

- temperature at a location, measured hourly;
- electrical current at a point in a circuit;
- count of students attending weekly lectures;

Signals

What is a **signal**?

- a time-varying quantity;
- any quantity varying over space or time.
- (maths: *function*; physics: *field*)

Examples:

- temperature at a location, measured hourly;
- electrical current at a point in a circuit;
- count of students attending weekly lectures;
- intensity of light at a location on a photosensitive receptor;
- temperature at all places within a room.

Signals

For now, restrict to one-dimensional time signals.

- *continuous-time* signals:
  - current at a point in a circuit;
  - temperature at a location.

Signals

For now, restrict to one-dimensional time signals.

- *continuous-time* signals:
  - current at a point in a circuit;
  - temperature at a location.
- *discrete-time* signals:
  - current at a point in a circuit, measured once per second;
  - temperature at a location, measured hourly;
  - count of students attending weekly lectures.

Signals

Sampling

A continuous-time signal can be converted to a discrete-time one by **sampling**:

- choose an interval in time (the *sampling period*);
- measure the signal at a start time;
- after a sampling period has elapsed, measure again;
- (repeat)

A continuous-time signal can be converted to a discrete-time one by **sampling**:

- choose an interval in time (the *sampling period*);
- measure the signal at a start time;
- after a sampling period has elapsed, measure again;
- (repeat)

The *sampling frequency* or *sampling rate* is the reciprocal of the sampling period:

$$SR = f_s = \frac{1}{\tau}$$

Signals

Vector representation

Octave vectors can be used to represent one-dimensional discrete-time signals.

- each entry in the vector is the corresponding measured value;
- first entry (vector index 1) is measurement at start time;
- successive entries at successive measurement times.

Signals

Vector representation

Octave vectors can be used to represent one-dimensional discrete-time signals.

- each entry in the vector is the corresponding measured value;
- first entry (vector index 1) is measurement at start time;
- successive entries at successive measurement times.

Converting index i to and from time t:

$$i(t) = 1 + \frac{t - t_0}{\tau};$$

$$t(i) = t_0 + \tau(i - 1);$$

Signals

Audio Signals

Measurement of displacement of a microphone membrane:

- pressure difference causes motion;
- electrical signal proportional to displacement;
- sampled at some sampling frequency.

Signals

Audio Signals

Measurement of displacement of a microphone membrane:

- pressure difference causes motion;
- electrical signal proportional to displacement;
- sampled at some sampling frequency.

Common sampling frequencies for audio:

- **44.1kHz**;

Measurement of displacement of a microphone membrane:

- pressure difference causes motion;
- electrical signal proportional to displacement;
- sampled at some sampling frequency.

Common sampling frequencies for audio:

- **44.1kHz**;
- 192kHz, 96kHz, 48kHz;
- 22.05kHz, 11.025kHz;
- 32kHz, 16kHz, 8kHz.

Why does the sampling frequency matter?

- playback: need to know how to convert discrete-time index to real time;
- information: sampling destroys information.

Maximum frequency representable (*Nyquist frequency*) is half the sampling frequency:

$$f_{\text{Nyquist}} = \frac{1}{2} f_s$$

Sinusoid (generic sin and cos)

- using only sinusoids, can build any signal whatsoever
- **Fourier** theory
  - discrete-time: Fourier *series*;
  - continuous-time: Fourier *transforms*.

In general:

$$s_f(t) = \frac{a_0}{2} + \sum_{k=1}^{\infty} \left(a_k \cos(2\pi kft) + b_k \sin(2\pi kft) \right)$$

Example: square wave.

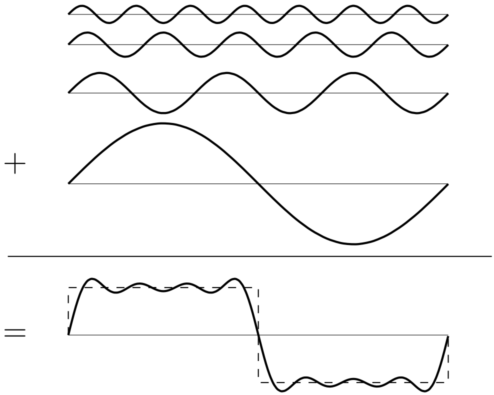
$$\begin{aligned} s_f(t) &= \sin(2\pi ft) + \frac{1}{3}\sin(2\pi 3ft) + \frac{1}{5}\sin(2\pi 5ft) + \ldots \\ &= \sum_{k=1}^{\infty} \frac{1}{2k-1} \sin(2\pi(2k-1)ft) \end{aligned}$$

Signals

Audio signals

Square wave:

$$s_f(t) = \sum_{k=1}^{\infty} \frac{1}{2k-1} \sin(2\pi(2k-1)ft)$$



How to go from a signal to its component sinusoids?

- Fourier analysis

Quick recipe:

- take one cycle (T) of the signal to be decomposed (frequency $\frac{1}{T}$);
- perform **Hadamard** product of signal with $\cos(2\pi\frac{t}{T})$;
- average resulting signal values and multiply by 2, giving a_1;

How to go from a signal to its component sinusoids?

- Fourier analysis

Quick recipe:

- take one cycle (T) of the signal to be decomposed (frequency $\frac{1}{T}$);
- perform **Hadamard** product of signal with $\cos(2\pi\frac{t}{T})$;
- average resulting signal values and multiply by 2, giving a_1;
- repeat above steps with $\cos(2 \times 2\pi\frac{t}{T})$, giving a_2;

How to go from a signal to its component sinusoids?

- Fourier analysis

Quick recipe:

- take one cycle (T) of the signal to be decomposed (frequency $\frac{1}{T}$);
- perform **Hadamard** product of signal with $\cos(2\pi\frac{t}{T})$;
- average resulting signal values and multiply by 2, giving a_1;
- repeat above steps with $\cos(2 \times 2\pi\frac{t}{T})$, giving a_2;
- repeat above steps with $\cos(k \times 2\pi\frac{t}{T})$, giving a_k;

How to go from a signal to its component sinusoids?

- Fourier analysis

Quick recipe:

- take one cycle (T) of the signal to be decomposed (frequency $\frac{1}{T}$);
- perform **Hadamard** product of signal with $\cos(2\pi\frac{t}{T})$;
- average resulting signal values and multiply by 2, giving a_1;
- repeat above steps with $\cos(2 \times 2\pi\frac{t}{T})$, giving a_2;
- repeat above steps with $\cos(k \times 2\pi\frac{t}{T})$, giving a_k;
- repeat above steps with $\sin(k \times 2\pi\frac{t}{T})$, giving b_k.

Amplitude mapping:

- convert measurements at the microphone to number between -1 and 1;
- maximum displacement (in either direction) mapped to ± 1;
- all other displacements mapped linearly.
- (similar to time mapping)

Signals

Audio signals

Amplitude mapping:

- convert measurements at the microphone to number between -1 and 1;
- maximum displacement (in either direction) mapped to ± 1;
- all other displacements mapped linearly.
- (similar to time mapping)

Consequences:

- displacement amplitude directly corresponds to pressure;
- SPL $= 20\times$ log(amplitude) (with 0dB $=$ maximum)
- with both (amplitude and sampling frequency) mappings, can reproduce corresponding analogue signal.

Signals

Audio signals and Octave

Construct a signal:

```
octave> 0.1*sin(2*pi*440*[0:1/8000:1]);
```

- pure sinusoid;
- amplitude one-tenth of the maximum;
- frequency 440 Hz;
- one second long;
- sampling frequency 8kHz.

Play a signal:

```
octave> sound(0.1*sin(2*pi*440*[0:1/8000:1]), 8000)
```

or

```
octave> wavwrite(0.1*sin(2*pi*440*[0:1/8000:1]),8000,'foo.wav')
```

Multiple channels: represented as multiple *columns*:

```
octave> t = [0:1/8000:1];
octave> signal = [sin(2*pi*440*t);sin(2*pi*550*t)]';
octave> sound(0.1*signal,8000)
```

or

```
octave> wavwrite(0.1*signal,8000,'foo.wav')
```

(conventionally, first channel: left; second channel: right)

Additive synthesis:

- making signals by adding together other signals;
- (square wave example);
- perceived pitch is the *fundamental* frequency.

Minim: `AudioSignal` interface:

- additive synthesis;
- implement `generate()` methods:
  - `void generate(float[])`
  - `void generate(float[], float[])`
- live generation context; interface implementor must:
  - keep track of state;
  - be able to work with many different parameter values.

Signals

Driver:

```
import ddf.minim.*; import ddf.minim.signals.*;
Minim minim; AudioOutput out; Sine sine;

void setup () {
  minim = new Minim(this);
  sine = new Sine(440);
  out = minim.getLineOut(Minim.STEREO, 1024);
  out.addSignal(sine);
}
void draw () { }
void stop () {
  out.close();
  minim.stop();
  super.stop();
}
```

Sine class, first attempt:

```
class Sine implements AudioSignal {
  float f;
  Sine(float frequency) { f = frequency; }
  void generate(float[] buf) {
    for(int i = 0; i < buf.length; i++) {
      buf[i] += sin(TWO_PI * f * i / 44100);
    }
  }
  void generate(float[] left, float[] right) {
    generate(left);
    generate(right);
  }
}
```

Sine class, second attempt:

```
class Sine implements AudioSignal {
  float f;
  int t = 0;
  Sine(float frequency) { f = frequency; }
  void generate(float[] buf) {
    for(int i = 0; i < buf.length; i++) {
      buf[i] += sin(TWO_PI * f * t++ / 44100);
    }
  }
  void generate(float[] left, float[] right) {
    generate(left);
    generate(right);
  }
}
```

Sine class, third attempt:

```
class Sine implements AudioSignal {
  float f;
  int t = 0;
  Sine(float frequency) { f = frequency; }
  void generate(float[] buf) {
    for(int i = 0; i < buf.length; i++) {
      buf[i] += sin(TWO_PI * f * t++ / 44100);
    }
  }
  void generate(float[] left, float[] right) {
    int savet = t;
    generate(left);
    t = savet;
    generate(right);
  }
}
```

Audio signals: a summary

- sinusoids are the building blocks of audio signals;
- sampling frequency an important parameter;
- overall amplitude maximum is 1;
- representation as *Octave* vectors or *Processing* classes;
- playback using *Octave* `sound` function or Minim (*Processing*) `AudioSignal`.