Lecture 12: I want to wake up in the city that never sleeps
Lecture 12: The city never sleeps, better slip you an Ambien
Last time

We covered a bit of history that led to the development of S/R -- This history is important in that it helps us see what the originating use cases were (and how they have or have not shifted over the intervening decades)

We had a look at some of the basic data types in R -- From the atomic types like integers and real numbers and characters and booleans, to (slightly) higher level structures like vectors and lists

We also had a brief introduction to object-oriented programming in R -- We saw S3 classes and the simple dispatch mechanism behind their design and use
When reviewing the history of S, we started with an early meeting at Bell Labs that had a group of statisticians asking some pretty basic questions:

“**What do we want?** We want to have easy, flexible, availability of basic or higher level operations, with convenient data manipulation, bookkeeping and IO capacity. We want to be able easily to modify data, output formats, small and large programs and to do this and more with a standard language adapted to statistical usage.”

If we ask this question today, would the answer be different? What do you think?
At each annual Supercomputing Conference a handful of innovations are selected as the year’s “disruptive technologies” that are most likely to revolutionize high-performance computing. These are described as “drastic innovations in current practices...that have the potential to completely transform” the landscape.

At this year’s event in New Orleans, the focus will be on “new computing architectures and interfaces that will significantly impact the high-performance computing field throughout the next five to 15 years,” a focus that is reflected in the list of disruptive exhibitors who were selected by an SC committee.

Another “qualification” of those selected innovations is that they cannot have already emerged into the landscape in any meaningful way—that they sit on the bleeding edge waiting for impetus to burst forth and cause a paradigm shift.

At the edge of this potential sea-change in HPC—and included on that SC10 list of innovations this year is a one-man show run by Karim Chine of his newly-minted company, Cloud Era, Ltd.

Chine’s opportunity to showcase his “Google Docs-like portal for scientific computing in the cloud” could mean that his three-year effort, which he
Software tools for data analysis; The future of R

This meeting will have 2 parts:

In **Part I.** we'd like to develop a framework for talking in subsequent meetings about software tools that people use for data analysis. This part will be a moderated discussion on what are the main tools (such as R, C, Perl, Python, SPSS, Matlab, SAS, SQL, Hadoop to just mention a few), and which ones we'd like to hear more about in subsequent meetings (short 10-15 mins talks). We'll also discuss what information should these short talks contain for each tool, for example usefulness for data analysis, comparison with R, extensions, does the tool interface with R (both directions), what kind of data is this tool used for analyzing (e.g. numeric, categorical, unstructured text), what part of the data analysis process is it best used for (e.g. pre-processing, exploration/visualization, modeling etc.), learning curve of the tool etc. We'll also discuss how to select/blend together all these tools for an optimal data analysis workflow.

Moderators: **Szilard and Jan**

In **Part II.** we'll have some discussions on the future of R. This has been inspired by the recent discussions on the same topic on quite a few blogs. Avram Aelony will introduce the topic and we'll follow with open discussions.

Avram will speak about:
- What R is good/great at
- What R is not good/great at (e.g. Big Data processing, brief mention of Ross
Today

We will go over classes again and exhibit a simple implementation of the Shazam algorithm in R -- Our goal is not to “solve” the problem for you, but to provide you with a lower bound and an existence proof.

You are not expected to make use of this code, but instead we hope that you’ll improve on it! You are also not expected to program in R (we’re just starting after all) but R is good for getting acquainted with the suite of new concepts and tools this project has thrown at you.

Also, this humble piece of code is not a complete working system -- At best it computes some small part of a larger chain of tasks that you all mapped out with your previous assignment.
A tour of R through Shazam

But this is just the briefest of introductions -- For the rest of the lecture, I want to apply this to solve a real problem, well, “real” in the sense that it’s one of your homework assignments.

I promised that we’d make the underlying components of the Shazam algorithm **a bit more comprehensible** -- Delivering on that promise will give us a pretty good introduction to many of the features of R. Rather than keep on with vectors, let’s plunge ahead and look at a (slightly) more complicated “object” in R, a “class” that represents audio or sounds -- We’ll see that it’s been implemented as a kind of extension of vector.

The first task, it would seem would be to bring “audio data” into R -- We are going to start with a very simple type of sound file but will indicate at the end of the lecture how things can be made more complex.
WAV files

The WAV format (or WAVE, which is short for the Waveform Audio File Format) is actually pretty simple -- A audio waveform is represented as **amplitude values recorded at a sequence of times** (there’s some extra detail we could go into about how these amplitudes are quantized, but that’s enough for now)

Matthias Heymann (now at Duke) has developed “sound”, a package that allows you to load a WAV file into R and perform **basic manipulations on the underlying waveform** -- At its core, a sound file is a sequence of amplitudes and the R “object” Matthias developed behaves like a **vector**

So, let’s have a look at CRAN, the Comprehensive R Archive Network...
sos
A Sound Interface for R

sp
classes and methods for spatial data

spBayes
Univariate and Multivariate Spatial Modeling

spaa
SSpecies Association Analysis

space
Sparse PArtial Correlation Estimation

spacetime
classes and methods for spatio-temporal data

spam
SPArse Matrix

sparcl
Perform sparse hierarchical clustering and sparse k-means clustering

sparr
Sparklines and graphical tables for tex and html

sparkTable
The sparr package: SPAtial Relative Risk

spatial
c Sparse Discriminant Analysis

spatialCovariance
Spatial count regression

spatialkernel
Graphs for spatial point patterns

spatialsegregation
Functions for Kriging and Point Pattern Analysis

spatstat
Computation of spatial covariance matrices for data on rectangles

spc
Nonparametric estimation of spatial segregation in a multivariate point process

spcosa
Segregation measures for multitype spatial point patterns

spdep
Spatial Point Pattern analysis, model-fitting, simulation, tests

spe
Statistical Process Control

spectralGP
Stochastic Proximity Embedding

speedglm
Approximate Gaussian processes using the Fourier basis

spf
Fitting Linear and Generalized Linear Models to large data sets

spef
Semiparametric estimating functions
sound: A Sound Interface for R

Basic functions for dealing with wav files and sound samples.

Version: 1.3
Depends: R (≥ 1.4.1)
Published: 2010-09-28
Author: Matthias Heumann
Maintainer: Matthias Heumann <mail at MatthiasHeumann.de>
License: GPL (≥ 2)
URL: http://www.MatthiasHeumann.de
CRAN checks: sound results

Downloads:
Package source: sound_1.3.tar.gz
MacOS X binary: sound_1.3.tgz
Windows binary: sound_1.3.zip
Reference manual: sound.pdf
Old sources: sound archive

Reverse dependencies:
Reverse suggests: seewave
Package ‘sound’

September 28, 2010

Version 1.3
Date 2010-09-27
Title A Sound Interface for R
Author Matthias Heymann <mail@MatthiasHeymann.de>
Maintainer Matthias Heymann <mail@MatthiasHeymann.de>
Depends R (>= 2)
Description Basic functions for dealing with wav files and sound samples.
License GPL (>= 2)
URL http://www.MatthiasHeymann.de
Repository CRAN
Date/Publication 2010-09-28 07:22:36

R topics documented:

appendSample ........................................ 2
bits .................................................. 3
center .............................................. 4
channels ............................................ 5
cutSample .......................................... 6
cutSampleEnds .................................... 7
duration ............................................ 8
fitSampleParameters .............................. 10
left .................................................. 11
loadSample ......................................... 12
mirror .............................................. 13
normalize ......................................... 14
noSilence ......................................... 15
nullSample ........................................ 16
Ops.Sample ......................................... 17
Loading packages

One of the great success stories of R (as opposed to S) is it’s **package management system** -- It is very easy to share code with others and to make use of other people’s code.

In the lab, the sound package should already be installed and it is also available on homework.stat202a.org -- If you would like to follow along with this lesson on your own machine, however, you will first have to install the package...

```r
> install.packages("sound")  # not necessary in the lab or homework
> library(help="sound")      # what's this package about?
> library("sound")          # what's this package about?
```
Information on package 'sound'

Description:

Package: sound
Version: 1.3
Date: 2010-09-27
Title: A Sound Interface for R
Author: Matthias Heymann <mail@MatthiasHeymann.de>
Maintainer: Matthias Heymann <mail@MatthiasHeymann.de>
Depends: R (>= 1.4.1)
Description: Basic functions for dealing with wav files and sound samples.
License: GPL (>= 2)
URL: http://www.MatthiasHeymann.de
Packaged: 2010-09-27 18:40:30 UTC; Matthias
Repository: CRAN
Date/Publication: 2010-09-28 07:22:36
Built: R 2.11.1; ; 2010-09-29 06:59:30 UTC; unix

Index:

Ops.Sample        Basic Operations for Sample Objects
Sample            Sample Objects
Sine               Create Sample Objects for the Basic waveforms
WavPlayer          Set or Get the System Command for Playing WAV Files
appendSample      Append Sample Objects
bits               Bits per Sample
center            Center a Sample Object.
channels          Number of Channels of a Sample Object
cutSample         Cut Sample Objects
cutSampleEnds     Prepare Sample Object for appendSample
Now, let’s get busy!

The sound package was built around a use case in which audio files are stored on your local computer, that is, we refer to WAV files with a file name -- One small improvement to this package would be to allow for general “connections” so that one could pick up data from the web

For now, if you are working on a lab machine (or your own laptop), you can just download files from our course web site using the following R command

```r
> download.file(
+   "http://www.stat.ucla.edu/~cocteau/stat202a/jay-z.wav",
+   "jay-z.wav")
```

trying URL 'http://www.stat.ucla.edu/~cocteau/stat202a/jay-z.wav'
Content type 'audio/x-wav' length 4430002 bytes (4.2 Mb)
opened URL
===============================================
downloaded 4.2 Mb

(You can also use files directly from /data/music on homework)
# read "empire state" by jay-z into r using a function from the sound package

```r
> jz <- loadSample("jay-z.wav")
> class(jz)
[1] "Sample"

> jz

  type    : mono
  rate    : 8000 samples / second
  quality : 16 bits / sample
  length  : 2214979 samples
  R memory: 8859916 bytes
  HD memory: 4430002 bytes
  duration: 276.872 seconds

# a Sample object consists of the sound (audio waveform) as a matrix (in case there
# was a left and right channel) and constants representing the sampling rate and the
# number of bits used per sample

> names(jz)
[1] "sound" "rate"  "bits"

> class(jz$sound)
[1] "matrix"

> dim(jz$sound)
[1] 1 2214979

> jz$rate
[1] 8000
> jz$bits
[1] 16
```
# in terms of methods, we can subset a Sample object, focusing on just the first
# 4 seconds, say (remember the WAV file had a sample rate of 8000 samples/second)...

> jz[1:32000]

- type : mono
- rate : 8000 samples / second
- quality : 16 bits / sample
- length : 32000 samples
- R memory : 128000 bytes
- HD memory : 64044 bytes
- duration : 4 seconds

# ... or we can make a “custom” plot, specially designed for this class of object
# about 55 seconds into the song, alicia keys starts to sing (8000 samples/second)

> plot(jz[(53*8000+1):(57*8000)])
R and objects

And with this, we see our first hint at proper **objects in R** -- In this case, we say that `Sample` is an example of an S3 class (although we won’t go into a lot of detail now except to say that the S refers to the S language and 3 refers to its version; and there are also S4 classes associates with S version 4)

To apply the term “object” to `jz`, we expect to find descriptions of **both state and behavior**, data and methods -- While objects as containers for data are not philosophically dissimilar from those we saw in Python, the way we invoke methods in R is considerably different
R and objects: Data-directed programming

We have already noted that, as with Python, everything in R is an object -- The designers of S/R believe strongly that through objects our programs become easier to design and, ultimately, more trustworthy

According to Robert Gentleman “… it is often easier to design, write and maintain software when there is some clear separation of the data representation from the operations that are to be performed on it.”

Or, as John Chambers puts it “As you become more involved with software for data analysis, however, creating related classes and methods... makes the analysis more natural and convenient... and the results more trustworthy”
Data-directed v. object-oriented programming

Many object-oriented languages like Python and Java are (as Gentleman describes them) "class-centric", meaning they classes define objects and are "repositories for the methods that act on" them.

R, on the other hand, separates the class information from the creation of so-called generic functions and (again, quoting Gentleman) can be thought of as a "function-centric" system.
Generic functions

The S3 class and method system is designed around the concept of a generic function; a generic function has different behaviors depending on the class of one or more of its arguments (this is known as polymorphism).

Generics perform a kind of method “dispatch” that, in turn, selects the appropriate method to be called; we’ve seen a number of S3 generics before...
# spotting a generic function...

> **print**
function (x, ...)
UseMethod("print")
<environment: namespace:base>

> **summary**
function (object, ...)
UseMethod("summary")
<environment: namespace:base>

> **residuals**
function (object, ...)
UseMethod("residuals")
<environment: namespace:stats>

> **fitted**
function (object, ...)
UseMethod("fitted")
<environment: namespace:stats>
S3 classes and methods

You can access the class of an S3 object with the function `class()` -- This can be used to both determine as well as set the class of an object* and `is.object()` tests to see if an R object has a class attribute

The function `UseMethod` **dispatches on the class of the object** returned by `class()`; methods are simply ordinary functions that are identified by a **special naming convention**

Specifically, methods are given names that are **concatenations of the name of the generic method and the name of the class** that they are intended to apply to, separated by a “.”

You can list the methods associated with a particular generic with a call to the function `methods()`

* Except for special cases involving implicit classes, this is the same as creating an attribute called `class` with value the string with the class name
> summary
function (object, ...)
UseMethod("summary")
<environment: namespace:base>

# find all the methods for this function that R knows about...

> methods("summary")
[1] summary.aov            summary.aovlist         summary.aspell*
[4] summary.connection     summary.data.frame    summary.Date
[7] summary.default        summary.ecdf*           summary.factor
[10] summary.glm            summary.infl           summary.lm
[13] summary.loess*         summary.loglm*          summary.manova
[16] summary.matrix         summary.mlm             summary.negbin*
[19] summary.nls*           summary.packageStatus* summary.polr*
[22] summary.POSIXct         summary.POSIXlt         summary.ppr*
[25] summary.prcomp*        summary.princomp*       summary.rlm*
[28] summary.srcfile        summary.srcref          summary.stepfun
[31] summary.stl*           summary.table           summary.tukeysmooth*

Non-visible functions are asterisked
> methods("summary")

[1] summary.aov summary.aovlist summary.connection
[4] summary.data.frame summary.Date summary.default
[7] summary.ecdf* summary.factor summary.glm
[10] summary.infl summary.lm summary.loess*
[16] summary.nls* summary.packageStatus* summary.POSIXct
[19] summary.POSIXlt summary.ppr* summary.prcomp*
[22] summary.princomp* summary.stepfun summary.stl*
[25] summary.table summary.tukeysmooth*

Non-visible functions are asterisked

> methods("residuals")

[1] residuals.default residuals.glm residuals.HoltWinters*
[4] residuals.isoreg* residuals.lm residuals.nls*
[7] residuals.smooth.spline* residuals.tukeyline*

Non-visible functions are asterisked

> methods("AIC")

[1] AIC.default* AIC.logLik*

Non-visible functions are asterisked
> methods("print")

Non-visible functions are asterisked
> methods("[")
[1] [.acf* [.AsIs [.data.frame
[4] [.Date [.difftime [.factor
[7] [.formula* [.getAnywhere* [.hexmode
[10] [.listof [.noquote [.numeric_version
[13] [.octmode [.POSIXct [.POSIXlt
[16] [.roman* [.simple.list [.terms*
[19] [.ts* [.tskernel* [.XMLInternalDocument*
[22] [.XMLInternalNode* [.XMLNode*

Non-visible functions are asterisked

> methods("["]
[1] [[.data.frame [[.Date [[.dendrogram*
[4] [[.factor [[.numeric_version [[.POSIXct
[7] [[.tclArray* [[.XMLDocumentContent* [[.XMLHashTreeNode*
[10] [[.XMLInternalDocument* [[.XMLInternalNode* [[.XMLNode*

Non-visible functions are asterisked

> methods("<-")
[1] <-.CURLOptions* <-.data.frame <-.Date <-.factor
[5] <-.POSIXct <-.POSIXlt <-.ts* <-.XMLNode*

Non-visible functions are asterisked
Aside: Some history

The earliest versions of the S language were developed prior to widespread adoption of object-oriented programming principles; as a result, some of the basic classes in R do not define themselves via the class attribute and are referred to as implicit classes.

For example, functions are implicitly of class function, while matrices and arrays are implicitly of classes matrix and array, respectively -- As a result, is.object() will return FALSE when applied to objects having an implicit class; UseMethod dispatches, however, on even implicit classes (depending only on the result of a call to class()).

These constructions were relatively late additions to the language and hence the two versions; S3 evolved out of a significant effort to introduce modeling functions (lm, glm, gam, loess) into the language in the early 1990s, while S4 was, well, a more reliable second attempt in the late 1990s.

S3 lacks formal specification of classes, and is really about function dispatch, about generic functions and polymorphism (we will call this data-directed programming); S4 introduces formal class definitions and a complete system for inheritance.
Returning to our example

In the next few slides, we'll make use of two classes of objects (**one for the audio signal**, `Sample`, and **one for the spectrogram**, `specgram`) -- We'll make use of commands like `help()`, `class()`, `names()`, `typeof()` and `methods()` to do a little exploration of what each object is all about...
Returning to our example

We have already seen how to read a WAV file into R and have found that the resulting object is of class `Sample` -- We can have a look at the various methods Matthias provided us to work with this class

```r
> jz <- loadSample("jay-z.wav")
> class(jz)
[1] "Sample"

> jz

  type      : mono
  rate      : 8000 samples / second
  quality   : 16 bits / sample
  length    : 2214979 samples
  R memory  : 8859916 bytes
  HD memory : 4430002 bytes
  duration  : 276.872 seconds

> methods(class="Sample")
[1] [.Sample Ops.Sample play.Sample
[7] sampleLength<-.Sample sum.Sample
A second sound file

We’ll use as a comparison Sinatra’s “New York, New York” (why not?) --
Again, you can either download the file or use it directly from /data/music on homework

```r
> download.file(
+   "http://www.stat.ucla.edu/~cocteau/stat202a/sinatra.wav",
+   "sinatra2.wav")

trying URL 'http://www.stat.ucla.edu/~cocteau/stat202a/sinatra.wav'
Content type 'audio/x-wav' length 3299244 bytes (3.1 Mb)
opened URL
==================================================
downloaded 3.1 Mb

> sinatra <- loadSample("sinatra.wav")
> class(sinatra)
[1] "Sample"

> sinatra

  type    : mono
  rate    : 8000 samples / second
  quality : 16 bits / sample
  length  : 1649600 samples
  R memory: 6598400 bytes
  HD memory: 3299244 bytes
  duration: 206.2 seconds
```
For homework you outlined the basic steps of the Shazam algorithm -- After we've acquired some data, the first step involves computing a time-frequency representation of an audio signal known as a spectrogram.

For this, we appeal to a second package, \texttt{signal}, which is basically a port to \texttt{R} of a number of signal processing functions from Matlab.

\begin{verbatim}
> install.packages("signal")  # not necessary in the lab/homework
> library(help="signal")      # what's this package about?
> library("signal")
\end{verbatim}
# compute the spectrogram for 15 seconds of each of jay-z and sinatra -- the
# exact choice was made so that something changed in the song during the interval

# the constants define the sampling rate for the signal and the size of the
# window we want to use to compute the fourier transform (here, 64ms)

> jzspec <- specgram(jz$sound[(50*8000+1):(65*8000)],64*8000/1000,8000,64*8000/1000)

> class(jzspec)
[1] "specgram"

# a specgram object has data representing the “matrix” of intensities, and vectors
# of the times and frequencies at which it was computed -- the matrix is made up of
# complex numbers

> names(jzspec)
[1] "S" "f" "t"

> class(jzspec$S)
[1] "matrix"

> dim(jzspec$S)
[1] 256 467

> typeof(jzspec$S)
[1] "complex"
Spectogram plot

Description:

Generate a spectrogram for the signal. This chops the signal into overlapping slices, windows each slice and applies a Fourier transform to determine the frequency components at that slice.

Usage:

\[
\text{specgram}(x, \ n = \min(256, \ \text{length}(x)), \ \text{Fs} = 2, \ \text{window} = \text{hanning}(n), \ \text{overlap} = \text{length} (\text{window})/2)
\]

## S3 method for class 'specgram':
plot(x, \ ...)

## S3 method for class 'specgram':
print(x, \ ...)

Arguments:

- \text{x}: the vector of samples.
- \text{n}: the size of the Fourier transform window.
- \text{Fs}: the sample rate, Hz.
- \text{window}: shape of the fourier transform window, defaults to 'hanning(n)'. The window length for a hanning window can be specified instead.
- \text{overlap}: overlap with previous window, defaults to half the window length.
Then there is the choice of colormap. A brightness varying colormap such as copper or bone gives good shape to the ridges and valleys. A hue varying colormap such as jet or hsv gives an indication of the steepness of the slopes. The final spectrogram is displayed in log energy scale and by convention has low frequencies on the bottom of the image.

Value:

For `specgram` list of class `specgram` with items:

- `S` : complex output of the FFT, one row per slice.
- `f` : the frequency indices corresponding to the rows of `S`.
- `t` : the time indices corresponding to the columns of `S`.

Author(s):

Original Octave version by Paul Kienzle <email:pkienzle@users.sf.net>. Conversion to R by Tom Short.

References:

Octave Forge <URL:http://octave.sf.net>

See Also:

`fft`, `image`

Examples:

```r
specgram(chirp(seq(-2, 15, by = 0.001), 400, 10, 100, 'quadratic'))
specgram(chirp(seq(0, 5, by = 1/8000), 200, 2, 500, "logarithmic"), Fs = 8000)
```
# and now examine the methods associated with specgram objects

> class(jzspec)
[1] "specgram"

> methods(class ="specgram")
[1] plot.specgram  print.specgram

# have a look at the plot function, for example -- it’s just a call to image()
# note that it maps black to low intensity and white to high...

> plot.specgram
function (x, ...)
{
  image(20 * log10(t(abs(x$S))), col = gray(0:512/512), axes = FALSE, ...)
}

> plot(jzspec)

# finally, compute the spectrogram for francis albert...

> sinspec <- specgram(sinatra$sound[(10*8000+1):(25*8000)],64*8000/1000,8000,64*8000/1000)
default specgram plot: 15 seconds of jay-z.wav
Before we move on...

We can take a moment with the spectrogram and try to see if we understand what it’s doing as a display tool -- For this we can use the tools that both the sound and signal packages provide

First, we can generate simple audio signals and check our intuition about this transform -- We can generate pure sounds (a sine wave) or simple composite sounds (two sine waves) or add noise to a simple signal...
# pure sine wave
> sig <- Sine(500,3,rate=8000)
> sp <- specgram(sig$sound,64*8000/1000,8000,64*8000/1000)
>
# black is high intensity, white is low
> plot(sp)
> plot(sp$f,abs(sp$S)[,5],type="l",xlab= "frequency", ylab= "intensity")

# mixture of two sine waves
> sig <- 0.75*Sine(500,3,rate=8000) + 0.25 *Sine(1000,3,rate=8000)
> sp <- specgram(sig$sound,64*8000/1000,8000,64*8000/1000)
>
> plot(sp)
> plot(sp$f,abs(sp$S)[,5],type="l",xlab= "frequency", ylab= "intensity")

# ... adding noise
> sig <- Sine(500,3,rate=8000) + 0.25 *Noise(3,rate=8000)
> sp <- specgram(sig$sound,64*8000/1000,8000,64*8000/1000)
>
> plot(sp)
> plot(sp$f,abs(sp$S)[,5],type="l",xlab= "frequency", ylab= "intensity")
single sine wave
two sine waves
two sine waves
noisy sine wave
noisy sine wave
A local transform

For these simple signals, we could compute a **single Fourier transform** (well, the Fast Fourier Transform or FFT) **for the whole (discrete) signal**, and you are no doubt familiar with this kind of spectral breakdown -- For music, however, **the frequency profile is constantly changing**, necessitating a transformation that is more “localized” in time

To achieve this, the spectrogram is computing a **number of FFTs in short intervals of time** -- In the examples we have chosen intervals of 64 ms which means $64 \times \frac{8000}{1000} = 512$ samples

Dividing the signal in this way, however, introduces artifacts into the computed spectrum (FFT) that were not present in the original signal (at the ends of an interval the signal’s amplitude drops abruptly) -- To avoid this we **often apply a “window” to the signal** to smooth things out a bit
# exhibiting the hanning window...

```r
> find("hanning")
[1] "package:signal"

> hanning
function (n)
{
    if (!(length(n) == 1 & (n == round(n)) & (n > 0)))
        stop("hanning: n has to be an integer > 0")
    if (n == 1)
        c = 1
    else {
        n = n - 1
        c = 0.5 - 0.5 * cos(2 * pi * (0:n)/n)
    }
    c
}

> plot(hanning(512),type="l",xlab="sample number",ylab= "window value")
```
A local transform

Finally, the window function we’ve introduced, while it reduces artifacts at the edges of each interval, results in a kind of periodic damping -- To prevent this, we often allow the intervals to overlap

By default, the function specgram staggers intervals so that a new one starts halfway through an existing one -- In our case, with intervals consisting of 512 samples, a new interval would begin every 256 samples
A local transform

For more information on the spectrogram, a readable introduction can be found at

http://cnx.org/content/m0505/latest/

We can also do a little work using special parts of our song signals -- Take for example Alicia Keys’ contribution to Empire State...
Jay-Z

Here we examining part of the song (just under a second) where Alicia Keys is singing “Yo-o-o-rk” and ending with Jay-Z saying “Hey” (hmm) -- The original signal is plotted and a spectrogram of just this section of the song is computed

Make sure this matches your intuition!

```r
# what kind of object is this?
> yooork <- jz[(55.75*8000+1):(56.65*8000)]
> sp <- specgram(yooork$sound,64*8000/1000,8000,64*8000/1000)
> plot(sp)
```
Jay-Z

Here we see the higher-frequency components that track her singing of “Yo-o-ork”, we see the fuzzy line in the middle that corresponds to a drum (thud) that hits while she’s singing, and we see Jay-Z’s “Hey” at the end of the track.

Keep in mind that the default plot for a specgram object has higher intensities colored white and weaker intensities colored black -- The flip of what we saw in the Shazam paper

Let’s see if we can change that...
we can make a plot that’s closer to a figure in the shazam paper by letting white
denote low intensity and black be high intensity -- we also show the values un-logged
and add axis labels

> grays <- gray((100:0)/100)

> image(jzspec$t,jzspec$f,t(abs(jzspec$S)),col=grays,xlab="time",ylab="frequency")
> title("jay-z, empire state of mind")

... and now 15 seconds of sinatra -- it’s band until 6 seconds in, then he
"starts spreading the news..."

> image(sinspec$t,sinspec$f,t(abs(sinspec$S)),col=grays,xlab="time",ylab="frequency")
> title("sinatra, (new york)^2")
sinatra, (new york)^2
THE FREQUENCIES OF MUSIC
(Ranges of the fundamental frequencies of instruments and voices)

The harmonic frequencies generated by instruments and voices extend off the right side of the chart, though at volume levels far below those of the fundamental frequencies shown. The A above middle C is usually set at the standard tuning pitch of 440 Hz.

http://www.psbspeakers.com/audio-topics/The-Frequencies-of-Music
Shazam

Since the spectrogram is really a matrix, we can look at slices at fixed times (columns) to compare the peaks in intensity -- We subset a matrix with a pair of indices, the first representing rows, the second columns

```r
# start by pulling out some of the components of the specgram object --
# technically it’s better to leave them with the object but for
# readability of the next few slides, i’ll pull them out

> freqs <- jzspec$f
> times <- jzspec$t

> jzmat <- abs(jzspec$S)
> sinmat <- abs(sinspec$S)

> plot(freqs,jzmat[,400],type="l",xlab="frequency",ylab="intensity")
> lines(freqs,jzmat[,100],col=5)
> title("slices before and after alicia keys (cyan=before, black=after)")
```
slices before and after alicia keys (cyan = before, black = after)
a slice from the full spectrogram for jay–z

# the code, for completeness -- how many times does alicia keys sing?  
# (hint: rhymes with door)

> bigspec <- specgram(jz$sound, 64*8000/1000, 8000, 64*8000/1000)  
> plot(spec$t, abs(spec$S[177,]), type="l", xlab="times", ylab="intensity at 2750 Hz")  
> title("a slice from the full spectrogram for jay-z")
Identifying constellations

The paper is somewhat vague about the way in which high points are identified -- We’ll take a relatively simple approach for this first introduction and leave it to you to enhance it.

We’ll divide the 256 frequencies (rows in the specgram object) into 16 bins and create strips that cut across time -- We then compute the maximum in each column in the strip and tag as constellation points the top 5% of these maxima.

This simple approach is particularly well-suited to fast computations in R -- It will give us a chance to see vector and matrix operations...
# let’s consider a chunk of rows that go from 209 to 224 (or 3250 Hz to 3484.375 Hz)

> i <- 14
> ((i-1)*16+1):(i*16)

> strip <- jzmat[((i-1)*16+1):(i*16),]

# now find the max values in each column -- that is, for each column in the strip
# look for the maximum of the 16 values (the second line creates a matrix where
# each column is assigned its max value

> maxs <- apply(strip,2,max)
> mmaxs <- matrix(maxs,nrow=16,ncol=ncol(strip),byrow=T)

# finally, take the top 5% of the max values...

> cutoff <- quantile(maxs,p=0.95)

# ... and create a binary matrix

> const <- (strip==mmaxs) & (mmaxs>cutoff)

# and let’s have a look...

> image(times, freqs[((i-1)*16+1):(i*16)],t(strip),col=gray((100:0)/100))
> image(times, freqs[((i-1)*16+1):(i*16)],t(const),col=gray(c(1,0)))
# collecting these lines into a function that will repeat the operation for
# all 16 strips, pasting together the result into a matrix that’s the same
# dimension as the original spectrogram

find_constellation <- function(mat){

  const <- matrix(TRUE,nrow=nrow(mat),ncol=ncol(mat))

  for(i in 1:16){
    strip <- mat[((i-1)*16+1):(i*16),]
    maxs <- apply(strip,2,max)
    mmaxs <- matrix(maxs,nrow=16,ncol=ncol(strip),byrow=T)
    cutoff <- quantile(maxs,p=0.95)
    const[((i-1)*16+1):(i*16),] <- (strip==mmaxs)&(mmaxs>cutoff)
  }

  return(const)
}

# and applying it to jz and sinatra sounds

> jzmap <- find_constellation(jzmat)
> sinmap <- find_constellation(sinmat)

> image(times,freqs,t(jzmap),col=gray(c(1,0)))
> image(times,freqs,t(sinmap),col=gray(c(1,0)))
constellation for jay–z
constellation for sinatra
It’s likely that one of the next components of your system involves identifying a set of targets associated with each point in the constellation -- Because time and frequency have such different scales, we opted to represent each point by its row and column indices in the spectrogram.

Then, given a point in the constellation, we identify its targets as the 10 nearest points that are ahead of it in time -- Nearest is judged using Euclidean distance with the row and column indices of candidates.

A better approach would be to use a weighted Euclidean distance that rescales each coordinate (time and frequency) so that one can more precisely tailor the extent to which targets should be in front of versus above and below a point in the constellation.
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>T</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\((1,6)\)
\((2,2)\)  \((2,4)\)  \((2,9)\)
\((3,7)\)

\(d^2 = 5\)
\(d^2 = 25\)
\(d^2 = 10\)
# start by changing from a matrix with TRUE/FALSE to a collection
# of pairs, each representing the row and column index of the point

> mfreqs = matrix(1:length(freqs),ncol=length(times),nrow=length(freqs))
> mtimes = matrix(1:length(times),ncol=length(times),nrow=length(freqs),byrow=T)

> jzconst <- cbind(mtimes[jzmap],mfreqs[jzmap])

# and have a look...

> plot(jzconst,col=5,pch=19,xlab="columns",ylab="rows")

# pick a point in the constellation (here we just take the 186th, somewhat
# randomly) and identify 10 targets ahead of it in time

> i <- 186

> candidates <- jzconst[jzconst[,1]>jzconst[i,1],]
> candidates <- candidates[order(idist(jzconst[i,],candidates)),]

> for(j in 1:10) lines(rbind(jzconst[i,],candidates[j,]))

# doing this a few times creates the plot on the next page... the order() function
# returns the set of indices required to rearrange a vector into ascending order --
# here we provide it another variable to sort on, the distance between the
# 186th constellation point and all the candidate points (we’ll talk about its
# format later)

> idist <- function(pt,candidates){
    apply(candidates,1,function(x,y){ sqrt(sum((y-x)^2)) },y=pt)
}
sample of targets for sinatra
Hashing

Presumably a later step involves **computing a hash of each of the triples** representing a segment joining a constellation point and a target (the first and second frequencies and the time difference between them)

The purpose of the hash function is to provide us with **a mapping of these triples of information into a single number** so that we can search a database more quickly -- Having to hunt along for matching end points could take a long time if you have to, in effect, run a program for each match (even a fraction of a second per song could add up to hours if your collection of music is large enough)

Databases can be highly optimized to conduct searches on single columns of numbers (among other things, of course)
Suppose for the moment that we need to simply encode the row numbers associated with the frequencies in the spectrogram matrix -- Each of these are integers that run from 1 to 256

Recalling that 256 = 2^8 we can use the following simple mapping to go from the row number for frequency 1, r1, and that for frequency 2, r2:

\[
> r1 \leftarrow 129 \\
> r2 \leftarrow 205 \\
> h \leftarrow (r2-1) \times 2^8 + (r1-1) \\
> h \\
\text{[1]} \quad 52352
\]
Hashing

We can then “decode” the hash using the following mod operators -- First we look at the remainder after division by $2^8$ and then rescale the rest...

```r
> h
[1] 52352

> r1d <- (h %% 2^8) + 1
> r2d <- (h-r1d+1)/2^8

> r1d
[1] 129

> r2d
[1] 204
```
Hashing

To encode the time difference, we’ll use another 8 bits -- Each **unit difference between the columns of the spectrogram represents 32 ms** by design so that a difference of 255 means $0.032 \times 255 = 8.16$ seconds

That’s plenty of time as we’re assuming we will have between 3 and 30 constellation points per second -- To encode this value along with the others is really no more than another shift and rescaling

See? Hashing is not so mysterious -- And now, with this function in hand, we can create our “database” of constellations...
# here are two usable hash functions -- we should be careful about time differences
# larger that 256 (we’ll talk about how to protect ourselves from this kind of
# thing later)

> ihash <- function(r1,r2,dtr){
    dtr*2^16 + (r2-1)*2^8 + (r1-1)
}

> dihash <- function(h){
    r1 <- (h %% 2^8) + 1
    h <- (h-r1+1)/2^8
    r2 <- (h %% 2^8) + 1
    dtr <- (h-r2+1)/2^8
    return(c(r1,r2,dtr))
}

> ihash(15,224,131)
[1] 8642318

> dihash(ihash(15,224,131))
[1]  15 224 131
Pulling it all together

We now create a single function that will go from a Sample object through to a collection of hashes -- We are assuming we’ll process each song in overlapping 15 second segments, building hashes for all the constellation points that occur before the 10 second mark.

The following function also assumes a mono sound file with a sample rate of 8000 samples per second and 16bits per sample -- It’s possible to reformat just about any sound file to these specifications (or you can make this all more robust!)
shazam <- function(snd){

  spec <- specgram(snd$sound, 64*8000/1000, 8000, 64*8000/1000)
  map <- find_constellation(abs(spec$S))

  mfreqs = matrix(1:length(spec$f), ncol=length(spec$t), nrow=length(spec$f))
  mtimes = matrix(1:length(spec$t), ncol=length(spec$t), nrow=length(spec$f), byrow=T)

  const <- cbind(mtimes[map], mfreqs[map])

  hashes <- NULL

  for(i in 1:nrow(const)){
    # we are going to process each song in overlapping 15 second chunks and
    # want to stop building constellation/target segments at 10 seconds

    if(const[i,1] < 314){
      candidates <- const[const[,1]>const[i,1],]
      candidates <- candidates[order(idist(const[i,],candidates)),]

      for(j in 1:min(c(10,nrow(candidates)))){
        hashes = rbind(hashes, c(const[i,],
                                  ihash(const[i,2], candidates[j,2], candidates[j,1]-const[i,1])))
      }
    }

  }

  return(hashes)
}
> sinshaz <- shazam(sinatra[(10*8000+1):(25*8000)])
> jzshaz <- shazam(jz[(50*8000+1):(65*8000)])

# test to see if we have any hashes that match between jay-z and sinatra...

> sum(jzshaz[,3] %in% sinshaz[,3])
[1] 4

# now try an unknown sample recorded by my laptop

> unknown <- loadSample("longsample.wav")
> sunknown <- unknown[(30*8000+1):(45*8000)]
> ushaz <- shazam(sunknown)

> sum(ushaz[,3] %in% sinshaz[,3])
[1] 10
> dim(ushaz)
[1] 1890  3

> sum(ushaz[,3] %in% jzshaz[,3])
[1] 86

> plot(ushaz[,1:2],pch=19,col=5,cex=0.5,xlab="time",ylab="frequency")
> points(ushaz[,1:2][ushaz[,3] %in% jzshaz[,3],])

> sunknown <- unknown[(35*8000+1):(50*8000)]
> ushaz <- shazam(sunknown)
> sum(ushaz[,3] %in% jzshaz[,3])
[1] 483

> plot(ushaz[,1:2],pch=19,col=5,cex=0.5,xlab="time",ylab="frequency")
> points(ushaz[,1:2][ushaz[,3] %in% jzshaz[,3],])
What a ride!

Ok, so this lecture was packed -- Some basic components in the language and a brief introduction to at least one of the object models in R

It’s a lot to digest and over the next few lectures we’ll pull things together properly -- For the moment, I just wanted to give you a sense of how we can use R to help us refine our Shazam algorithm

As an aside, we can use the shazam function we created to piece together constellations for the entire song -- If we then run pieces of our unknown sample through it, we see big matches at sensible places...
seconds 25–40 of unknown
seconds 5–20 of unknown
seconds 35–50 of unknown

Frequency

0 100 200 300 400 500

0 100 200 300 400 500 600 700 800

time differences