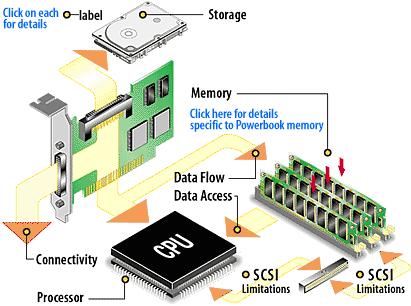
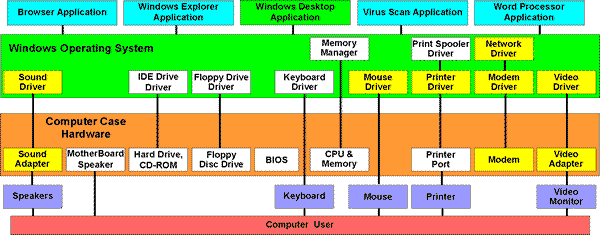
# programming and python introduction

## Computer Basics



## Computer and Software Conceptual Layers



## Example Software / Program

function [] = Mortgage()

function [newvalues] = getnewvalues(oldvalues)

prompt = {'Value', 'Down payment', 'Annual Interest', ‘’’’’’’’’’’’’’’’’’'Term (Years)'};

dlg\_title = 'Mortgage Parameters';

num\_lines = 1;

def = {num2str(oldvalues(1)), num2str(oldvalues(2)), ‘’’’’’’’’’’’’’’num2str(oldvalues(3)), num2str(oldvalues(4))};

valuesstr = inputdlg(prompt, dlg\_title, num\_lines, def);

newvalues = str2double(valuesstr(:));

end

% Initialize parms before entering the

% loop.

Finished=false;

parms=[110000,5000,6,20];

while ~Finished

Change = input('Change loan parameters?', 's');

if isempty(Change);

Change = '0';

end

if Change == 'N' || Change == 'n';

Finished = true;

elseif Change == 'Y' || Change == 'y';

Finished = false;

parms=getnewvalues(parms);

loan\_amount = parms(1) - parms(2);

display(loan\_amount);

else disp('Enter y or n');

Finished = false;

end

end

% disp('Great!')

end

## Variables

A variable is a named reference to a storage location.

* The name is used to reference the location
* In a more abstract sense, the name is used to reference an object
* The value of the variable can change during program execution
* Generally the referenced object has an associated type like integer, real number, character, string, function, etc. In statically typed language like Java, the type is associated with the variable. However, with dynamically typed languages like Python, the type is associated with the value stored in the variable.

### Variable Scope

A variable’s scope is the portion of the program code for which the variable’s name is “visible” meaning that the variable can be referenced to access its contents. See 1.6.5 Functions and Scope below.

## Logic Statements

Logic statements allow the programmer to alter the flow of a program by testing conditions.

### IF conditional

y = 1

if y == 1:

print "y still equals 1, I was just checking"

if y < 1:

print “What now?”

if y <= 1:

print “Did this work?”

### IF THEN ELSE conditional

a = 1

if a > 5:

print "This shouldn't happen."

else:

print "This should happen."

And now add an else-if

if a > 5:

print "Big number!"

elif a % 2 != 0:

print "This is an odd number"

print "It isn't greater than five, either"

else:

print "this number isn't greater than 5"

print "nor is it odd"

print "feeling special?"

What is the result if a=10, a=3, a=4?

### SELECT (CASE) conditional

Assume n is entered by a user

switch(n) {

case 0:

printf("You typed zero.\n");

break;

case 1:

case 9:

printf("n is a perfect square\n");

break;

case 2:

printf("n is an even number\n");

case 3:

case 5:

case 7:

printf("n is a prime number\n");

break;

case 4:

printf("n is a perfect square\n");

case 6:

case 8:

printf("n is an even number\n");

break;

default:

printf("Only single-digit numbers are allowed\n");

break;

}

## Looping Constructs

### FOR Loops

list = [2, 4, 6, 8]

sum = 0

for num in list:

sum = sum + num

print("The sum is:", sum)

### WHILE Loops

a = 0

while a < 10:

a = a + 1

print a

x = 10

while x != 0:

print x

x = x - 1

print "wow, we've counted x down, and now it equals", x

print "And now the loop has ended."

### DO UNTIL Loops

Pseudo-code (NOT Python or R)

X = 1 Y=1

Do while Y <= 10 do

X = X + 1 Y = Y + 1

Until X > 10 End do

What are X and Y when the loop exits?

X = 11 Y=11

Do while Y <= 10 do

X = X + 1 Y = Y + 1

Until X > 10 End do

What are X and Y when the loop exits?

### Implicit Loops

b1 <- c(3, 4, 8, 23, 25, 7, 1, 23, 1, 9, 14, 22, 1, 0, 32)

first1a <- function(x) return (which (x == 1)[1])

first1a(b1)

### Functions and Scope

function [] = Mortgage()

function [newvalues] = getnewvalues(oldvalues)

prompt = {'Value', 'Down payment', 'Annual Interest', ‘’’’’’’’’’’’’’’’’’'Term (Years)'};

dlg\_title = 'Mortgage Parameters';

num\_lines = 1;

def = {num2str(oldvalues(1)), num2str(oldvalues(2)), ‘’’’’’’’’’’’’’’num2str(oldvalues(3)), num2str(oldvalues(4))};

valuesstr = inputdlg(prompt, dlg\_title, num\_lines, def);

newvalues = str2double(valuesstr(:));

end

% Initialize parms before entering the

% loop.

Finished=false;

parms=[110000,5000,6,20];

while ~Finished

Change = input('Change loan parameters?', 's');

if isempty(Change);

Change = '0';

end

if Change == 'N' || Change == 'n';

Finished = true;

elseif Change == 'Y' || Change == 'y';

Finished = false;

parms=getnewvalues(parms);

loan\_amount = parms(1) - parms(2);

display(loan\_amount);

else disp('Enter y or n');

Finished = false;

end

end

% disp('Great!')

end

## Python

About the origin of Python, Guido Van Rossum wrote in 1996:

Over six years ago, in December 1989, I was looking for a "hobby" programming project that would keep me occupied during the week around Christmas. My office ... would be closed, but I had a home computer, and not much else on my hands. I decided to write an interpreter for the new scripting language I had been thinking about lately: a descendant of ABC that would appeal to Unix/C hackers. I chose Python as a working title for the project, being in a slightly irreverent mood (and a big fan of Monty Python's Flying Circus).[13]

An object oriented programming language that helps you create software applications.

* Some people call it a scripting language
* Can create scripts/programs or use interactively
* Compiles to Byte Code (under the hood) and creates a .pyc file
* Byte Code is portable across most platforms
* Extensions exist to compile Byte Code to binary machine code
* Cpython – the python standard
* Jpython – specific extensions for Java
* IronPython – specific extensions for .Net
* Stackless – Support concurrency
* PyPy – Specific implementation for processing speed

Conceptual Hierarchy

1. Programs are comprised of modules
2. Modules contain statements
3. Statements contain expressions
4. Expressions create and process objects

## Python Interactive Prompt

* Python commands only
* Print statements required in files (and maybe more…)
* Do not indent at the interactive prompt unless you are entering a compound statement
* Watch out for prompt changes (indicates you are entering a compound statement)
* Terminate compound statements in interactive mode with a blank line
* Interactive statement executes one statement at a time (you cannot cut and paste multiple lines of code into the interactive session unless the code contains a blanks line after every compound statement)

## Starting Python:

At the system command prompt type python (you may have to specify the full path) or double click the python executable.

pythonw is used if you are calling python from another program or using python to execute a script and want to suppress the terminal window.

python -3.4 myscript.py #the -3.1 tells python which version to run, runs latest version if omitted

## Python Modules

A module is a collection of variable names, known as a name space.

The names within a module are called attributes. An attribute is a variable name attached to a specific object.

## Loading Modules

import sys #Load the sys library

dir(sys)

import os #Load the os library

dir(os)

import re #Load regular expression pattern matching library

dir(re)

**Importing a user Module**

title = 'Something Else'

print(title)

#uses Class\_1\_Module name space

import Class\_1\_Module #has to be in current working directory

dir(Class\_1\_Module)

Class\_1\_Module.title

Class\_1\_Module.MyAge

print(title)

#copies into local name space, overwrites same named variable

#without warning

from Class\_1\_Module import title

print(title)

## Python Functions

def fib(n):

a, b = 0, 1

while a < n:

print(a, end=' ')

a, b = b, a+b

print(a)

fib(1000)

## Python – Reading and Writing Data

Useful programming languages provide methods to read and write files to a directory.

**Reading**

iris = open('Iris\_Data.txt', 'r')

iris.read()

**Writing**

myfile = open('myfile.txt', 'w')

myfile.write('Write something\n')

myfile.write('to the file\n')

myfile.close()

myfile = open('myfile.txt', 'r')

text = myfile.read()

print(text)

## Debugging

>>> mylen(blah)  
Traceback (most recent call last):  
  File "<pyshell#26>", line 1, in <module>  
    mylen(blah)  
NameError: name 'blah' is not defined

def Overlapping(list1, list2):

for i in range(len(list1)):

for j in range(len(list2)):

if list1[i] == list2[j]:

return True

else:

return False

What type is i and j?

This is a list of numbers: [0, 1, 2, 3, 4]

This is a string of numerical characters and commas '1,2,3,4,5,6'

This is a string of only numerical characters '01234'

>>> def filter\_long\_words(list,n):  
    for words in list:  
        if len(words)>n:  
            return words

What’s wrong with this function?

List\_of\_long\_words = []

for words in list:  
        if len(words)>n:  
 List\_of\_long\_words = List\_of\_long\_words + words

return List\_of\_long\_words

def pangram(str):  
     alphabet = [‘q’, ‘w’, ‘e’, ‘r’, ’t’, ‘y’, ‘u’, ‘i’, ‘o’, ‘p’, ‘a’, ’s’, ‘d’, ‘f’, ‘g’, ‘h’, ‘j’, ‘k’, ‘l’, ‘z’, ‘x’, ‘c’, ‘v’, ‘b’, ’n’, ‘m’]  
     for char in str:  
          if char == alphabet:  
              return True  
          else:  
              return False

What’s wrong with this function?

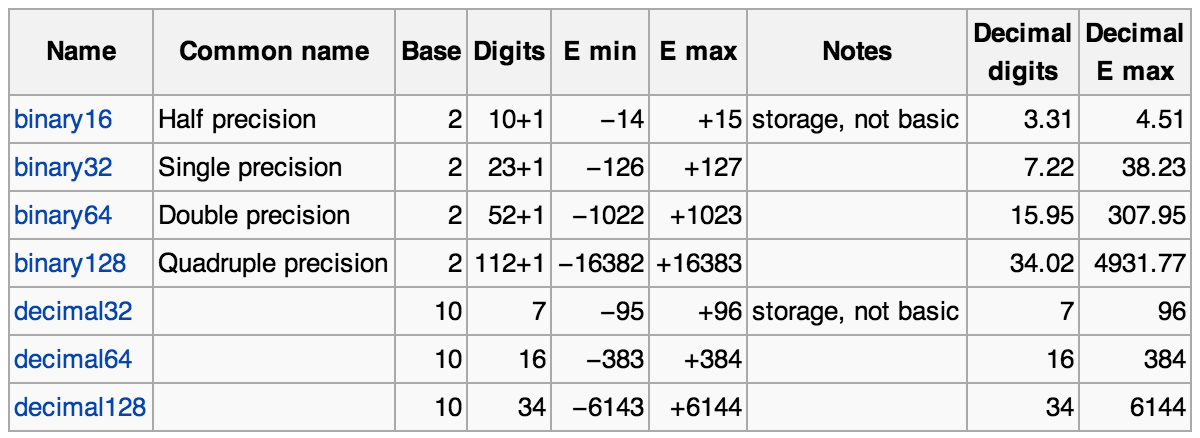
# Real Numbers

Why does it matter?

Representing the infinite number of Real numbers

* Computers are mostly binary
* Need a way to represent real numbers in a finite number of bits
* IEEE 754-2008 is the most common floating point standard (International standard ISO/IEC/IEEE 60559:2011)
* Base and Precision

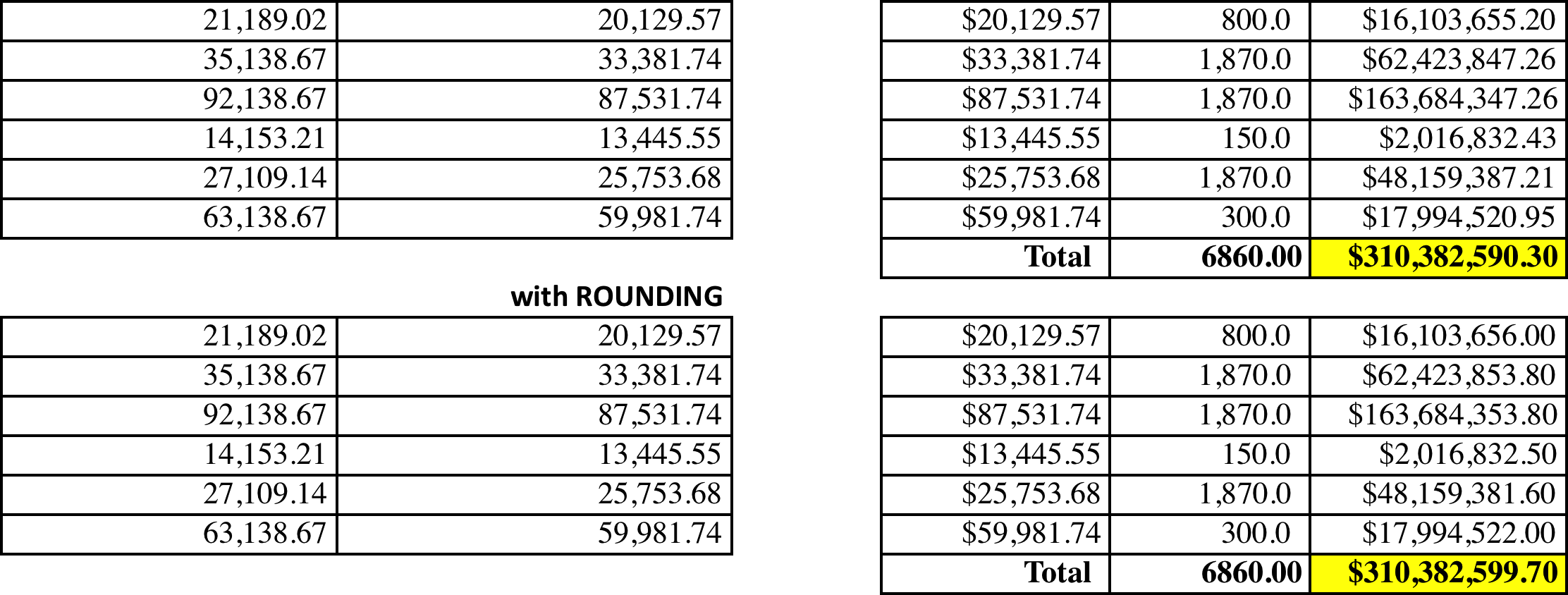
IEEE 754-2008



**Simple Example representing 0.1**



**How does this affect me? … Excel example**



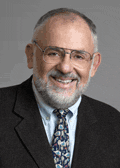
# MATLAB

## History (written by Professor Shaw)

**MATLAB** is a numerical computing environment and programming language. Created by The MathWorks, MATLAB allows easy matrix manipulation, plotting of functions and data, implementation of algorithms, creation of user interfaces, and interfacing with programs in other languages. Although it is numeric only, an optional toolbox interfaces with the Maple symbolic engine, allowing access to computer algebra capabilities.

Short for "matrix laboratory", MATLAB was invented in the late 1970s by Cleve Moler, then chairman of the computer science department at the University of New Mexico. He designed it to give his students access to LINPACK and EISPACK without having to learn Fortran. It soon spread to other universities and found a strong audience within the applied mathematics community. Jack Little, an engineer, was exposed to it during a visit Moler made to Stanford University in 1983. Recognizing its commercial potential, he joined with Moler and Steve Bangert. They rewrote MATLAB in C and founded The MathWorks in 1984 to continue its development. These rewritten libraries were known as JACKPAC.

MATLAB was first adopted by control design engineers, Little's specialty, but quickly spread to many other domains. It is now also used in education, in particular the teaching of linear algebra and numerical analysis, and is popular amongst scientists involved with image processing.

Cleve Barry Moler is a mathematician and computer programmer specializing in numerical analysis. In the mid to late 1970s, he was one of the authors of LINPACK and EISPACK, Fortran libraries for numerical computing. He invented MATLAB, a numerical computing package, to give his students at the University of New Mexico easy access to these libraries without writing Fortran. In 1984, he co-founded The MathWorks with Jack Little to commercialize this program.

He received his bachelor's degree from Caltech in 1961, and a Ph.D. from Stanford University.

He was a professor of math and computer science for almost 20 years at the University of Michigan, Stanford University, and the University of New Mexico. Before joining The MathWorks full time in 1989, he also worked for Intel Hypercube and Ardent Computer Corporation. He is also co-author of four textbooks on numerical methods and is a member of the Association for Computing Machinery. He was vice-president of the Society for Industrial and Applied Mathematics and currently sits on its Board of Trustees. He will serve as president of SIAM for two years starting in January 2007.

He was elected to the National Academy of Engineering on February 14, 1997. He received an honorary degree from Linköping University, Sweden. He received an honorary degree of Doctor of Mathematics from the University of Waterloo on June 16, 2001. On April 30, 2004, he was appointed Honorary Doctor (doctor technices, honoris causa) at the Technical University of Denmark.

Professor G’s Summary:

MATrix LABoratory

Invented in the late 1970s

Cleve Moler, Chair of CSC dept, Univ of New Mexico

Designed it to give his students access to LINPACK and EISPACK (without having to learn Fortran).

Jack Little recognized commercial potential (1983)

Joined with Moler and Steve Bangert.

The three founded The MathWorks in 1984

Rewrote MATLAB in C. Rewritten libraries were known as JACKPAC.

## Matlab Intro

1. Moving around the MATLAB interface and getting help

See the [Matlab Primer](http://www.mathworks.com/help/pdf_doc/matlab/getstart.pdf) (also on Blackboard)

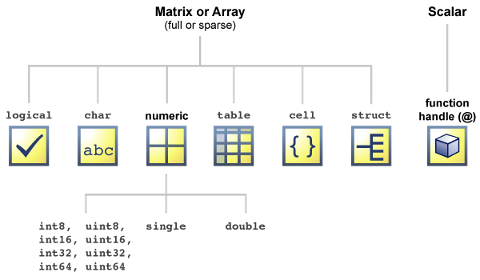
* + Command Window
  + Current Folder
  + Details
  + Workspace
  + Command History
  + Getting Help

1. Solve a simple set of equations – Ancient grain pricing (from 200 BC).

Three sheafs of a good crop, two sheafs of a mediocre crop, and one sheaf of a bad crop are sold for 39 dou. Two sheafs of good, three mediocre, and one bad are sold for 34 dou; and one good, two mediocre, and three bad are sold for 26 dou. What is the price received for each sheaf of a good crop, each sheaf of a mediocre crop, and each sheaf of a bad crop?

## Matlab Classes (Types)

All of the fundamental MATLAB classes are shown in the diagram below:



There are 16 fundamental classes in MATLAB. Each of these classes is in the form of a matrix or array. With the exception of function handles, this matrix or array is a minimum of 0-by-0 in size and can grow to an n-dimensional array of any size. A function handle is always scalar (1-by-1).

Numeric classes in the MATLAB software include signed and unsigned integers, and single- and double-precision floating-point numbers. By default, MATLAB stores all numeric values as double-precision floating point. (You cannot change the default type and precision.) You can choose to store any number, or array of numbers, as integers or as single-precision. Integer and single-precision arrays offer more memory-efficient storage than double-precision.

All numeric types support basic array operations, such as subscripting, reshaping, and mathematical operations.

You can create two-dimensional double and logical matrices using one of two storage formats: [full](http://www.mathworks.com/help/matlab/ref/full.html) or [sparse](http://www.mathworks.com/help/matlab/ref/sparse.html). For matrices with mostly zero-valued elements, a sparse matrix requires a fraction of the storage space required for an equivalent full matrix. Sparse matrices invoke methods especially tailored to solve sparse problems

These classes require different amounts of storage, the smallest being a logical value or 8-bit integer which requires only 1 byte. It is important to keep this minimum size in mind if you work on data in files that were written using a precision smaller than 8 bits.

The following table describes the fundamental classes in more detail.

| **Class Name** | **Documentation** | **Intended Use** |
| --- | --- | --- |
| [double](http://www.mathworks.com/help/matlab/ref/double.html), [single](http://www.mathworks.com/help/matlab/ref/single.html) | [Floating-Point Numbers](http://www.mathworks.com/help/matlab/matlab_prog/floating-point-numbers.html) | * Required for fractional numeric data. * [Double](http://www.mathworks.com/help/matlab/matlab_prog/floating-point-numbers.html#f2-101310) and [Single](http://www.mathworks.com/help/matlab/matlab_prog/floating-point-numbers.html#f2-108458) precision. * Use [realmin](http://www.mathworks.com/help/matlab/ref/realmin.html) and [realmax](http://www.mathworks.com/help/matlab/ref/realmax.html) to show [range of values](http://www.mathworks.com/help/matlab/matlab_prog/floating-point-numbers.html#f2-98685). * Two-dimensional arrays can be [sparse](http://www.mathworks.com/help/matlab/math/full-and-sparse-matrices.html). * Default numeric type in MATLAB. |
| [int8](http://www.mathworks.com/help/matlab/ref/int8.html), uint8,  int16, uint16, int32, uint32, int64, uint64 | [Integers](http://www.mathworks.com/help/matlab/matlab_prog/integers.html) | * Use for signed and unsigned whole numbers. * More [efficient use of memory.](http://www.mathworks.com/help/matlab/matlab_prog/strategies-for-efficient-use-of-memory.html#brh72ex-37) * Use [intmin](http://www.mathworks.com/help/matlab/ref/intmin.html) and [intmax](http://www.mathworks.com/help/matlab/ref/intmax.html) to show [range of values](http://www.mathworks.com/help/matlab/matlab_prog/integers.html#f2-96232). * Choose from 4 sizes (8, 16, 32, and 64 bits). |
| [char](http://www.mathworks.com/help/matlab/ref/char.html) | [Characters and Strings](http://www.mathworks.com/help/matlab/characters-and-strings.html) | * Data type for text. * Native or Unicode®. * Converts to/from numeric. * Use with [regular expressions](http://www.mathworks.com/help/matlab/matlab_prog/regular-expressions.html). * For multiple strings, use cell arrays. |
| [logical](http://www.mathworks.com/help/matlab/ref/logical.html) | [Logical Operations](http://www.mathworks.com/help/matlab/logical-operations.html) | * Use in relational conditions or to test state. * Can have one of two values: [true](http://www.mathworks.com/help/matlab/ref/true.html) or [false](http://www.mathworks.com/help/matlab/ref/false.html). * Also useful in array indexing. * Two-dimensional arrays can be sparse. |
| [function\_handle](http://www.mathworks.com/help/matlab/ref/function_handle.html) | [Function Handles](http://www.mathworks.com/help/matlab/function-handles.html) | * Pointer to a function. * Enables passing a function to another function * Can also call functions outside usual scope. * Useful in Handle Graphics callbacks. * Save to MAT-file and restore later. |
| [table](http://www.mathworks.com/help/matlab/ref/table.html) | [Tables](http://www.mathworks.com/help/matlab/tables.html) | * Rectangular container for mixed-type, column-oriented data. * Row and variable names identify contents. * Use [Table Properties](http://www.mathworks.com/help/matlab/ref/tableproperties.html) to store metadata such as variable units. * Manipulation of elements similar to numeric or logical arrays. * Access data by numeric or named index. * Can select a subset of data and preserve the table container or can extract the data from a table. |
| [struct](http://www.mathworks.com/help/matlab/ref/struct.html) | [Structures](http://www.mathworks.com/help/matlab/structures.html) | * Fields store arrays of varying classes and sizes. * Access one or all fields/indices in single operation. * Field names identify contents. * Method of passing function arguments. * Use in [comma-separated lists](http://www.mathworks.com/help/matlab/matlab_prog/comma-separated-lists.html). * More memory required for overhead |
| [cell](http://www.mathworks.com/help/matlab/ref/cell.html) | [Cell Arrays](http://www.mathworks.com/help/matlab/cell-arrays.html) | * Cells store arrays of varying classes and sizes. * Allows freedom to package data as you want. * Manipulation of elements is similar to numeric or logical arrays. * Method of passing function arguments. * Use in comma-separated lists. * More memory required for overhead |

### Valid Combinations of Unlike Classes

Matrices and arrays can be composed of elements of most any MATLAB® data type as long as all elements in the matrix are of the same type. If you do include elements of unlike classes when constructing a matrix, MATLAB converts some elements so that all elements of the resulting matrix are of the same type.

Data type conversion is done with respect to a preset precedence of classes. The following table shows the five classes you can concatenate with an unlike type without generating an error (that is, with the exception of character and logical).

| **TYPE** | **character** | **integer** | **single** | **double** | **logical** |
| --- | --- | --- | --- | --- | --- |
| **character** | character | character | character | character | invalid |
| **integer** | character | integer | integer | integer | integer |
| **single** | character | integer | single | single | single |
| **double** | character | integer | single | double | double |
| **logical** | invalid | integer | single | double | logical |

For example, concatenating a double and single matrix always yields a matrix of type single. MATLAB converts the double element to single to accomplish this.

## Matlab Basic Analysis (This section is taken directly from the matlab primer for R2014A)[[1]](#footnote-1)

Data Analysis Introduction

Every data analysis has some standard components:

**Preprocessing — Consider outliers and missing values, and smooth data to identify possible models.**

**Summarizing — Compute basic statistics to describe the overall location, scale, and shape of the data.**

**Visualizing — Plot data to identify patterns and trends.**

**Modeling — Give data trends fuller descriptions, suitable for predicting**  **new values. Data analysis moves among these components with two basic goals in mind:**

1  **Describe the patterns in the data with simple models that lead to accurate predictions.**

2  **Understand the relationships among variables that lead to the model.**

**This section explains how to carry out a basic data analysis in the MATLAB**  **environment.**  Preprocessing Data  **This example shows how to preprocess data for analysis.**  Overview  **Begin a data analysis by loading data into suitable MATLAB® container variables and sorting out the "good" data from the "bad." This is a preliminary step that assures meaningful conclusions in subsequent parts of the analysis.**  Loading the Data  **Begin by loading the data in count.dat: load count.dat**

The 24-by-3 array count contains hourly traffic counts (the rows) at three intersections (the columns) for a single day.

Missing Data

The MATLAB NaN (Not a Number) value is normally used to represent missing data. NaN values allow variables with missing data to maintain their structure - in this case, 24-by-1 vectors with consistent indexing across all three intersections.

Check the data at the third intersection for NaN values using the isnan function:

c3 = count(:,3); % Data at intersection 3

c3NaNCount = sum(isnan(c3))

c3NaNCount =

0

isnan returns a logical vector the same size as c3, with entries indicating the presence (1) or absence (0) of NaN values for each of the 24 elements in the data. In this case, the logical values sum to 0, so there are no NaN values in the data.

NaN values are introduced into the data in the section on Outliers. Outliers

Outliers are data values that are dramatically different from patterns in the rest of the data. They might be due to measurement error, or they might represent significant features in the data. Identifying outliers, and deciding what to do with them, depends on an understanding of the data and its source.

One common method for identifying outliers is to look for values more than a certain number of standard deviations from the mean . The following code plots a histogram of the data at the third intersection together with lines at and + ,for =1,2:

bin\_counts = hist(c3); % Histogram bin counts

N = max(bin\_counts); % Maximum bin

count mu3 = mean(c3); % Data mean

sigma3 = std(c3); % Data standard deviation

hist(c3) % Plot histogram

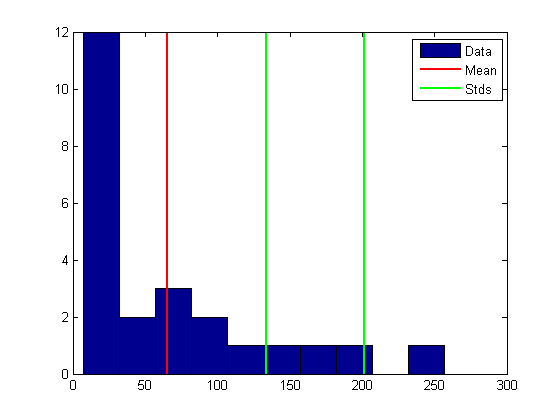
hold on

plot([mu3 mu3],[0 N],'r','LineWidth',2) %

Mean X = repmat(mu3+(1:2)\*sigma3,2,1); Y = repmat([0;N],1,2); plot(X,Y,'g','LineWidth',2) % Standard deviations legend('Data','Mean','Stds')

hold off



The plot shows that some of the data are more than two standard deviations above the mean. If you identify these data as errors (not features), replace them with NaN values as follows:

outliers = (c3 - mu3) > 2\*sigma3; c3m = c3; % Copy c3 to c3m

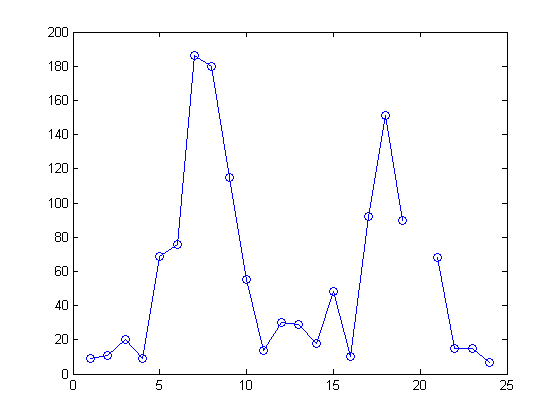
c3m(outliers) = NaN; % Add NaN values

Smoothing and Filtering

A time-series plot of the data at the third intersection (with the outlier removed in Outliers) results in the following plot:

plot(c3m,'o-')

hold on



The NaN value at hour 20 appears as a gap in the plot. This handling of NaN values is typical of MATLAB plotting functions.

Noisy data shows random variations about expected values. You might want to smooth the data to reveal its main features before building a model. Two basic assumptions underlie smoothing:

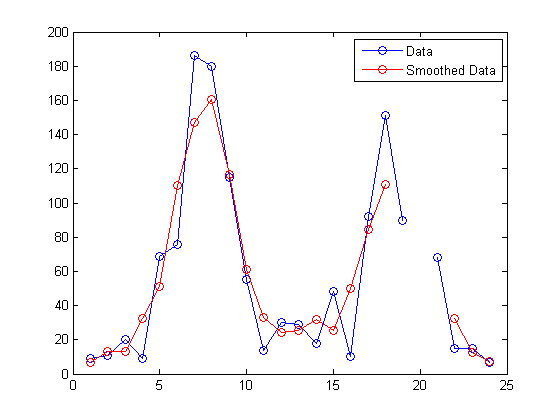
- The relationship between the predictor (time) and the response (traffic volume) is smooth.

- The smoothing algorithm results in values that are better estimates of expected values because the noise has been reduced.

Apply a simple moving average smoother to the data using the MATLAB convn function:

span = 3; % Size of the averaging window window = ones(span,1)/span; smoothed\_c3m = convn(c3m,window,'same');

h = plot(smoothed\_c3m,'ro-'); legend('Data','Smoothed Data')

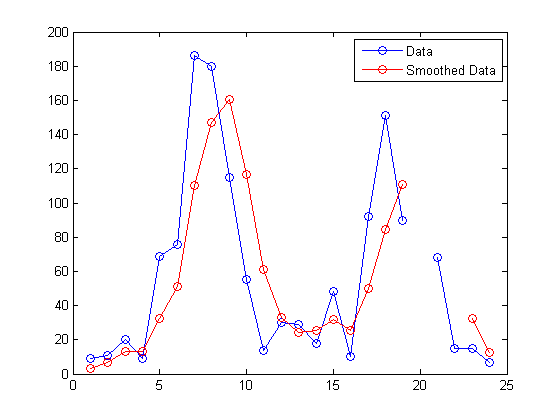


The extent of the smoothing is controlled with the variable span. The averaging calculation returns NaN values whenever the smoothing window includes the NaN value in the data, thus increasing the size of the gap in the smoothed data.

The filter function is also used for smoothing data: smoothed2\_c3m = filter(window,1,c3m);

delete(h)

plot(smoothed2\_c3m,'ro-');



The smoothed data are shifted from the previous plot. convn with the 'same' parameter returns the central part of the convolution, the same length as the data. filter returns the initial part of the convolution, the same length as the data. Otherwise, the algorithms are identical.

Smoothing estimates the center of the distribution of response values at each value of the predictor. It invalidates a basic assumption of many fitting algorithms, namely, that the errors at each value of the predictor are independent. Accordingly, you can use smoothed data to identify a model, but avoid using smoothed data to fit a model.

Summarizing Data

This example shows how to summarize data.

Overview

Many MATLAB® functions enable you to summarize the overall location, scale, and shape of a data sample.

One of the advantages of working in MATLAB® is that functions operate on entire arrays of data, not just on single scalar values. The functions are said to be vectorized. Vectorization allows for both efficient problem formulation, using array-based data, and efficient computation, using vectorized statistical functions.

Measures of Location

Summarize the location of a data sample by finding a "typical" value. Common measures of location or "central tendency" are computed by the functions mean, median, and mode:

load count.dat

x1 = mean(count)

x2 = median(count)

x3 = mode(count)

x1 =

32.0000 46.5417

x2 =

23.5000 36.0000

65.5833

39.0000

x3 = 11 9 9

Like all of its statistical functions, the MATLAB® functions above summarize data across observations (rows) while preserving variables (columns). The functions compute the location of the data at each of the three intersections in a single call.

Measures of Scale

There are many ways to measure the scale or "dispersion" of a data sample. The MATLAB® functions max, min, std, and var compute some common measures:

dx1 = max(count)-min(count) dx2 = std(count) dx3 = var(count)

dx1 = 107 136

dx2 = 25.3703

dx3 =

1.0e+03 \*

0.6437

250

41.4057

1.7144

68.0281

4.6278

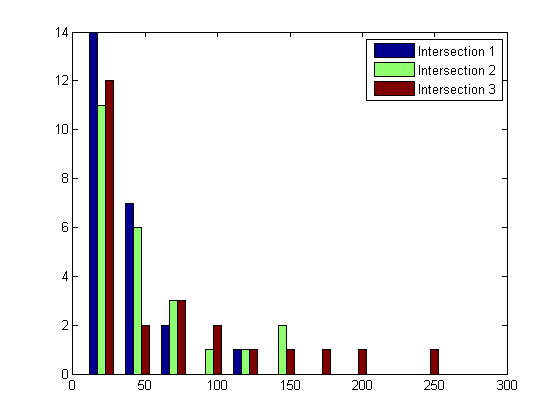
Like all of its statistical functions, the MATLAB® functions above summarize data across observations (rows) while preserving variables (columns). The functions compute the scale of the data at each of the three intersections in a single call.

Shape of a Distribution

The shape of a distribution is harder to summarize than its location or scale. The MATLAB® hist function plots a histogram that provides a visual summary:

figure hist(count) legend('Intersection 1',...

'Intersection 2',... 'Intersection 3')



Parametric models give analytic summaries of distribution shapes. Exponential distributions, with parameter mu given by the data mean, are a good choice for the traffic data:

c1 = count(:,1); % Data at intersection 1 [bin\_counts,bin\_locations] = hist(c1); bin\_width = bin\_locations(2) - bin\_locations(1); hist\_area = (bin\_width)\*(sum(bin\_counts));

figure

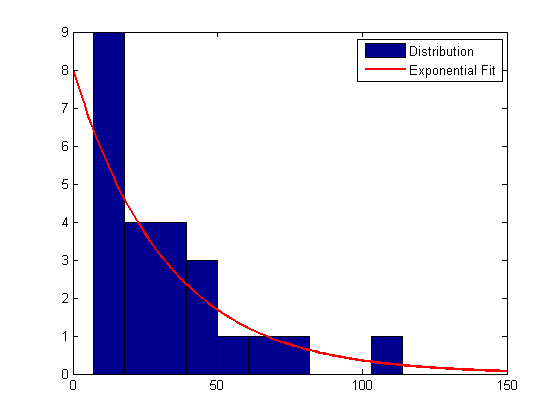
hist(c1)

hold on

mu1 = mean(c1); exp\_pdf = @(t)(1/mu1)\*exp(-t/mu1); % Integrates

% to 1

t = 0:150; y = exp\_pdf(t); plot(t,(hist\_area)\*y,'r','LineWidth',2) legend('Distribution','Exponential Fit')



3-62

Methods for fitting general parametric models to data distributions are beyond the scope of this section. Statistics ToolboxTM software provides functions for computing maximum likelihood estimates of distribution parameters.

Visualizing Data

• “Overview” on page 3-63 • “2-D Scatter Plots” on page 3-63 • “3-D Scatter Plots” on page 3-66 • “Scatter Plot Arrays” on page 3-68 • “Exploring Data in Graphs” on page 3-69

Overview

You can use many MATLAB graph types for visualizing data patterns and trends. Scatter plots, described in this section, help to visualize relationships among the traffic data at different intersections. Data exploration tools let you query and interact with individual data points on graphs.

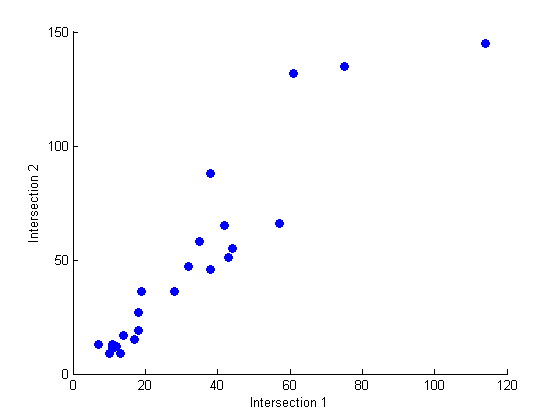
Note This section continues the data analysis from “Summarizing Data” on page 3-58.

2-D Scatter Plots

A two-dimensional scatter plot, created with the scatter function, shows the relationship between the traffic volume at the first two intersections:

load count.dat c1 = count(:,1); % Data at intersection 1 c2 = count(:,2); % Data at intersection 2

figure scatter(c1,c2,'filled') xlabel('Intersection 1') ylabel('Intersection 2')



3-64

The covariance, computed by the cov function measures the strength of the linear relationship between the two variables (how tightly the data lies along a least-squares line through the scatter):

C12 = cov([c1 c2])

C12 =

1.0e+03 \*

0.6437 0.9802

0.9802 1.7144

The results are displayed in a symmetric square matrix, with the covariance of the ith and jth variables in the (i, j)th position. The ith diagonal element is the variance of the ith variable.

Covariances have the disadvantage of depending on the units used to measure the individual variables. You can divide a covariance by the standard deviations of the variables to normalize values between +1 and –1. The corrcoef function computes correlation coefficients:

R12 = corrcoef([c1 c2])

R12 =

1.0000 0.9331

0.9331 1.0000

r12 = R12(1,2) % Correlation coefficient

r12 = 0.9331

r12sq = r12^2 % Coefficient of determination

r12sq =

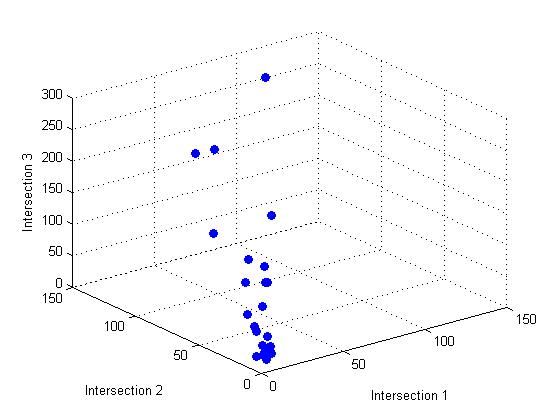
0.8707

Because it is normalized, the value of the correlation coefficient is readily comparable to values for other pairs of intersections. Its square, the coefficient of determination, is the variance about the least-squares line divided by the variance about the mean. Thus, it is the proportion of variation in the response (in this case, the traffic volume at intersection 2) that is eliminated or statistically explained by a least-squares line through the scatter.

3-D Scatter Plots

A three-dimensional scatter plot, created with the scatter3 function, shows the relationship between the traffic volume at all three intersections. Use the variables c1, c2, and c3 that you created in the previous step:

figure c3 = count(:,3); % Data at intersection 3 scatter3(c1,c2,c3,'filled') xlabel('Intersection 1') ylabel('Intersection 2') zlabel('Intersection 3')



Measure the strength of the linear relationship among the variables in the three-dimensional scatter by computing eigenvalues of the covariance matrix with the eig function:

vars = eig(cov([c1 c2 c3]))

vars =

1.0e+03 \*

0.0442

0.1118

6.8300

explained = max(vars)/sum(vars)

explained =

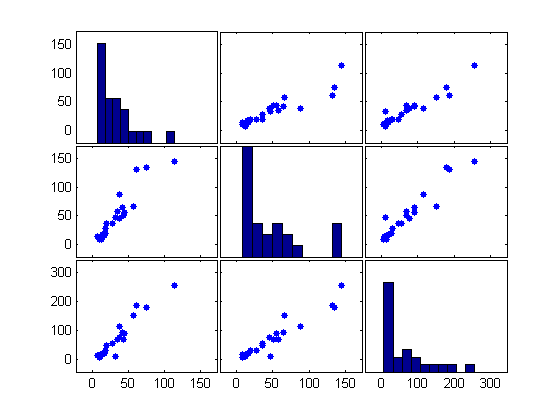
0.9777

The eigenvalues are the variances along the principal components of the data. The variable explained measures the proportion of variation explained by the first principal component, along the axis of the data. Unlike the coefficient of determination for two-dimensional scatters, this measure distinguishes predictor and response variables.

Scatter Plot Arrays

Use the plotmatrix function to make comparisons of the relationships between multiple pairs of intersections:

figure plotmatrix(count)

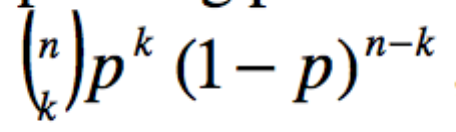
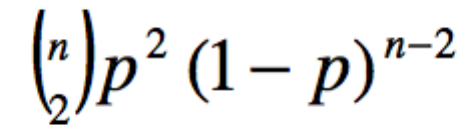


The plot in the (i, j)th position of the array is a scatter with the ith variable on the vertical axis and the jth variable on the horizontal axis. The plot in the ith diagonal position is a histogram of the ith variable.

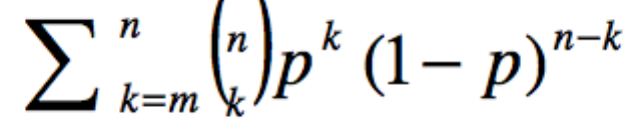
## More on Matlab functions and loops

● **Nested For Loops Example**. To go in a different direction, consider a type of trial or experiment that is repeated n times, each trial independent, with a probability p of success each time, and of course probability 1-p of failure. What is the probability of at least m successes in n trials? The answer involves the binomial distribution and works like this. The probability of no successes in n trials is (1-p)n, since the trials are independent. What is the probability of exactly 1 success? There are n ways that one success could occur, one for each repeat, and the probability of each is p. So the probability of 1 success is np(1-p)n-1 (independence again).

How ***many ways*** can there be 2 successes? Or more generally, how many ways can there be k successes? tbinom nk or tfrac{n!}{k!\,(n-k)!}

The probability of 2 successes is: and the general probability of k successes is:

Therefore the probability of at least m successes is:



The task at hand is simply to compute these numbers by scripts and loops.

As a first pass, fix m and n and just do the sum. It's a simple sum again and here is one way to do the coding.

% binom1.m % Binomial distribution, at least m successes in n tries with prob p

clear

p=.5;n=3;m=3;c=0;

for k=m:n c=c+(nchoosek(n,k)\*(p^k)\*((1-p)^(n-k)));

end

Going further, how would you code this for general n and m? Since there are loops for m, n and then the running index k, it looks like 3 nested loops. That's what it turns out to be and here is some acceptable code.

% binom2.m % Binomial distribution, at least m successes in n tries with prob p % Full matrix of successes with N trials

clear

p=.5;N=4;c=zeros(N,N);

for n=1:N

for m=1:n

for k=m:n c(m,n)=c(m,n)+(nchoosek(n,k)\*(p^k)\*((1-p)^(n-k)));

end

end

end

[ones(1,N);c]

Try p = 1/2 and N = 4.

The command ones(1,N) is to fill in the top row corresponding to m=0. Matlab does not allow the use of 0 as an index in a matrix c(n,m). For this problem the fix is easy because the binomial coefficients nchoosek are all 1 for k=0 (or m=0 for the top row). The code can avoid this by shifting indices, but for this tutorial it seemed best to take the easy solution.

You should check a few simple cases by hand.

● **Functions**. There is an extensive array of built-in ML functions including exp, log, sin, cos, etc; a partial list was given earlier. However, the user can create and store individualized functions, which typically are for use in special purpose coding.

To create a function, open the editor and comment in some descriptive language about what the function does. Then the actual commands are very simple (see manual p 6-21). To create the

function *f* (*x*) *x*2 1 , for example, use this code.

% Function rt\_x\_sq\_plus1 % Special function example

function y=rt\_x\_sq\_plus1(x)

y=sqrt((x.\*x)+1);

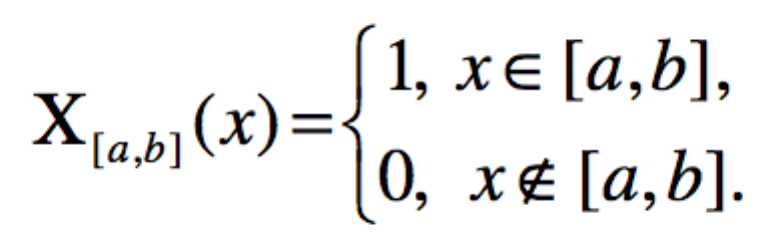
Save the M-file (in the correct directory and with the right path) rt\_x\_sq\_plus1.m.

Now you can make direct calls to the function from the command line, or from other M-files. From the command line it would look like this.

x=-10:.1:10;y=rt\_x\_sq\_plus1(x);plot(x,y,’r’)

Note the red color, resulting from the command plot(x,y,'r').

● **More Function Examples**. A handy plotting tool to have is a "window" function that selects a given interval [a,b] on the real line. It's sometimes called the characteristic function or indicator function of the interval and is defined as follows.



Here is an M-file for an arbitrary window function.

% Window function on [a,b]

function y=winab(a,b,x) y=.5\*(sign(x-a)-sign(x-b));

This is saved as winab.m (appropriate folder and path of course). Here's what it looks like from the command line.

EDU>> x=-2:.01:2;y=winab(-1,1,x);plot(x,y), axis([-2 2 0,2])

To construct a piecewise function, all one has to do is sum up a set of desired functions multiplied by appropriate windows.

% plotfcn1 % Plot a function using the window

x=-3:.1:3; y=winab(-3,-1,x).\*sin(x)+winab(-1,1,x).\*cos(x)+winab(1,3,x).\*sin(x); plot(x,y)

Here's a way to take a given function and chop off everything except a window portion (or to filter the function).

% plotfcn2 % Plot a function using the window x=0:.01:10; y=winab(0,4,x).\*x.\*exp(-x); plot(x,y) axis([0 10 -.5 .5])

● **Bisection Method.** Bisection is the simplest method for finding roots for equations f(x) = 0 where f(x) is continuous. The idea is simple. By trial and error find points a and b where f(a) and f(b) have opposite signs. By the intermediate value theorem there is a root between a and b. Take the midpoint c = (a + b)/2 = a + ((b-a)/2) and compare f(c) with f(a). If they have opposite signs then a root lies in (a,c); otherwise a root lies in (c,b). Iterate the procedure, each time trapping a root within an interval of width (b-a)/2n. Of course, the code has to keep track of the location. One can set a tolerance variable ε and stop execution when the trapping width is less than ε.

As an example, consider f(x) = cos(x) - x. A homework problem given later is to find a root of cos(x) = x by iteration. This root will now be found by bisection. The code uses the If command in ML, p 6-2. The if command is a conditional statement which can be followed by an alternative ("else") or a termination. The code is below; it uses a for loop but this is unnecessary.

First define a function M-file for cos(x) - x.

% csxx(x).m % Function csxx(x)=cos(x)-x defined

function y=csxx(x) y=cos(x)-x;

Now write script to call the function and perform bisection.

% rootcsxx.m

% Bisection method for cos(x)-x=0

clear

a=0;b=1;N=20;tol=.0001;

for n=1:N

x=a+(b-a)/2;

if csxx(a)\*csxx(x)>0

a=x;

else

b=x;

end

[n a b x cos(x)]

if (b-a)/2<tol

break end

end

EDU>> rootcsxx

This code set N = 20 iterations but put in a break conditional using the If command. The break stopped execution at n = 13. The strategy of the code is exactly the implementation of the bisection method.

**Miscellaneous Remarks**.

'Anonymous' functions are built-in ML functions that are path independent. They may be called/executed from any directory. These include the standard elementary and trig functions such as sin, cos, sqrt, sqr (square), exp, log, etc.

A common problem is a call to script using undefined variables. You may know what the variable is but it may not be accessible to a call in an M-file because of the path. One way to overcome this is to use global variables (p 6-25). For example, plotting a superposition of arbitrary Gaussians could be done by defining a function arbgauss as follows.

% arbgauss % to define an arbitrary Gaussian function

function y=arbgauss(x)

global mu

global sig

  y=exp(-(x-mu).^2/(2\*sig\*sig));

Then a typical plotting call might look like this.

% plotarbgauss

% plot superposition of Gaussians

global mu

global sig

mu=2;sig=.5;

x=-8:.1:8;y=arbgauss(x);

plot(x,y)

hold

mu=-2;sig=1.5;

x=-8:.1:8;y=arbgauss(x);

plot(x,y,'.')

hold

And here's the command line.

EDU>> plotarbgauss Current plot held Current plot released

1 0.9 0.8 0.7 0.6 0.5 0.4 0.3 0.2 0.1 0

-8 -6 -4 -2 0 2 4 6 8

This allows the user to make changes in the mu (mean) and sig (variance) variables in the executable file only, and not in the function file.

Advanced topics such as function handles and vectorization are in the manual but will not be covered in these notes.

# R and RStudio

## How to think about R

Although we don’t have enough time to delve deeply into all aspects of R, what I hope to push you towards is to develop your own strategies and tactics to struggle through (successfully!) using R to implement analysis using unfamiliar techniques (packages).

R is, in a sense, an object-oriented language

1. Everything in R is considered an object
2. R supports encapsulation (grouping distinct but related data items into a single object class instance)
3. R is polymorphic (a function call can lead to different operations for objects of different classes enabling generic functions)
4. R allows inheritance (extending a given class to a more specialized class)

Examples of generic functions: print(), str(), summary()

Objects in R are instances of R classes. An R class is simply a definition of elements that make up the object and the methods (operations) you can perform on that object. There are multiple ways to discover the class of an object but the simplest is by using the class() function.

Classes are used by generic functions. A generic function is a actually a family of functions like summary(). Summary() provides output *tailored* to the specific class of data passed to the function.

An S3 class (still the dominant class paradigm in R) consists of a list of class names and dispatch capabilities (for each class name there is a dispatch capability – dispatch capability is analogous to method or operation). For example, if you type print on the command line, you can see (below) that the print function is simply a call to UseMethod(“print”).

> print

function (x, ...)

UseMethod("print")

<bytecode: 0x7fc1059c4fb8>

<environment: namespace:base>

Based on the packages I have loaded, there are 220 methods available to print. I’ve just provided a sample below and notice that ggplot has its own print method.

> methods(print)

[1] print.abbrev\*

[2] print.acf\*

[3] print.AES\*

[4] print.anova

…

[92] print.ggplot\*

[93] print.gList\*

[94] print.glm

…

[115] print.lm

…

[188] print.summary.lm\*

[189] print.summary.loess\*

…

[202] print.table

[203] print.tables\_aov\*

[204] print.terms

…

[217] print.warnings

[218] print.xgettext\*

[219] print.xngettext\*

[220] print.xtabs\*

Non-visible functions are asterisked

Why should you care? Primarily because I don’t need to remember or look-up 220 different print functions based on the object I would like to print.

Referring to your future ggplot homework, this is important because the parameters for the print command depend on the method being called (which of course is associated with the object instance passed to the generic function).

Compare the help panels for these two calls:

?print

?print.ggplot

The ggplot print method includes the vp= parameter. To successfully complete your homework, you need to create the “what” (these are the three plot variables created using ggplot constructs) and then the “where” (these are the viewport definitions created using grid constructs). The vp= parameter refers to the “where”.

Every object in R has a definition and we can find out what’s included in an instance of an object using the str() function. This even works to some extent to view the parameters of a function but, ?<function\_name> works better if the documentation is decent.

Lets look at the structure of an ***instance of*** a plot object created using ggplot (at the very bottom you also notice that the class of this object is “gg” “ggplot”):

# Load grid package

require("grid")

# Bring Diamonds data into memory

data(diamonds)

# Plot relationship of carat on price by color

carat\_prc <- ggplot(diamonds, aes(x=carat,y=price))

carat\_prc + labs(title="Diamonds - Weight to Price by Color") +

theme(plot.title = element\_text(size = rel(2), colour = "blue")) +

labs(x="Weight", y="Price") +

geom\_point(aes(color = factor(color)))

> str(carat\_prc)

List of 9

$ data :'data.frame': 53940 obs. of 13 variables:

..$ carat : num [1:53940] 0.23 0.21 0.23 0.29 0.31 0.24 0.24 0.26 0.22 0.23 ...

..$ cut : Ord.factor w/ 5 levels "Fair"<"Good"<..: 5 4 2 4 2 3 3 3 1 3 ...

..$ color : Ord.factor w/ 7 levels "D"<"E"<"F"<"G"<..: 2 2 2 6 7 7 6 5 2 5 ...

..$ clarity : Ord.factor w/ 8 levels "I1"<"SI2"<"SI1"<..: 2 3 5 4 2 6 7 3 4 5 ...

..$ depth : num [1:53940] 61.5 59.8 56.9 62.4 63.3 62.8 62.3 61.9 65.1 59.4 ...

..$ table : num [1:53940] 55 61 65 58 58 57 57 55 61 61 ...

..$ price : int [1:53940] 326 326 327 334 335 336 336 337 337 338 ...

..$ x : num [1:53940] 3.95 3.89 4.05 4.2 4.34 3.94 3.95 4.07 3.87 4 ...

..$ y : num [1:53940] 3.98 3.84 4.07 4.23 4.35 3.96 3.98 4.11 3.78 4.05 ...

..$ z : num [1:53940] 2.43 2.31 2.31 2.63 2.75 2.48 2.47 2.53 2.49 2.39 ...

..$ logprice: num [1:53940] 5.79 5.79 5.79 5.81 5.81 ...

..$ logcarat: num [1:53940] -1.47 -1.56 -1.47 -1.24 -1.17 ...

..$ presids : num [1:53940] -0.1989 -0.0464 -0.1958 -0.5631 -0.6718 ...

$ layers :List of 1

..$ :Classes 'proto', 'environment' <environment: 0x7fc109034200>

$ scales :Reference class 'Scales' [package "ggplot2"] with 1 fields

..$ scales: list()

..and 21 methods, of which 9 are possibly relevant:

.. add, clone, find, get\_scales, has\_scale,

.. initialize, input, n, non\_position\_scales

$ mapping :List of 2

..$ x: symbol logcarat

..$ y: symbol presids

$ theme :List of 2

..$ plot.title :List of 8

.. ..$ family : NULL

.. ..$ face : NULL

.. ..$ colour : chr "blue"

.. ..$ size :Class 'rel' num 2

.. ..$ hjust : NULL

.. ..$ vjust : NULL

.. ..$ angle : NULL

.. ..$ lineheight: NULL

.. ..- attr(\*, "class")= chr [1:2] "element\_text" "element"

..$ legend.position: chr "top"

..- attr(\*, "class")= chr [1:2] "theme" "gg"

..- attr(\*, "complete")= logi FALSE

$ coordinates:List of 1

..$ limits:List of 2

.. ..$ x: NULL

.. ..$ y: NULL

..- attr(\*, "class")= chr [1:2] "cartesian" "coord"

$ facet :List of 1

..$ shrink: logi TRUE

..- attr(\*, "class")= chr [1:2] "null" "facet"

$ plot\_env :<environment: R\_GlobalEnv>

$ labels :List of 4

..$ x : chr "Weight"

..$ y : chr "Price Residuals"

..$ title : chr "Diamonds - Weight to Price by Color"

..$ colour: chr "factor(color)"

* attr(\*, "class")= chr [1:2] "gg" "ggplot"

Why should you care? From the questions I receive, it seems people struggle with how to think about R conceptually. The “grammar” of graphics from an R perspective is really about the process of setting attributes in a ggplot object instance to control the rendering of a plot (what a plot looks like). Because you can build plots in layers, the order of the process matters and can help simplify the task. The grid package provides a set of functions and object classes (viewport being the primary object class) to position plots within a space. In the same sense, positioning objects is about learning the process of setting the attributes and calling the proper functions to set up the plotting space.

Remember to use start small, fail fast as an approach understanding how things work in any new programming language. For example, run some simple experiments when using a new function or data structure before trying to combine its use in a larger script or function. Once you understand each atomic element, then it’s much easier to combine into a larger solution.

Hypothetically, you may need the grid package in an upcoming assignment. Here’s one approach to exploring grid:

Grid documentation is easy to access in R-Studio by selecting the Packages tab (in the pane with Files, Help, Plots, Viewer tabs). Find the grid package and click on the link. This pulls up the Help Tab with the following (partial):

Documentation for package ‘grid’ version 3.0.2

* [DESCRIPTION file](http://127.0.0.1:38521/help/library/grid/DESCRIPTION).
* [User guides, package vignettes and other documentation.](http://127.0.0.1:38521/help/library/grid/doc/index.html)

Click on the link “User guides, package vignettes and other documentation”, and this page appears:

Vignettes from package 'grid'

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| [grid::displaylist](http://127.0.0.1:38521/help/library/grid/doc/displaylist.pdf) |  | Display Lists in grid | [PDF](http://127.0.0.1:38521/help/library/grid/doc/displaylist.pdf) | [source](http://127.0.0.1:38521/help/library/grid/doc/displaylist.Rnw) | [R code](http://127.0.0.1:38521/help/library/grid/doc/displaylist.R) |
| [grid::frame](http://127.0.0.1:38521/help/library/grid/doc/frame.pdf) |  | Frames and packing grobs | [PDF](http://127.0.0.1:38521/help/library/grid/doc/frame.pdf) | [source](http://127.0.0.1:38521/help/library/grid/doc/frame.Rnw) | [R code](http://127.0.0.1:38521/help/library/grid/doc/frame.R) |
| [grid::grid](http://127.0.0.1:38521/help/library/grid/doc/grid.pdf) |  | Introduction to grid | [PDF](http://127.0.0.1:38521/help/library/grid/doc/grid.pdf) | [source](http://127.0.0.1:38521/help/library/grid/doc/grid.Rnw) | [R code](http://127.0.0.1:38521/help/library/grid/doc/grid.R) |
| [grid::grobs](http://127.0.0.1:38521/help/library/grid/doc/grobs.pdf) |  | Working with grid grobs | [PDF](http://127.0.0.1:38521/help/library/grid/doc/grobs.pdf) | [source](http://127.0.0.1:38521/help/library/grid/doc/grobs.Rnw) | [R code](http://127.0.0.1:38521/help/library/grid/doc/grobs.R) |
| [grid::interactive](http://127.0.0.1:38521/help/library/grid/doc/interactive.pdf) |  | Editing grid Graphics | [PDF](http://127.0.0.1:38521/help/library/grid/doc/interactive.pdf) | [source](http://127.0.0.1:38521/help/library/grid/doc/interactive.Rnw) | [R code](http://127.0.0.1:38521/help/library/grid/doc/interactive.R) |
| [grid::locndimn](http://127.0.0.1:38521/help/library/grid/doc/locndimn.pdf) |  | Locations versus Dimensions | [PDF](http://127.0.0.1:38521/help/library/grid/doc/locndimn.pdf) | [source](http://127.0.0.1:38521/help/library/grid/doc/locndimn.Rnw) | [R code](http://127.0.0.1:38521/help/library/grid/doc/locndimn.R) |
| [grid::moveline](http://127.0.0.1:38521/help/library/grid/doc/moveline.pdf) |  | Demonstrating move-to and line-to | [PDF](http://127.0.0.1:38521/help/library/grid/doc/moveline.pdf) | [source](http://127.0.0.1:38521/help/library/grid/doc/moveline.Rnw) | [R code](http://127.0.0.1:38521/help/library/grid/doc/moveline.R) |
| [grid::nonfinite](http://127.0.0.1:38521/help/library/grid/doc/nonfinite.pdf) |  | Non-finite values | [PDF](http://127.0.0.1:38521/help/library/grid/doc/nonfinite.pdf) | [source](http://127.0.0.1:38521/help/library/grid/doc/nonfinite.Rnw) | [R code](http://127.0.0.1:38521/help/library/grid/doc/nonfinite.R) |
| [grid::plotexample](http://127.0.0.1:38521/help/library/grid/doc/plotexample.pdf) |  | Writing grid Code | [PDF](http://127.0.0.1:38521/help/library/grid/doc/plotexample.pdf) | [source](http://127.0.0.1:38521/help/library/grid/doc/plotexample.Rnw) | [R code](http://127.0.0.1:38521/help/library/grid/doc/plotexample.R) |
| [grid::rotated](http://127.0.0.1:38521/help/library/grid/doc/rotated.pdf) |  | Rotated Viewports | [PDF](http://127.0.0.1:38521/help/library/grid/doc/rotated.pdf) | [source](http://127.0.0.1:38521/help/library/grid/doc/rotated.Rnw) | [R code](http://127.0.0.1:38521/help/library/grid/doc/rotated.R) |
| [grid::saveload](http://127.0.0.1:38521/help/library/grid/doc/saveload.pdf) |  | Persistent representations | [PDF](http://127.0.0.1:38521/help/library/grid/doc/saveload.pdf) | [source](http://127.0.0.1:38521/help/library/grid/doc/saveload.Rnw) | [R code](http://127.0.0.1:38521/help/library/grid/doc/saveload.R) |
| [grid::esharing](http://127.0.0.1:38521/help/library/grid/doc/sharing.pdf) |  | Modifying multiple grobs simultaneously | [PDF](http://127.0.0.1:38521/help/library/grid/doc/sharing.pdf) | [source](http://127.0.0.1:38521/help/library/grid/doc/sharing.Rnw) | [R code](http://127.0.0.1:38521/help/library/grid/doc/sharing.R) |
| [grid::viewports](http://127.0.0.1:38521/help/library/grid/doc/viewports.pdf) |  | Working with viewports | [PDF](http://127.0.0.1:38521/help/library/grid/doc/viewports.pdf) | [source](http://127.0.0.1:38521/help/library/grid/doc/viewports.Rnw) | [R code](http://127.0.0.1:38521/help/library/grid/doc/viewports.R) |

(not included here but part of the help panel are links to pdf, source, r-code.

The first thing that catches my attention is [grid::plotexample](http://127.0.0.1:38521/help/library/grid/doc/plotexample.pdf). If you pull up the pdf, there are some examples you can execute to gain confidence in the use of grid.

## NA and NULL in R

### A note about NULL and NA

**NA** is equivalent to “missing”. Data is unknown for a particular observation.

**NULL** is analogous to “doesn’t exist”

Work through these examples until it clicks for you.

x1 <- c(88, NA, 12, 168, 13)

x2 <- c(88, NULL, 12, 168, 13)

mean(x1)

mean(x2)

mean(x1, na.rm=TRUE)

# Initialize z1 to NULL and put even numbers into a vector

z1 <- NULL

for (i in 1:20) if (i %%2 == 0) z <- c(z1,i)

z1

## Exploring Implicit loops in R

Building on the previous discussion of objects, because R uses generic functions that use class specific methods (or Dispatch Capabilities) to operate on objects, the methods can “assume” that you want to apply an operation to an entire vector without have to write an explicit loop to traverse through each element of the vector or matrix or whatever class of object you’re dealing with at the time.

# Two implementations of the same problem

# Explicit looping

b1 <- c(3, 4, 8, 23, 25, 7, 1, 23, 1, 9, 14, 22, 1, 0, 32)

first1 <- function(x) {

for (i in 1:length(x)) { if (x[i] == 1) break #break out of loop

}

return(i)

}

first1(b1)

# take advantage of implicit looping in the which function

# the disadvantage is that the function will cycle through

# the entire vector to create a boolean values and then return

# the index of the first place where the condition is true

# For a very large vector, this might be less efficient

# than an explicit loop that terminates when the first

# value is found.

first1a <- function(x) return (which (x == 1)[1])

first1a(b1)

In addition to the standard if – else constructs found in other languages, R also has a vectorized if-then-else function called ifelse().

Here’s an example:

c1 <- 1:10

c2 <- ifelse(c1 %% 2 == 0, "EVEN", "ODD")

c2

What happens here is that the expression c1 %%2 == 0 results in a vector:

[F, T, F, T, F, T, F, T, F, T]. The second argument is treated as a vector [“EVEN” , “EVEN”, .., “EVEN”] with ten elements created by recycling. The third argument is treated as a vector [“ODD”, “ODD”, .., “ODD”] with ten elements also created by recycling.

Here’s another example:

c3 <- c(9,2,12,5,16,4)

c4 <- ifelse(c3 > 6, 2\*c3, 3\*c3)

c4

Now let’s look at a measure of association based on Kendall’s Tau.

# Using ifelse to create an association measure based

# on made up data.

AirTemp <- c(5, 12, 13, 3, 6, 0, 1, 15, 16, 8, 88)

Pressure <- c(4, 2, 3, 23, 6, 10, 11, 12, 6, 3, 2)

# The measure is defined as the proportion of time

# that AirTemp and Presure move in the same direction

# findud() converts vector v to 1s and -1s representing

# whether an element is increasing or decreasing relative

# to the previous element. Output length will necessarily

# be one less than input. v[-1] is a vector of 15 elements

# starting with the second element (removes 1st element)

# v[-length(v)] is also a vector of 15 elements starting

# with the first element but removing the last.

# Then use ifelse to create a 1 for a positive change

# and a -1 for a negative change

findud <- function(v) {

vud <- v[-1] - v[-length(v)]

return (ifelse(vud > 0, 1, -1))

}

# Instead of making two separate function calls, use

# lapply to "apply" the findud function to the two

# vectors x and Y. ud[[1]] == ud[[2]] returns a boolean

# vector which is treated as 1's and 0's by the function

# mean, i.e. the mean of 4 1's and 6 0's is 0.4 or the

# proportion of 1's contained in the vector.

udcorr <- function (x,y) {

ud <- lapply(list(x,y), findud)

print(ud)

return(mean(ud[[1]] == ud[[2]]))

}

prop1 <- udcorr(AirTemp, Pressure)

prop1

**Investigate the sign and diff functions to see why this works.**

# A more compact way to accomplish the same task.

udcorr2 <- function(x,y) mean(sign(diff(x)) ==

sign(diff(y)))

prop2 <- udcorr2(AirTemp, Pressure)

prop2

**Following is an example of how to get a dataset from the web, load it into R and begin exploring the data.**

# One more problem to work through here. UCI is a great

# resource for datasets. Here's an example of how to

# retrieve the Abalone data

uciaddress <- "http://archive.ics.uci.edu/ml/machine-learning-databases/"

dataset <- "abalone/abalone.data"

getdataset <- paste(uciaddress, dataset, sep="")

abalone <- read.csv(getdataset)

colnames(abalone) <- c("Gender","Length","Diameter",

"Height", "Whole\_wgt",

"Shucked\_wgt", "Viscera wgt",

"Shell wgt", "Rings")

**# Quick way to get a sample from a dataframe**

randomSample = function(dataframe,n,replacement) {

return (dataframe[sample(nrow(dataframe), n, replace=replacement),]) }

absample<-randomSample(abalone, 40, FALSE)

# Let's recode the gender attribute to demonstrate using nested ifelse().

# We might want to do this to use Gender in a clustering algorithm.

# What are the actual argument names for ifelse

args(ifelse)

absample$GenderInt <- ifelse(absample$Gender == 'M', 1,

ifelse(absample$Gender == 'F', 2, 3))

absample$GenderInt

absample$Gender

# Find the mean number of rings for each gender type

?tapply

tapply(absample$Rings, absample$Gender, mean)

table(list(absample$Gender, absample$Rings))

# How good is the sample relative to whole\_wgt? How

# would you measure?

summary(absample$Whole\_wgt)

summary(abalone$Whole\_wgt)

# Why is the shell wgt reference different?

summary(absample$"Shell wgt")

# Create a new variable size as a binning of Whole\_wgt

absample$Size <- ifelse(absample$Whole\_wgt < 0.85,

"SMALL", "LARGE")

table(list(absample$Gender, absample$Size))

# Create a basic plot by weight

Wgt\_Gender <- ggplot(absample, aes(x=Diameter,y=Length))

Wgt\_Gender <- Wgt\_Gender + labs(title="Diameter vs Length by Gender") +

theme(plot.title = element\_text(size = rel(2), colour = "blue")) +

labs(x="Diameter", y="Length") +

geom\_point(aes(colour = factor(Gender)))

print(Wgt\_Gender)

# Subset the males from the sample

malesample <- absample[which(absample$Gender == 'M'),]

str(malesample)

# Subset the females from the sample

femalesample <- absample[which(absample$Gender == 'F'),]

str(femalesample)

# Now plot the subsets on the same plot using

# different colors for Gender

Wgt\_Gender2 <- ggplot(absample, aes(x=Diameter,y=Length))

Wgt\_Gender2 <- Wgt\_Gender2 + labs(title="Diameter vs Length by Gender") +

theme(plot.title = element\_text(size = rel(2), colour = "blue")) +

labs(x="Diameter", y="Length") +

geom\_point(data=femalesample, color = "red", size = 3) +

geom\_point(data=malesample, color = "blue", size = 3)

print(Wgt\_Gender2)

## R Statistical Plotting using ggPlot

**Statistical Graphing in R**

**Reference for Slides (Wickham 2009)**

**Math 510 - Page 2**

### Plotting in R

**R's base graphics**: implemented in the same way as in the Bell Labs S3 system developed by Becker, Chambers, and Wilks.

**grid** was designed by Paul Murrell to overcome some of these limitations and as a result packages like lattice, **ggplot2**, vcd or hexbin (on Bioconductor) use grid for the underlying primitives.

**Math 510 - Page 3**

### Plotting in R – Many Applications

**Plotting**: plotrix, vcd, hexbin (on Bioconductor ), gclus and gplots

**Graphic Applications:** This area aids in creating effective displays.

* Effect ordering: gclus,cba and seriation
* Large Data Sets: ash, hexbin, scagnostics
* Trees and Graphs: ape, ade4, igraph, diagram, Rgraphviz

**Graphics Systems:** lattice, **ggplot2**

**Devices:** cairoDevice, RGtk2, RSvgDevice, rgl, JavaGD.

**Colors:** colorspace, RColorBrewer. dichromat

**Interactive Graphics:** rggobi (Ggobi), iplots, iwidgets, playwith

**Development:** grid, rgl, RoSuDA, gridBase

CRAN Task View: Graphic Displays & Dynamic Graphics & Graphic Devices & Visualization

**Math 510 - Page 4**

### ggPlot2

• Based on the grammar of graphics (Wilkinson,2005)

• Hadley Wickham(2005+): Primacy of layers and adaptation for embedding in R

• Grammar tells us that a statistical graphic, drawn on a coordinate system, is a mapping from data to aesthetic attributes (color, shape, size) of geometric objects (points, lines, bars) potentially containing statistical transformations and facets of the data.

**Math 510 - Page 5**

### So what?

**Most graphic packages are a collection of special cases**

• ggplot

• Designed to be layered

• Produce graphics using structured thinking

• Encourages the creation of totally new graphics

• Easily combine existing plots

• Represent multiple data sets

**Math 510 - Page 6**

### ggPlot2 Grammar

• data

• aestheticmappings

• geoms

• stats

• scales

• Coord

**Math 510 - Page 7**

### What is a plot?

1. Default dataset and mappings from variables to aesthetics

2. One or more layers

* Geom
* Stat
* Position adjustment
* Dataset
* Aesthetic mappings (optional)

3. Scale

4. Coordinate system

5. Faceting specification

**Math 510 - Page 8**

### Layers

Responsible for creating objects perceived on a plot

• Data and aesthetic mapping

• Statistical transformation (stat) • Geometric object (geom)

• Position adjustment

**Math 510 - Page 9**

### Scales

• Controls mapping from data to aesthetic attributes

• Operates across all data in the plot

• Ensures consistent mapping

• Example: Color Gradient – Maps a segment of a real line to a path through the color space

**Math 510 - Page 10**

### Coordinate system

Maps the position of objects onto the plane of the plot.

• Cartesian most common

• Polar

• Various map projections

• Affect all position variables simultaneously

• Change the appearance of geometric objects

• Scaling is performed before statistical transformation, coordinate transformations after.

**Math 510 - Page 11**

### Facets

• General case of conditioned or trellised plots

• Makes it easy to plot different subsets of the data

**Math 510 - Page 12**

### Rendering

• Print()–Renders to the screen

• ggsave()–Renders to a disk

• save()–Saves a cached copy (including data!)

**Math 510 - Page 13**

### Geoms

geom\_abline geom\_area geom\_bar geom\_bin2d geom\_blank geom\_boxplot geom\_contour geom\_crossbar geom\_density geom\_density2d geom\_dotplot geom\_errorbar geom\_errorbarh

geom\_freqpoly geom\_hex geom\_histogram geom\_hline geom\_jitter geom\_line geom\_linerange geom\_map geom\_path geom\_point geom\_pointrange geom\_polygon geom\_quantile

geom\_raster geom\_rect geom\_ribbon geom\_rug geom\_segment geom\_smooth geom\_step geom\_text geom\_tile geom\_violin geom\_vline

**Math 510 - Page 14**

### stats

stat\_bin stat\_bin2d stat\_bindot stat\_binhex stat\_boxplot stat\_contour stat\_density stat\_density2d stat\_ecdf stat\_function stat\_identity stat\_qq stat\_quantile

stat\_smooth stat\_spoke stat\_sum stat\_summary stat\_summary\_hex stat\_summary2d stat\_unique stat\_ydensity

**Math 510 - Page 15**

**scales**

expand\_limits

guide\_legend

guide\_colourbar(guide\_colorbar)

scale\_alpha(scale\_alpha\_continuous, scale\_alpha\_discrete)

scale\_area

scale\_colour\_brewer(scale\_color\_brewer, scale\_fill\_brewer)

scale\_colour\_gradient(scale\_color\_continuous, scale\_color\_gradient, scale\_colour\_continuous, scale\_fill\_continuous, scale\_fill\_gradient)

scale\_colour\_gradient2(scale\_color\_gradient2, scale\_fill\_gradient2) scale\_colour\_gradientn(scale\_color\_gradientn, scale\_fill\_gradientn) scale\_colour\_grey(scale\_color\_grey, scale\_fill\_grey)

scale\_colour\_hue(scale\_color\_discrete, scale\_color\_hue, scale\_colour\_discrete, scale\_fill\_discrete, scale\_fill\_hue)

scale\_identity(scale\_alpha\_identity, scale\_color\_identity, scale\_colour\_identity, scale\_fill\_identity, scale\_linetype\_identity,

scale\_shape\_identity, scale\_size\_identity)

scale\_manual(scale\_alpha\_manual, scale\_color\_manual, scale\_colour\_manual, scale\_fill\_manual, scale\_linetype\_manual, scale\_shape\_manual, scale\_size\_manual)

scale\_linetype(scale\_linetype\_continuous, scale\_linetype\_discrete)

scale\_shape(scale\_shape\_continuous, scale\_shape\_discrete)

scale\_size(scale\_size\_continuous, scale\_size\_discrete)

scale\_x\_continuous(scale\_x\_log10, scale\_x\_reverse, scale\_x\_sqrt, scale\_y\_continuous, scale\_y\_log10, scale\_y\_reverse, scale\_y\_sqrt)

scale\_x\_date(scale\_y\_date) scale\_x\_datetime(scale\_y\_datetime) scale\_x\_discrete(scale\_y\_discrete) labs(ggtitle, xlab, ylab) update\_labels

xlim(ylim)

**Math 510 - Page 16**

## R Conditional logic, loops, and functions (Chapters 9 and 10 from an introduction to R)

### Grouped Expressions

R is an expression language in the sense that its only command type is a function or expression which returns a result. Even an assignment is an expression whose result is the value assigned, and it may be used wherever any expression may be used; in particular multiple assignments are possible.

Commands may be grouped together in braces, {expr\_1; ...; expr\_m}, in which case the value of the group is the result of the last expression in the group evaluated. Since such a group is also an expression it may, for example, be itself included in parentheses and used a part of an even larger expression, and so on.

### Conditionals

The language has available a conditional construction of the form

> if (expr\_1) expr\_2 else expr\_3

where expr 1 must evaluate to a single logical value and the result of the entire expression is

then evident.

The “short-circuit” operators && and || are often used as part of the condition in an if statement. Whereas & and | apply element-wise to vectors, && and || apply to vectors of length one, and only evaluate their second argument if necessary.

There is a vectorized version of the if/else construct, the ifelse function. This has the form ifelse(condition, a, b) and returns a vector of the length of its longest argument, with elements a[i] if condition[i] is true, otherwise b[i].

### Loops

There is also a for loop construction which has the form

> for (name in expr\_1) expr\_2

where name is the loop variable. expr 1 is a vector expression, (often a sequence like 1:20), and expr 2 is often a grouped expression with its sub-expressions written in terms of the dummy name. expr 2 is repeatedly evaluated as name ranges through the values in the vector result of expr 1.

As an example, suppose ind is a vector of class indicators and we wish to produce separate plots of y versus x within classes. One possibility here is to use coplot(),1 which will produce an array of plots corresponding to each level of the factor. Another way to do this, now putting all plots on the one display, is as follows:

> xc <- split(x, ind)

> yc <- split(y, ind)

> for (i in 1:length(yc)) {

plot(xc[[i]], yc[[i]])

abline(lsfit(xc[[i]], yc[[i]]))

}

(Note the function split() which produces a list of vectors obtained by splitting a larger vector according to the classes specified by a factor. This is a useful function, mostly used in connection with boxplots. See the help facility for further details.)

Warning: for() loops are used in R code much less often than in compiled languages. Code that takes a ‘whole object’ view is likely to be both clearer and faster in R.

Other looping facilities include the

> repeat expr statement and the

> while (condition) expr statement.

The break statement can be used to terminate any loop, possibly abnormally. This is the only way to terminate repeat loops.

The next statement can be used to discontinue one particular cycle and skip to the “next”.

### Functions

As we have seen informally along the way, the R language allows the user to create objects of mode function. These are true R functions that are stored in a special internal form and may be used in further expressions and so on. In the process, the language gains enormously in power, convenience and elegance, and learning to write useful functions is one of the main ways to make your use of R comfortable and productive.

It should be emphasized that most of the functions supplied as part of the R system, such as mean(), var(), postscript() and so on, are themselves written in R and thus do not differ materially from user written functions.

A function is defined by an assignment of the form

> name <- function(arg\_1, arg\_2, ...) expression

The expression is an R expression, (usually a grouped expression), that uses the arguments, arg i, to calculate a value. The value of the expression is the value returned for the function. A call to the function then usually takes the form name(expr\_1, expr\_2, ...) and may occur anywhere a function call is legitimate.

### Function Examples

As a first example, consider a function to calculate the two sample t-statistic, showing “all the steps”. This is an artificial example, of course, since there are other, simpler ways of achieving the same end.

The function is defined as follows:

> twosam <- function(y1, y2) {

n1 <- length(y1); n2 <- length(y2)

yb1 <- mean(y1); yb2 <- mean(y2)

s1 <- var(y1); s2 <- var(y2)

s <- ((n1-1)\*s1 + (n2-1)\*s2)/(n1+n2-2)

tst <- (yb1 - yb2)/sqrt(s\*(1/n1 + 1/n2))

tst

}

With this function defined, you could perform two sample t-tests using a call such as > tstat <- twosam(data$male, data$female); tstat

As a second example, consider a function to emulate directly the Matlab backslash com- mand, which returns the coefficients of the orthogonal projection of the vector y onto the column space of the matrix, X. (This is ordinarily called the least squares estimate of the regression coefficients.) This would ordinarily be done with the qr() function; however this is sometimes a bit tricky to use directly and it pays to have a simple function such as the following to use it safely.

Thus given a n by 1 vector y and an n by p matrix X then X y is defined as (XT X)−XT y, where (XT X)− is a generalized inverse of X’X.

> bslash <- function(X, y) {

X <- qr(X)

qr.coef(X, y)

}

After this object is created it may be used in statements such as

> regcoeff <- bslash(Xmat, yvar)

￼and so on.

The classical R function lsfit() does this job quite well, and more. It in turn uses the functions qr() and qr.coef() in the slightly counterintuitive way above to do this part of the calculation. Hence there is probably some value in having just this part isolated in a simple to use function if it is going to be in frequent use. If so, we may wish to make it a matrix binary operator for even more convenient use.

### Defining New Binary Operators

Had we given the bslash() function a different name, namely one of the form %anything%

it could have been used as a binary operator in expressions rather than in function form. Suppose, for example, we choose ! for the internal character. The function definition would then start as

> "%!%" <- function(X, y) { ... }

(Note the use of quote marks.) The function could then be used as X %!% y. (The backslash

symbol itself is not a convenient choice as it presents special problems in this context.)

The matrix multiplication operator, %\*%, and the outer product matrix operator %o% are other examples of binary operators defined in this way.

### Named Arguments and Defaults

As first noted in Section 2.3 [Generating regular sequences], page 8, if arguments to called functions are given in the “name=object” form, they may be given in any order. Furthermore the argument sequence may begin in the unnamed, positional form, and specify named arguments after the positional arguments.

Thus if there is a function fun1 defined by

> fun1 <- function(data, data.frame, graph, limit) {

[function body omitted]

}

then the function may be invoked in several ways, for example

> ans <- fun1(d, df, TRUE, 20)

> ans <- fun1(d, df, graph=TRUE, limit=20)

> ans <- fun1(data=d, limit=20, graph=TRUE, data.frame=df)

are all equivalent.

In many cases arguments can be given commonly appropriate default values, in which case they may be omitted altogether from the call when the defaults are appropriate. For example, if fun1 were defined as

> fun1 <- function(data, data.frame, graph=TRUE, limit=20) { ... }

it could be called as

> ans <- fun1(d, df)

which is now equivalent to the three cases above, or as

> ans <- fun1(d, df, limit=10)

which changes one of the defaults.

It is important to note that defaults may be arbitrary expressions, even involving other arguments to the same function; they are not restricted to be constants as in our simple example here.

### The ‘...’ argument

Another frequent requirement is to allow one function to pass on argument settings to another. For example many graphics functions use the function par() and functions like plot() allow the user to pass on graphical parameters to par() to control the graphical output. This can be done by including an extra argument, literally ‘...’, of the function, which may then be passed on. An outline example is given below.

fun1 <- function(data, data.frame, graph=TRUE, limit=20, ...) {

[omitted statements]

if (graph)

par(pch="\*", ...)

[more omissions]

}

Less frequently, a function will need to refer to components of ‘...’. The expression list(...) evaluates all such arguments and returns them in a named list, while ..1, ..2, etc. evaluate them one at a time, with ‘..n’ returning the n’th unmatched argument.

1. MATLAB® Primer © COPYRIGHT 1984–2014 by The MathWorks, Inc. [↑](#footnote-ref-1)