Elements of Scientific Computing in Julia

March 5, 2015

Contents

Lecture 1

What is Scientific Computing?

Scientific computing is the process of coming up with algorithms and computer programs to solve quantitative and qualitative problems arising in areas such as the engineering and behavioral sciences, as well as biology and finance.

As a scientist of a specific area, one might develop a formula for making predictions. In general, these formulas will have several variables, so that an analytic solution might be difficult or even impossible to find. Scientists must then rely on computers to find a combination of variables that will produce a good approximation to the solution of a given problem. Scientific computing will draw heavily from two subjects: mathematics and computer science. We need the mathematics to come up with a well defined mathematical model to solve a particular science problem, but we will also need computer science to design robust, precise and well performing algorithms as well as recognize the correct tools to use, since the concern for optimal approximate solutions rely heavily on precision and performance.

Usually, scientific problems are concerned with approximate and quantitative results to continuous (or numerical) problems. This course will take a broader view of what is usually covered in most scientific computing courses. It also includes topics from statistical learning and data analysis, therefore at times we will also be concerned with qualitative solutions and discretely modeled problems. Some examples of scientific computing tools and problems we will be studying are:

1. Numerical Integration: algorithms for computing the numerical value of definite integrals, such as the trapezoidal and Simpson's rule. As you know the definite integral of a given function can be interpreted as the area under the curve. There is several reasons a scientist may need to take an integral: calculating force, calculating errors, sampling, etc. Not all functions are easy to integrate analytically, thus numerical integration can be used to find approximate results. We will explore simple numerical integration problems as an introduction to Julia.

2. Stencil Computation: algorithms for computing the numerical solution to partial differential equations. As an example we will learn about the heat equation, its discretization, and ways to solve it numerically. The heat equation is a partial differential equation describing the distribution of heat in a given region over time.

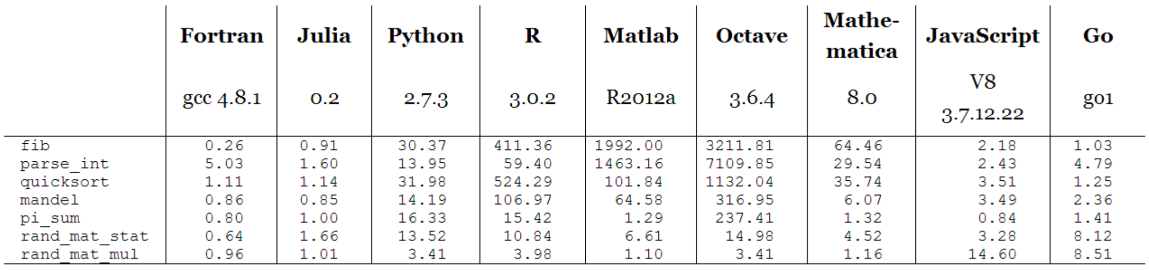
3. Numerical Linear Algebra: algorithms for matrix factorization and for solving systems of linear equations. We will focus on solving the linear least squares problem for linear regression analysis of data sets.

4. Numerical Optimization: gradient decent algorithm for finding local minimum of a function. We will go back to the problem of solving systems of linear equations via linear least squares and reformulate it as a quadratic minimization problem.

5. Logistic Regression: a qualitative form of regression analysis aiming to separate or classify a data set into qualitative (discrete) categories. We will focus on the binary case and work on classification algorithms for separating spam from ham.

6. And Many More..: other topics and examples that are relevant to scientific computing are recommender systems, sentiment analysis, solving non-linear systems of equations, parallel computing and high performance computing. As an overview course, we will have a chance to discuss many topics of these topics at a high level.

A computational scientist must be well versed with an efficient and easy to use programming language, so that he may constantly change his program in search of better results. In this course we will use Julia, a high performing and dynamic programming language. It's syntax is friendly and flexible for programming scientific computing algorithms. Julia's compiler is based on the LLVM (Low Level Virtual Machine) compiler infrastructure, so performance is not an issue. Here are some benchmarks for comparison:



	Fortran	Julia	Python	R	Matlab	Octave	Mathe-matica	JavaScript	Go
	gcc 4.8.1	0.2	2.7.3	3.0.2	R2012a	3.6.4	8.0	V8 3.7.12.22	go1
fib	0.26	0.91	30.37	411.36	1992.00	3211.81	64.46	2.18	1.03
parse_int	5.03	1.60	13.95	59.40	1463.16	7109.85	29.54	2.43	4.79
quicksort	1.11	1.14	31.98	524.29	101.84	1132.04	35.74	3.51	1.25
mandel	0.86	0.85	14.19	106.97	64.58	316.95	6.07	3.49	2.36
pi_sum	0.80	1.00	16.33	15.42	1.29	237.41	1.32	0.84	1.41
rand_mat_stat	0.64	1.66	13.52	10.84	6.61	14.98	4.52	3.28	8.12
rand_mat_mul	0.96	1.01	3.41	3.98	1.10	3.41	1.16	14.60	8.51

More information on Julia can be found at julialang.org

Finally, two other important aspects of scientific computing are presentation and collaboration. It is not enough for scientists to come up with formulas and long tables of coefficients obtained computationally. Scientists must make sense of it all and present their results in a clean, standard and modular format via the use of tools such as LaTeX (for typesetting documents) and Git (for revision control). Many other presentation and collaboration tools exist, but in this course we will cover the the aforementioned ones.

Installation and Tools Set-up

Our computational scientist's tool box will consist of the following:

1. *Mathematical tools*: through out the course I will be assigning required reading of freely available documents as well as optional non-free resources on the mathematical tools necessary for arriving at the solutions of many interesting scientific problems. There is no required textbook for this class, however I would like to recommend Michael Heath's *Scientific Computing: An Introductory Survey* as an appropriate textbook for this course if you wish to acquire one. These lecture notes, as well as the homework assignments found herein, are significantly inspired by Heath's textbook.

2. *Julia*: this is the programming language we will use in the classroom and for the homework assignments. You will need to install Julia on your machine. Please follow the instructions below.

3. *Git*: this is a revision (or version) control and source code management system focused on speed, originally created in 2005 by Linus Torvalds for Linux kernel development, now widely used for sharing source code and keeping track of different versions of computer programming projects. You will need to install Git on your machine. Please follow the instructions below.

4. *GitHub*: this is a web-based hosting service for computer programming projects using the Git RCS. We will be using GitHub as our class website, that is, there you will find our syllabus and homework descriptions. You will also submit your homework via Git on GitHub. First sign-up for a free public GitHub account at github.com with your *.edu* address, then send me an email with your user name so that I may provide you with a free private repository for use in this class.

5. LaTeX (pronounced lay-tech): this is a document markup language that uses the TeX typesetting program for formatting text output, it is widely used in academia and for presenting mathematical formulas on various websites. Its philosophy is that authors should focus on the contents of their writing rather than getting distracted on the organization and structure of their document (LaTeX does it all for you, although getting distracted is possible). Installation instructions can be found below.

Installing Julia:

1. If you have a Mac or a Windows machine, download the Anaconda package and run its installer. If you have a Linux machine, then install IPython and related scientific-Python packages (SciPy and Matplotlib).

   Important! on Windows, the Anaconda installer will give you the following options: "Add Anaconda to the System Path" and also "Register Anaconda" as default Python version of the system. Make sure those boxes are selected.

2. Download Julia version 0.2 and run the installer. Do not download version 0.1. Then run the Julia application (double-click on it); a window with a `julia>` prompt will appear. At the prompt, type:

   ```
   Pkg.add("IJulia")
   Pkg.add("PyPlot")
   ```

3. We will get started with IJulia in the section below.

Installing Git:

Please follow the instructions found at the Set Up Git page of GitHub, for installing and setting up Git on your operating system. To get started using Git, you may want to either install the easy-to-use GUI provided by GitHub or go through this short and simple tutorial.

Installing TeX:

1. If you have a Mac, you will need to install MacTeX. If you have Windows, you will have to install MikTeX. If you have Linux, simply `sudo apt-get install texlive`

2. You will also need a suitable editor for typesetting your documents. You are welcome to use your preferred editor, however I would like to recommend Texmaker as a fairly easy editor to get started with LaTeX.

3. You may want to go through this Getting Started with LaTeX guide, before starting to work on your first homework. Or you can always consult a wiki to find the command that does what you need, when you need it.

Approximations, Errors and Getting Started with IJulia

Scientific computing is all about approximations, and approximations produce errors! The source of inexactness can come from any stage in the process of formulating and solving a scientific computing problem. We may be using a simplified model (since there are many complexities in real world problems that are difficult to account for); our measurement equipment have finite precision or we were not able to gather that much data anyway; we may have to discretize a continuous problem in order to solve it; truncation and rounding will happen due to computer precision. Therefore the accuracy of our final result will depend on a combination of all of these errors and approximations. We must study their effect on our results in order to better understand our results. Such study is called *error analysis*.

Example 1 Consider computing the surface area of the Earth by using the formula: $A = 4\pi r^2$.

- This formula gives the surface area of a perfect sphere, which is not necessarily the true shape of the Earth;
- The radius of the Earth is based on a combination of measurements taken by equipment of finite precision;
- π is a transcendental infinite quantity, but it will have to be truncated at some point;
- The final result will be rounded by our computers depending on their precision.

The stereotypical scientific computing problem can be view as computing the value of a function $f : \mathbf{R} \to \mathbf{R}$ (actually most scientific computing problems will be multidimensional, but let us consider one dimension for now). If we denote the true value of the input data by x, the true result by $f(x)$, the inexact input by $\hat{x}$ and the approximated function by $\hat{f}$, then:

Total Error = $\hat{f}(\hat{x}) - f(x) = (\hat{f}(\hat{x}) - f(\hat{x})) + (f(\hat{x}) - f(x))$ = computational error + propagated data error.

That is, the difference between using the approximate function and the exact function on the same input is a *computational error*. While the difference between using the exact function on inexact input versus exact input is called a pure *data error*.

Computational errors will be either truncation or rounding errors. Data errors may have more to do with other biases or limitations in collecting the data. *Truncation error* is the difference between the result we get after truncating an infinite series (or other infinite quantities) and the result that would be produced if we were to use exact arithmetic. *Rounding error* is the difference between the result we get due to using finite precision computers and the result that would be produced if we were to use exact arithmetic. Thus, *computational error* is the sum of these two errors.

We can represent errors from numerical computations in either *absolute* or *relative* form:

Absolute error = approximate value - true value
Relative error = (approximate value - true value)/true value

Note that the relative error can be seen as a percentage of the true value, if we multiply it by 100.

Besides analyzing the error of a approximate solution, it is also wise to understand the conditioning of our model. A problem is *well-conditioned* if a relative change in the input date causes a similar relative change in the solution. While an *ill-conditioned* problem is one where a relative change in the input data causes a much larger relative change in the solution. Given these definitions, we can calculate the condition number of a problems as follows (we denote it by $\mathcal{K}$):

$$\mathcal{K} = \frac{|\text{relative change in solution}|}{|\text{relative change in input data}|} = \frac{|(f(\hat{x}) - f(x))/f(x)|}{|(\hat{x} - x)/x|}$$

Example 2 Consider the problem of computing the values of $y = \cos(x)$ near $\pi/2$. Let $x = \pi/2$ and let h be a small perturbation to x. Then we have:

$$\textit{Absolute error} = \cos(x+h) - \cos(x) \approx -h\sin(x) \approx -h$$

This follows by the definition of the derivative: $f'(x) = \lim_{h\to 0} \frac{f(x+h) - f(x)}{h}$. Recall that the derivative of $\cos(x)$ is $-\sin(x)$.

$$\textit{Relative error} = \frac{\cos(x+h) - \cos(x)}{\cos(x)} \approx -h\frac{\sin(x)}{\cos(x)} = -h\tan(x) \approx \infty$$

Note that a small change in the input data for $\cos(x)$ near $\pi/2$, causes a large relative change in the output, regardless of how we computed it.

Now let us explore errors and learn a bit of Julia using IJulia! Once you have followed the installation steps above, open up the command line and type:

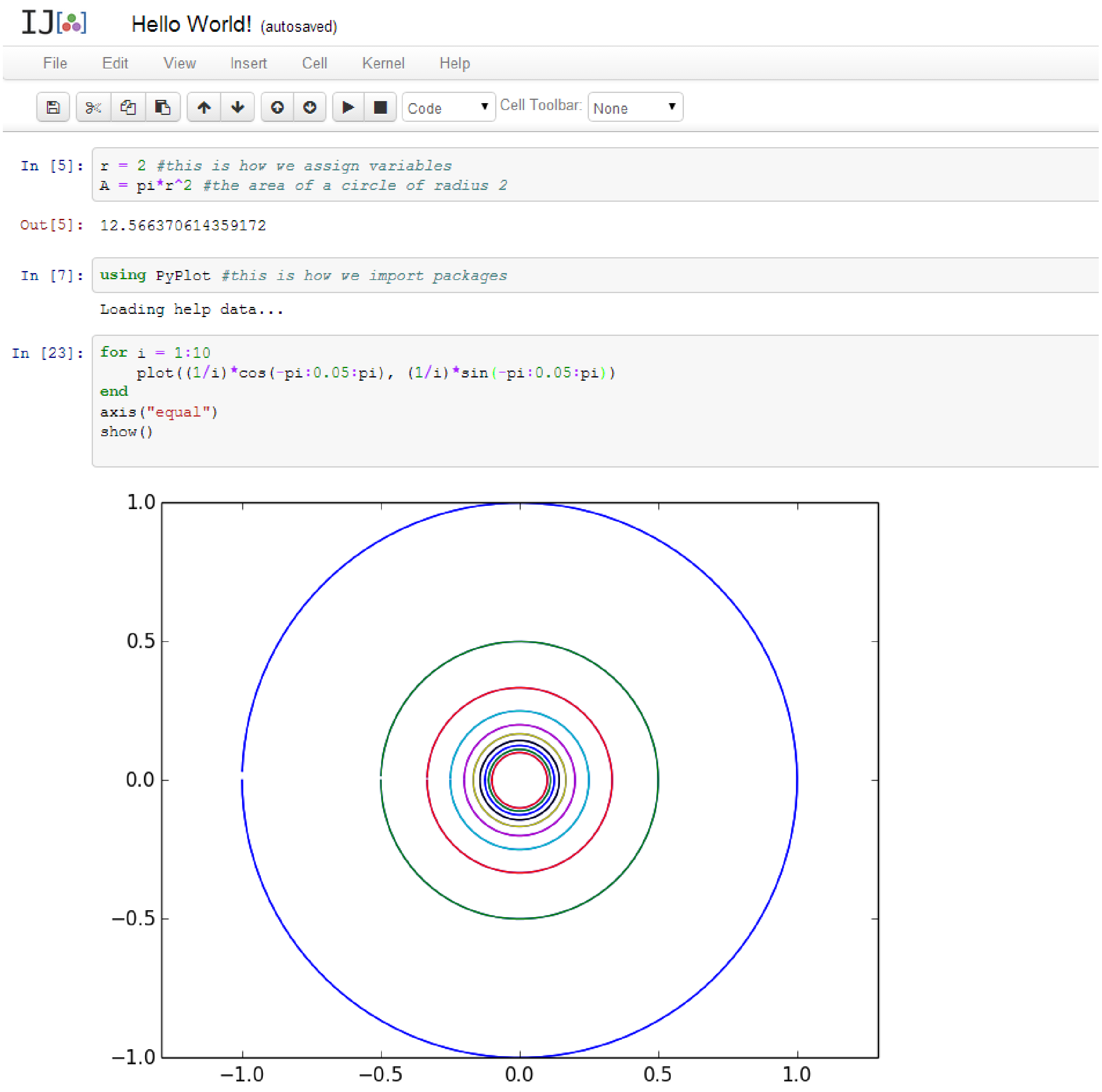
```
ipython notebook --profile julia
```

Important: On Windows, you must run the above command from the `bin` subdirectory of the Julia directory that you downloaded/unpacked.

A dashboard window will open in your web browser. Click on the "New Notebook" button to start a new notebook. A notebook will combine code, computed results, formatted text, and images; for example, you might use one notebook for each problem set. You can click the "Untitled0" at the top to change the name, e.g. to "Homework One". You can enter Julia code at the `In[ ]` prompt, and hit **shift-return** to execute it and see the results. If you hit **return** without the **shift** key, it will add additional lines to a single input cell.

Example 3 On the next page, you can see what IJulia will look like in your browser. We can assign variables like `r` and `A` (**note:** that no type is needed as Julia is dynamically typed). We can use the package `PyPlot` we installed earlier and plot some concentric circles.

Let us now work on a more relevant example...



```
In [5]:  r = 2 #this is how we assign variables
         A = pi*r^2 #the area of a circle of radius 2

Out[5]:  12.566370614359172

In [7]:  using PyPlot #this is how we import packages

         Loading help data...

In [23]: for i = 1:10
             plot((1/i)*cos(-pi:0.05:pi), (1/i)*sin(-pi:0.05:pi))
         end
         axis("equal")
         show()
```

Example 4 Let's write a Julia program to compute the mathematical constant e, the base of natural logarithms, from the definition:

$$e = \lim_{n\to\infty}(1+1/n)^n$$

We will compute $(1+1/n)^n$ for $n = 10^k$, $k = 1,2,...,20$. Our goal will be to determine the error in the successive approximations we get by comparing them with the value of the built in constant e. A question for us to think about: do you think the error always decreases as n increases?

If we enter the following code into IJulia:

```julia
for k = 1:20
    a = (1+1/(10^k))^(10^k)
    err = a - e
    @printf("For k = %d, we get the approximation: %1.12f and the error %1.12f \n",
    k, a, err)
end
```

Then we will get the following result:

```
For k = 1,  we get the approximation: 2.593742460100 and the error -0.124539368359
For k = 2,  we get the approximation: 2.704813829422 and the error -0.013467999038
For k = 3,  we get the approximation: 2.716923932236 and the error -0.001357896223
For k = 4,  we get the approximation: 2.718145926825 and the error -0.000135901634
For k = 5,  we get the approximation: 2.718268237192 and the error -0.000013591267
For k = 6,  we get the approximation: 2.718280469096 and the error -0.000001359363
For k = 7,  we get the approximation: 2.718281694132 and the error -0.000000134327
For k = 8,  we get the approximation: 2.718281798347 and the error -0.000000030112
For k = 9,  we get the approximation: 2.718282052012 and the error  0.000000223553
For k = 10, we get the approximation: 2.718282053235 and the error  0.000000224776
For k = 11, we get the approximation: 2.718282053357 and the error  0.000000224898
For k = 12, we get the approximation: 2.718523496037 and the error  0.000241667578
For k = 13, we get the approximation: 2.716110034087 and the error -0.002171794372
For k = 14, we get the approximation: 2.716110034087 and the error -0.002171794372
For k = 15, we get the approximation: 3.035035206549 and the error  0.316753378090
For k = 16, we get the approximation: 1.000000000000 and the error -1.718281828459
For k = 17, we get the approximation: 1.000000000000 and the error -1.718281828459
For k = 18, we get the approximation: 1.000000000000 and the error -1.718281828459
For k = 19, we get the approximation: 1.000000000000 and the error -1.718281828459
For k = 20, we get the approximation: 1.000000000000 and the error -1.718281828459
```

Note that we get the smallest error at $k = 8$, and then the error starts to increase again. When we reach $k = 16$ the error is pretty large, and after that the approximation is `1.000000000000`, which is quite wrong! A good discussion or personal research topic is to figure out why this is happening when it is happening.

Homework 1 (C)

The purpose of homework 1 is for you to become better acquainted with the software tools you will be using this quarter as well as check your understanding of what is scientific computing and how do errors of various types arise in this field.

Exercises:

1. (5 pts) Read the following paper and write a 1-2 pages summary using LaTeX formatting. Include at least one (numbered or bullet) list, a boldfaced/italicized/underlined sentence, a mathematical formula and a snippet of source code.

2. (5 pts) Solve the following problem using IJulia: If an amount a is invested at interest rate r compounded n times per year, the the final value f at the end of one year is given by $f = a(1+r/n)^n$. This is the familiar formula for compound interest (so if we compound annually, then $n = 1$). Typically, compounding is done quarterly ($n = 4$) or daily ($n = 365$). Note that the more we compound the greater the final amount at the end of the year, since more interest is paid on top of previous interest. But how much difference does this frequency actually make?

   Write a program that implements the compound interest formula. Test your program using an initial investment of $a = 100$, and interest rate of 5 percent (i.e. $r = 0.05$), and the following values for n: 1, 4, 365, 10,000, and 20,000. Implement the compound interest formula in two different ways:

   a) using a loop that repeatedly multiplies a by $(1+r/n)$ for a total of n times; and
   b) using built-in exponentiation and logarithms (note that in this case the compound interest formula can be re-written as such: $f = ae^{n\ln(1+r/n)}$).

   Compare your results from both parts a and b and write a few comments about the comparison in a comment block within your source code.

3. (5 pts) Solve the following exercises and submit your answers in a pdf file using any formatting (+2 points for using LaTeX formatting):

   a) What are the approximate absolute and relative errors in approximating π by each of the following quantities?
      i) 3
      ii) 3.14
      iii) 22/7
   b) Consider the problem of evaluating the function $\sin(x)$, in particular, the propagated data error, i.e., the error in the function value due to a perturbation h in the argument x.
      i) Estimate the absolute error in evaluating $\sin(x)$.
      ii) Estimate the relative error in evaluating $\sin(x)$.
      iii) Estimate the condition number for this problem.
      iv) For what values of the argument x is this problem highly sensitive?

Note: This homework is due one week after it has been assigned and it should be turned in via your private GitHub repository for this class.

Lecture 2

Programming with Julia

This lecture is meant as a crash course in Julia, a scientific computing language developed at MIT. Julia is a high-level, high-performance, dynamic programming language. This means that you can spend more time worrying about your scientific computing problems and models, rather than providing detailed computer instructions in your code. Some of the benefits of using Julia are:

- Julia is free and open source;
- Unlike MATLAB, de-vectorized code is fast;
- Julia is well equipped for parallel computing;
- We can call C functions directly;
- Power type system - support for arbitrary precision.

If you want to learn more details about Julia's history and features, I highly recommend reading this paper: Julia: A Fast Dynamic Language for Technical Computing.

Now on to the crash-course! (**Note:** This is a summary/selective copy-paste of the Learn X=Julia in Y Minutes page. For a more detail tutorial, you may find The Julia Manual useful).

```
# Single line comments start with a hash.

####################################################
## 1. Primitive Datatypes and Operators
####################################################

# Everything in Julia is an expression.

# There are several basic types of numbers.
3 #=> 3 (Int64)
3.2 #=> 3.2 (Float64)
2 + 1im #=> 2 + 1im (Complex{Int64})
2//3 #=> 2//3 (Rational{Int64})

# All of the normal infix operators are available.
1 + 1 #=> 2
8 - 1 #=> 7
10 * 2 #=> 20
35 / 5 #=> 7.0
5 / 2 #=> 2.5 # dividing an Int by an Int always results in a Float
div(5, 2) #=> 2 # for a truncated result, use div
5 \ 35 #=> 7.0
2 ^ 2 #=> 4 # power, not bitwise xor
12 % 10 #=> 2

# Enforce precedence with parentheses
(1 + 3) * 2 #=> 8
```

```
# Boolean values are primitives
true
false

# Boolean operators
!true #=> false
!false #=> true
1 == 1 #=> true
2 == 1 #=> false
1 != 1 #=> false
2 != 1 #=> true
1 < 10 #=> true
1 > 10 #=> false
2 <= 2 #=> true
2 >= 2 #=> true
# Comparisons can be chained
1 < 2 < 3 #=> true
2 < 3 < 2 #=> false

# Strings are created with "
"This is a string."

# Character literals are written with ’
’a’

# A string can be indexed like an array of characters
"This is a string"[1] #=> ’T’ # Julia indexes from 1
# However, this is will not work well for UTF8 strings,
# so iterating over strings is recommended (map, for loops, etc).

# $ can be used for string interpolation:
"2 + 2 = $(2 + 2)" #=> "2 + 2 = 4"
# You can put any Julia expression inside the parenthesis.

# Another way to format strings is the printf macro.
@printf "%d is less than %f" 4.5 5.3 # 5 is less than 5.300000

# Printing is easy
println("I’m Julia. Nice to meet you!")


####################################################
## 2. Variables and Collections
####################################################

# Variable names start with a letter.
# After that, you can use letters, digits, underscores, and exclamation points.
SomeOtherVar123! = 6 #=> 6

# You don’t declare variables before assigning to them.
some_var = 5 #=> 5
some_var #=> 5
```

```
# Accessing a previously unassigned variable is an error
try
    some_other_var #=> ERROR: some_other_var not defined
catch e
    println(e)
end

# Arrays store a sequence of values indexed by integers 1 through n:
a = Int64[] #=> 0-element Int64 Array

# Note that arrays must be given a type, and all elements in the array
must be of that type.

# 1-dimensional array literals can be written with comma-separated values.
b = [4, 5, 6] #=> 3-element Int64 Array: [4, 5, 6]
b[1] #=> 4
b[end] #=> 6

# 2-dimentional arrays use space-separated values and semicolon-separated rows.
matrix = [1 2; 3 4] #=> 2x2 Int64 Array: [1 2; 3 4]

# Add stuff to the end of an array with push! and append!
push!(a,1)     #=> [1]
push!(a,2)     #=> [1,2]
push!(a,4)     #=> [1,2,4]
push!(a,3)     #=> [1,2,4,3]
append!(a,b) #=> [1,2,4,3,4,5,6]

# Remove from the end with pop
pop!(b)        #=> 6 and b is now [4,5]

# Let's put it back
push!(b,6)   # b is now [4,5,6] again.

a[1] #=> 1 # remember that Julia indexes from 1, not 0!

# end is a shorthand for the last index. It can be used in any
# indexing expression
a[end] #=> 6

# we also have shift and unshift
shift!(a) #=> 1 and a is now [2,4,3,4,5,6]
unshift!(a,7) #=> [7,2,4,3,4,5,6]

# Function names that end in exclamations points indicate that they modify
# their argument.
arr = [5,4,6] #=> 3-element Int64 Array: [5,4,6]
sort(arr) #=> [4,5,6]; arr is still [5,4,6]
sort!(arr) #=> [4,5,6]; arr is now [4,5,6]

# You can initialize arrays from ranges
a = [1:5] #=> 5-element Int64 Array: [1,2,3,4,5]
```

```
# You can look at ranges with slice syntax.
a[1:3] #=> [1, 2, 3]
a[2:] #=> [2, 3, 4, 5]
a[2:end] #=> [2, 3, 4, 5]

# Remove elements from an array by index with splice!
arr = [3,4,5]
splice!(arr,2) #=> 4 ; arr is now [3,5]

# Check for existence in an array with in
in(1, a) #=> true

# Examine the length with length
length(a) #=> 8

# Tuples are immutable.
tup = (1, 2, 3) #=> (1,2,3) # an (Int64,Int64,Int64) tuple.
tup[1] #=> 1
try:
    tup[1] = 3 #=> ERROR: no method setindex!((Int64,Int64,Int64),Int64,Int64)
catch e
    println(e)
end

# Many array functions also work on tuples
length(tup) #=> 3
tup[1:2] #=> (1,2)
in(2, tup) #=> true

# You can unpack tuples into variables
a, b, c = (1, 2, 3) #=> (1,2,3)  # a is now 1, b is now 2 and c is now 3

# Tuples are created even if you leave out the parentheses
d, e, f = 4, 5, 6 #=> (4,5,6)

# Look how easy it is to swap two values
e, d = d, e  #=> (5,4) # d is now 5 and e is now 4

# Dictionaries store mappings
empty_dict = Dict() #=> Dict{Any,Any}()

# You can create a dictionary using a literal
filled_dict = ["one"=> 1, "two"=> 2, "three"=> 3]
# => Dict{ASCIIString,Int64}

# Look up values with []
filled_dict["one"] #=> 1

# Get all keys
keys(filled_dict)
#=> KeyIterator{Dict{ASCIIString,Int64}}(["three"=>3,"one"=>1,"two"=>2])
# Note - dictionary keys are not sorted or in the order you inserted them.
# Get all values
values(filled_dict)
#=> ValueIterator{Dict{ASCIIString,Int64}}(["three"=>3,"one"=>1,"two"=>2])
# Note - Same as above regarding key ordering.

# Check for existence of keys in a dictionary with in, haskey
in(("one", 1), filled_dict) #=> true
in(("two", 3), filled_dict) #=> false
haskey(filled_dict, "one") #=> true
haskey(filled_dict, 1) #=> false

# Trying to look up a non-existant key will raise an error
try
    filled_dict["four"] #=> ERROR: key not found: four in getindex at dict.jl:489
catch e
    println(e)
end

# Use the get method to avoid that error by providing a default value
# get(dictionary,key,default_value)
get(filled_dict,"one",4) #=> 1
get(filled_dict,"four",4) #=> 4

####################################################
## 3. Control Flow
####################################################

# Let’s make a variable
some_var = 5

# Here is an if statement. Indentation is not meaningful in Julia, but
it is good for readability.

if some_var > 10
    println("some_var is totally bigger than 10.")
elseif some_var < 10    # This elseif clause is optional.
    println("some_var is smaller than 10.")
else                    # The else clause is optional too.
    println("some_var is indeed 10.")
end
#=> prints "some var is smaller than 10"


# For loops iterate over iterables.
# Iterable types include Range, Array, Set, Dict, and String.
for animal=["dog", "cat", "mouse"]
    println("$animal is a mammal")
    # You can use $ to interpolate variables or expression into strings
end
# prints:
#    dog is a mammal
#    cat is a mammal
#    mouse is a mammal
# You can use 'in' instead of '='.
for animal in ["dog", "cat", "mouse"]
    println("$animal is a mammal")
end
# prints:
#    dog is a mammal
#    cat is a mammal
#    mouse is a mammal

for a in ["dog"=>"mammal","cat"=>"mammal","mouse"=>"mammal"]
    println("$(a[1]) is a $(a[2])")
end
# prints:
#    dog is a mammal
#    cat is a mammal
#    mouse is a mammal

for (k,v) in ["dog"=>"mammal","cat"=>"mammal","mouse"=>"mammal"]
    println("$k is a $v")
end
# prints:
#    dog is a mammal
#    cat is a mammal
#    mouse is a mammal

# While loops loop while a condition is true
x = 0
while x < 4
    println(x)
    x += 1  # Shorthand for x = x + 1
end
# prints:
#   0
#   1
#   2
#   3

# Handle exceptions with a try/catch block
try
   error("help")
catch e
   println("caught it $e")
end
#=> caught it ErrorException("help")


####################################################
## 4. Functions
####################################################

# The keyword 'function' creates new functions
#function name(arglist)
```

```
#  body...
#end
function add(x, y)
    println("x is $x and y is $y")

    # Functions return the value of their last statement
    x + y
end

add(5, 6) #=> 11 after printing out "x is 5 and y is 6"

# You can define functions that take a variable number of
# positional arguments
function varargs(args...)
    return args
    # use the keyword return to return anywhere in the function
end
#=> varargs (generic function with 1 method)

varargs(1,2,3) #=> (1,2,3)

# The ... is called a splat.
# We just used it in a function definition.
# It can also be used in a fuction call,
# where it will splat an Array or Tuple's contents into the argument list.
Set([1,2,3])    #=> Set{Array{Int64,1}}([1,2,3]) # produces a Set of Arrays
Set([1,2,3]...) #=> Set{Int64}(1,2,3) # this is equivalent to Set(1,2,3)

x = (1,2,3)     #=> (1,2,3)
Set(x)          #=> Set{(Int64,Int64,Int64)}((1,2,3)) # a Set of Tuples
Set(x...)       #=> Set{Int64}(2,3,1)


# You can define functions with optional positional arguments
function defaults(a,b,x=5,y=6)
    return "$a $b and $x $y"
end

defaults('h','g') #=> "h g and 5 6"
defaults('h','g','j') #=> "h g and j 6"
defaults('h','g','j','k') #=> "h g and j k"
try
    defaults('h') #=> ERROR: no method defaults(Char,)
    defaults() #=> ERROR: no methods defaults()
catch e
    println(e)
end

# You can define functions that take keyword arguments
function keyword_args(;k1=4,name2="hello") # note the ;
    return ["k1"=>k1,"name2"=>name2]
end
```

```
keyword_args(name2="ness") #=> ["name2"=>"ness","k1"=>4]
keyword_args(k1="mine") #=> ["k1"=>"mine","name2"=>"hello"]
keyword_args() #=> ["name2"=>"hello","k1"=>4]

# You can combine all kinds of arguments in the same function
function all_the_args(normal_arg, optional_positional_arg=2; keyword_arg="foo")
    println("normal arg: $normal_arg")
    println("optional arg: $optional_positional_arg")
    println("keyword arg: $keyword_arg")
end

all_the_args(1, 3, keyword_arg=4)
# prints:
#   normal arg: 1
#   optional arg: 3
#   keyword arg: 4

# Julia has first class functions
function create_adder(x)
    adder = function (y)
        return x + y
    end
    return adder
end

# This is "stabby lambda syntax" for creating anonymous functions
(x -> x > 2)(3) #=> true

# This function is identical to create_adder implementation above.
function create_adder(x)
    y -> x + y
end

# You can also name the internal function, if you want
function create_adder(x)
    function adder(y)
        x + y
    end
    adder
end

add_10 = create_adder(10)
add_10(3) #=> 13


# There are built-in higher order functions
map(add_10, [1,2,3]) #=> [11, 12, 13]
filter(x -> x > 5, [3, 4, 5, 6, 7]) #=> [6, 7]

# We can use list comprehensions for nicer maps
[add_10(i) for i=[1, 2, 3]] #=> [11, 12, 13]
[add_10(i) for i in [1, 2, 3]] #=> [11, 12, 13]
```

Hopefully, the brief tutorial above will give you most of the necessary Julia syntax knowledge for this quarter. One thing worth emphasizing is Julia's capabilities for arbitrary precision arithmetic. Recall the example we worked on in Lecture 1:

Write a Julia program to compute the mathematical constant e, the base of natural logarithms, from the definition:

$$e = \lim_{n\to\infty}(1+1/n)^n$$

Compute $(1+1/n)^n$ for $n = 10^k$, $k = 1,2,\ldots,20$ and determine the error in the successive approximations by comparing them with the value of the built in constant e.

If we enter the following *modified* code into IJulia:

```julia
for k = 1:2:20
    a = (1+BigFloat(1/10^k))^BigFloat(10^k)
    err = a - e
    @printf("For k = %d, we get the approximation: %1.12f and the error %1.12f \n",
    k, a, err)
end
```

We get the following output:

```
For k = 1,  we get the approximation: 2.593742460100 and the error -0.124539368359045110601357
For k = 3,  we get the approximation: 2.716923932236 and the error -0.001357896223152721491140
For k = 5,  we get the approximation: 2.718268237174 and the error -0.000013591284555344964105
For k = 7,  we get the approximation: 2.718281692545 and the error -0.000000135914079087169091
For k = 9,  we get the approximation: 2.718281827100 and the error -0.000000001359140743684725
For k = 11, we get the approximation: 2.718281828445 and the error -0.000000000013591573606460
For k = 13, we get the approximation: 2.718281828459 and the error -0.000000000000135831527022
For k = 15, we get the approximation: 2.718281828459 and the error -0.000000000000001147915738
For k = 17, we get the approximation: 2.718281828459 and the error  0.000000000000000180881062
For k = 19, we get the approximation: 2.718281828459 and the error -0.000000000000000253747066
```

Notice that I have chosen to loop from 1 to 20 with a step size of 2, this was just for the purpose of having a shorter output. You are welcome to inspect the output you get by using a step size of 1 as before. The main thing to notice in this modified version of our program is that we are using the `BigFloat` type. This means that we are now getting many more digits of accuracy than we were before and thus our results look pretty good and no longer seem to exhibit the imprecise behavior given larger values of k.

Another great use for the Julia language is parallel computing. This is a big idea in scientific computing, which we may touch on this quarter, however a course that focuses primarily on high-performance and parallel computing is advised for those who wish more than just a honorable mention.

Numerical Integration: Concepts

In your Integral Calculus class, you probably learned many different ways for analytically and algebraically solving for the definite integral of various different functions. That is, you were able to apply the Fundamental Theorem of Calculus to definite integrals and with some algebra you got an exact answer. However, you may have come across interesting integrable functions whose antiderivatives were difficult or even impossible to write, such as the famous probability density function for the normal distribution: $f(x) = \frac{1}{\sqrt{2\pi}} e^{-x^2/2}$. In this lecture we will look at quadrature methods for solving one dimensional definite integrals numerically.

Quadrature Methods

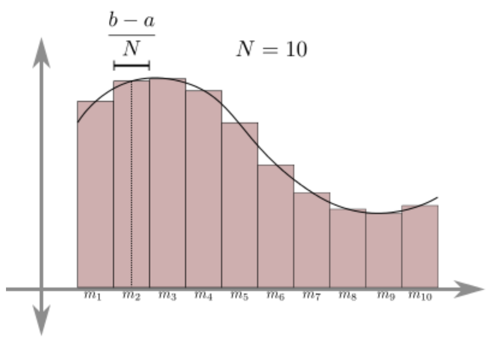
Quadrature means to calculate the area, and as you may recall, the definite integral of a function can be interpreted as the area under the graph of the function within a particular interval. We can chop up the area of an unfamiliar shape into slices, and approximate the area of each slice with the area of a familiar shape (such as a rectangle or a trapezoid). If we add up the area of all these slices, we should get an approximation of the are under the given curve, and thus the approximate solution of the definite integral of the given function.

For all of the rules below consider the definite integral $\int_a^b f(x)dx$, that is, the area under the curve f in the closed interval $[a,b]$. We divide $[a,b]$ into n subintervals of equal width $\Delta x = \frac{b-a}{n}$.

1. *Midpoint Rule* - This rule obtains the area under the curve by subdividing the area into rectangles of equal width Δx and height $f(x_i^*)$, where x_i^* is the **midpoint** of the subinterval $[x_{i-1}, x_i]$.

   The area of each rectangle is then $\Delta x f(x_i^*)$, and therefore, we get that:

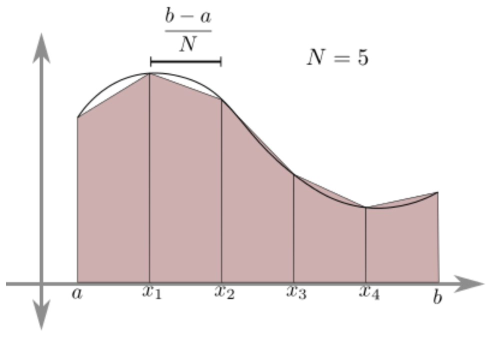
$$\int_a^b f(x)dx \approx \Delta x f(x_1^*) + \Delta x f(x_2^*) + \ldots + \Delta x f(x_n^*)$$



2. *Trapezoidal Rule* - This rule obtains the area under the curve by subdividing the area into trapezoids of equal trapezoidal height Δx, and base widths $f(x_{i-1})$ and $f(x_i)$ for each subinterval $[x_{i-1}, x_i]$.

   The area of each trapezoid is then $\frac{\Delta x}{2}(f(x_{i-1}) + f(x_i))$, and therefore, we get that:

$$\int_a^b f(x)dx \approx \frac{\Delta x}{2}(f(x_0) + f(x_1)) + \frac{\Delta x}{2}(f(x_1) + f(x_2)) + \ldots + \frac{\Delta x}{2}(f(x_{n-1}) + f(x_n))$$

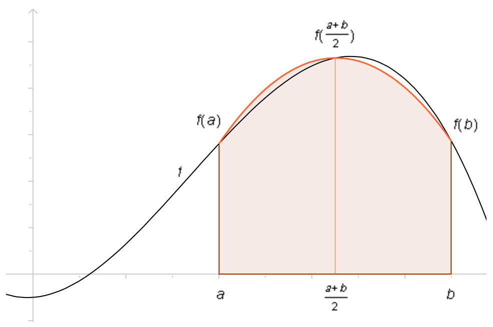


3. *Simpson's Rule* - This rule is an interpolative quadrature rule, since upon subdividing the curve into $n = 2k$ (that is, n must be even) slices, we interpolate the best fit quadratic curve for each slice. Using Lagrange polynomial interpolation on the endpoints a, b and the mid-point $m = (a+b)/2$, we find that the following quadratic function interpolates our curve on the interval $[a,b]$:

$$P(x) = f(a)\frac{(x-m)(x-b)}{(a-m)(a-b)} + f(m)\frac{(x-a)(x-b)}{(m-a)(m-b)} + f(b)\frac{(x-a)(x-m)}{(b-a)(b-m)}$$

With a bit of algebra and patience, it is easy to show that

$$\int_a^b P(x)dx = \frac{b-a}{6}\left[f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right]$$



If subdivide the interval $[a,b]$ into $n = 2k$ sub-intervals, then the integral of $f(x)$ is approximately

$$\frac{\Delta x}{3}[f(x_0) + 4f(x_1) + 2f(x_2) + \ldots + 2f(x_{n-2}) + 4f(x_{n-1}) + f(x_n)]$$

4. *Adaptive Quadrature* - this is a technique that can be used with any of the above rules and their respective error bounds (see next section). That is, we apply one of the above rules and if the error bound for a given interval is above a specified threshold, the algorithm then further sub-divides the interval. Thus, the term Δx is not fixed as before.

Sources for the images: http://calculus.seas.upenn.edu/?n=Main.DiscreteIntegration and https://en.wikipedia.org/wiki/Newton%E2%80%93Cotes_formulas

Numerical Integration Error Bounds

In the previous chapter we talked a great deal about errors and how important it is for computational scientists to understand the error caused by their mathematical models. We studied that error is equal to approximate value minus true value. Therefore, in other to determine the error of our numerical integration method, we can simply look at the difference between our result and the true result acquired via exact integration. That's easy! However, if we are able to get the true value, then why are we bothering with the approximation?

The cold truth is that for many problems we do not know the true value, but we can use mathematical theory to come up with upper bounds for our errors. *Error bounds* are quantities we can easily compute (unfortunately, we will not discuss their derivation in this course - here is an optional reading for the derivation of the trapezoidal rule error bound) and are set as ceilings for our errors, that is, the error obtained by a certain mathematical model is guaranteed to be less than its error bound. As you might have already guessed, an error bound depends on the mathematical model used (there is no universal error bound).

Below are the error bounds for the three methods of numerical integration we've discussed in this lecture. Let f be the function we are numerically integrating. Suppose that $\max\limits_{a\le x\le b}|f''(x)| = K$ and $\max\limits_{a\le x\le b}|f^{(4)}(x)| = M$ for $a \le x \le b$, then:

1. *The error bound for Midpoint Rule is*
$$\frac{K(b-a)^3}{24n^2}$$
2. *The error bound for Trapezoidal Rule is*
$$\frac{K(b-a)^3}{12n^2}$$
3. *The error bound for Simpson's Rule is*
$$\frac{M(b-a)^5}{180n^4}$$

An Example: The Error Function

$$\text{erf}(x) = \frac{2}{\sqrt{\pi}}\int_0^x e^{-t^2}\,dt$$

The error function (occurring in statistics to measure the behavior of a sample with respect to the population mean) given above is an example of a non-elementary function - that is, we cannot express erf(x) as a finite expression consisting of elementary operations (such as $+,-,\times,\div$) and components (such as logs, exponents, roots, and so on) . Thus it must be expressed as an integral, and if we desire to evaluate erf(x_0), then we must evaluate the integral that defines erf in the interval $[0,x_0]$.

However, this is not a trivial task, as the above integral does not have an analytic solution. Therefore we must rely on numerical approximation techniques such as the numerical integration methods we've discussed in this lecture.

Let's use IJulia to implement the erf function using the midpoint rule. Our function will take two inputs: the upper bound x of the interval of integration, and the number of sub-intervals n.

```
# erf's integrand
function integrand(t)
    e^(-t^2)
end

# erf function using the midpoint rule
# input: x - interval of integration upper bound,
#        n - number of sub-intervals
# output: [erf(x), midpoint rule error bound]

function middle_erf(x, n)

    # calculate delta_x
    delta_x = x / n

    result = 0
    # summing up the are of rectangles
    for i = 1:n
        x_mid = ((i-1)*delta_x + i*delta_x)/2
        result += delta_x*integrand(x_mid)
    end
    result *= 2 / sqrt(pi)

    # since the SECOND derivative of e^(-t^2) is (4t^2 - 2)*(e^(-t^2)),
    # and since it obtains a maximum at t = sqrt(3/2) on the interval
    # from 0 to x, when x greater than or equal to sqrt(3/2), OR just
    # at t = x, when x < sqrt(3/2), then our K in the error bound
    # formula is:

    if x >= sqrt(3/2)
      error_bound = (4/e^(3/2))*(x^3)/(24*n^2)
    else
      error_bound = abs((4*x^2 - 2)*(e^(-x^2)))*(x^3)/(24*n^2)
    end

    return ["result" => result, "error bound" => error_bound]
end
```

Homework 2 (P)

The purpose of homework 2 is for you to continue becoming acquainted with the tools we will be using this quarter, as well as put into practice the integration techniques we have learned here. Also, I will be checking if you are using git as a revision system (as opposed to a homework submission system). Make sure you have finished homework 1 before embarking on a new homework adventure!

Exercises:

1. (1 pt) In a `.tex` file (with appropriate mark-up) tell something about Julia I don't know.

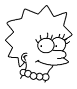
2. (8 pts) Implement a function in Julia called `simp_erf`, which takes as input the **upper bound of the integration interval** (that is, the "x" in the build-in erf(x)) and an **error bound** (that is, instead of giving the function the number of sub-intervals n as in the lecture example, we should be able to give the error bound instead) in order to calculate the value given by the error function (erf) using the Simpson's Rule. Compare your results with those of `middle_erf` and write a few sentences about it using LaTeX (use the same `.tex` file as in exercise 1).

3. (2 pts each) Use numerical integration to verify or refute each of the following conjectures. Also, provide a plot of each integrand:

   a) $\int_0^1 \sqrt{x^3} = 0.4$

   b) $\int_0^{10} \frac{50}{\pi(2500x^2+1)} dx = 0.5$

   c) $\int_{-9}^{100} \frac{1}{\sqrt{|x|}} dx = 26$

 > "This homework is so much fun!"

Lecture 3

In this lecture we will talk about numerical differentiation. In particular, we will explore the discretization and numerical solutions of partial differential equations (or PDE's). Unlike integration (where we deal with multiplication and addition), differentiation (where we deal with differences and division) is a sensitive problem, as small perturbations in the data can cause large change in the results.

If we only know the values of a function we wish to differentiate at a discrete set of points, then we could try to fit a smooth curve through those points and differentiate this curve. However we must be careful, since interpolating the appropriate curve is often difficult, specially if our data set is very noisy.

There are several methods for numerically solving PDE's and there is no single best method to solve all PDE's. That is, which method you use to solve a particular problem will depend on the problem and the available data. In this lecture we will explore the finite difference method for solving PDE's. This is one of many other methods I would encourage you to research on your own based on your interests.

Quick Introduction to PDE's

Partial differential equations are mathematical equations relating multivariable functions with their partial derivatives. PDE's are use to describe a variety of physical phenomena such as heat and fluid flow, but they also have been seen in finance, applied to such problems as option pricing.

The stereotypical PDE will involve derivatives with respect to time and at least one other dimension (length, depth, price of stock, etc). These PDE's describe phenomena occurring in different substances that have different boundaries, thus we must take into account boundary conditions when trying to solve PDE's. For concreteness, imagine that you are trying to understand the heat flow within a pool of water - it is not surprising that you might have to consider a different behavior in the heat flow by the walls of the pool.

Here are some famous PDE's (just a small sample) you might have heard about (I am listing the one spacial dimension equations here for simplicity, but all of these can be generalized to n spacial dimensions by simply adding the partial derivatives of the respective dimensions):

- $\frac{\partial^2 u}{\partial t^2} - c^2 \frac{\partial^2 u}{\partial x^2} = 0 \rightarrow$ the wave equation
- $\frac{\partial u}{\partial t} - \alpha \frac{\partial^2 u}{\partial x^2} = 0 \rightarrow$ the heat equation (time dependent)
- $\frac{\partial u}{\partial t} - c \frac{\partial u}{\partial x} - d \frac{\partial^2 u}{\partial x^2} = 0 \rightarrow$ the transport (or convection-diffusion) equation
- $\frac{\partial u}{\partial t} + u \frac{\partial u}{\partial x} = 0 \rightarrow$ inviscid Burger's equation
- $\hbar \frac{\partial u}{\partial t} + \frac{\partial^2 u}{\partial x^2} = 0 \rightarrow$ time-dependent Schrödinger's equation

An Example: The Steady State Heat Equation

As an example, let us study the two dimensional steady state heat equation:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0$$

where $u(x, y)$ is a multivariable function describing the temperature in a two dimensional space (e.g. such as an aluminum lamina) with a steady time independent heat distribution. Note that we may generalize this heat equation to model heat transfer in higher dimensional spaces by adding more partial derivative terms related to the other dimensions, however for sake of simplicity we will stay in flatland.

Recall from Calculus I that differentiable functions must be continuous, thus u is a continuous function here. However, as discussed in the first lecture, computers operate on discrete values and so we must first discretize the heat equation before we can use a computer to approximate its solution.

We discretize the dimensions x and y into points that are Δx and Δy distance apart, respectively. So that the integer tuple (i, j) corresponds to the point $(x,y) = (i\Delta x, j\Delta y)$ in continuous space. Note that since we are using discretized dimensions, we can only compute an approximation to u at any given point. Let us denote $U_{i,j}$ as the approximation to $u(i\Delta x, j\Delta y)$. In order to compute this approximate solution, we must discretize the heat equation given above as follows:

$$\frac{U_{i-1,j} - 2U_{i,j} + U_{i+1,j}}{\Delta x^2} + \frac{U_{i,j-1} - 2U_{i,j} + U_{i,j+1}}{\Delta y^2} = 0$$

This discretization is obtained by applying the principles behind the finite difference method, one of many methods for solving PDE's. This method works particularly well on smooth data sets. If we set the grid spacing to be equal in all spacial dimensions (that is, let $\Delta x = \Delta y = h$) and solve for $U_{i,j}$ above, we get the following update:

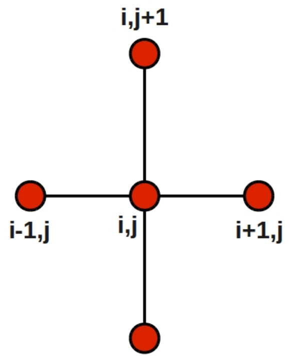
$$\boxed{U_{i,j} = \frac{1}{4}(U_{i-1,j} + U_{i+1,j} + U_{i,j-1} + U_{i,j+1})} \tag{1}$$

If h is sufficiently small, then equation (1) is a good finite difference scheme for finding the solution to the steady state heat equation. Note that this formula tells us that the temperature at a particular grid point is the average of the temperature of the grid points surrounding it, which makes sense since a discretize derivative is no more than an average rate of change over a discrete (hopefully sufficiently small) interval. We can see this formula as a special case of a more general finite difference where the grid spacing may not be equal or where there may be extra non-spacial dimensions, such as time.

One more detail we must address on discretizing continuous equations is the specification of boundary conditions. Even though u and U are conceptually continuous everywhere, computationally we can only simulate a grid for a finite set of points, finitely spaced. In this example we will assume that $u = 0$ at the bottom and left boundaries, and that $u = 1$ at the top and right boundaries, however feel free to experiment and change these values.

Stencil Computation

The finite difference approximation scheme above falls into the general category of stencil computation - that is, equation (1) above is a way to update our two dimensional array elements with a fixed pattern in an iterative way. This fixed pattern is called a stencil. In our case, we have a five point stencil (i.e. the temperature of a grid point is determined by the four points surrounding it each possible dimensional direction).

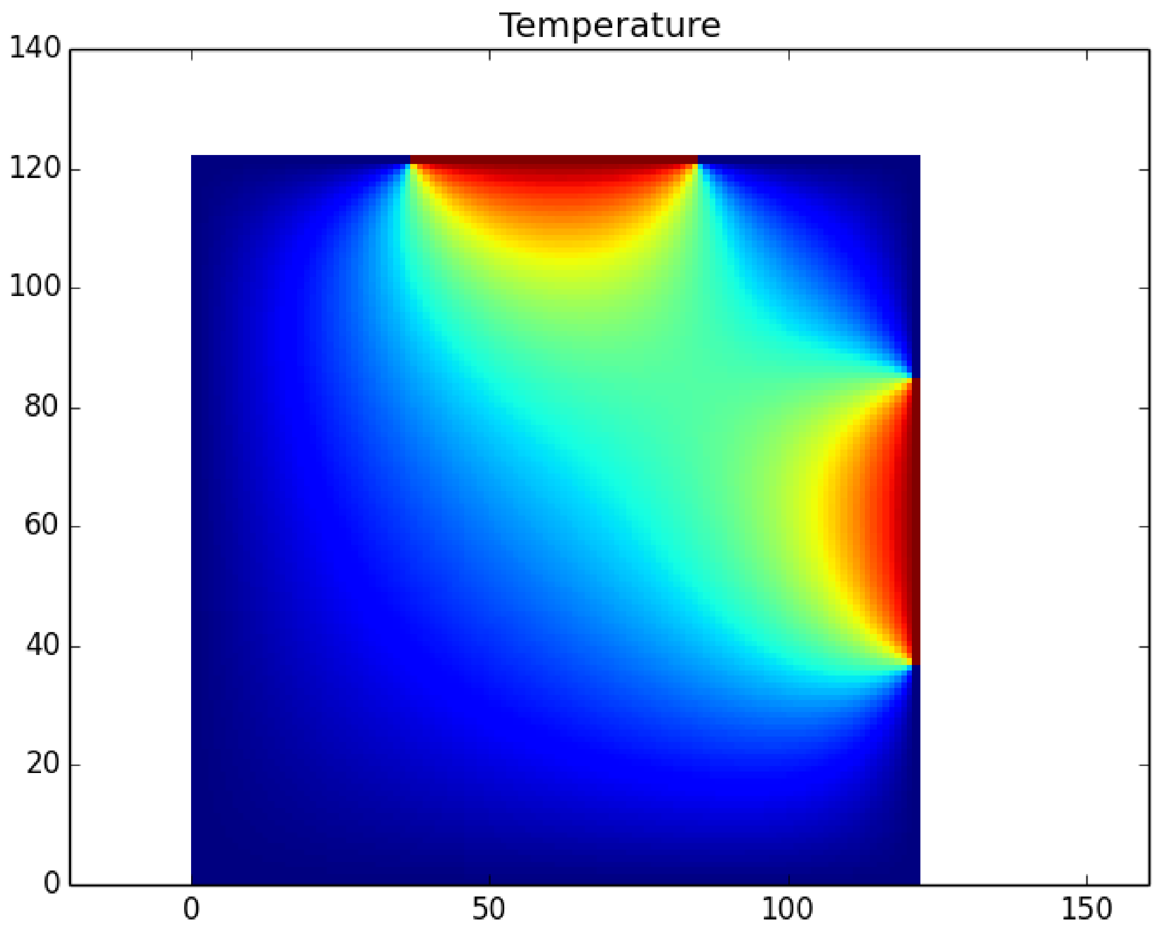


The iterative method resulting from our model so far is called the Jacobi method. However, please note that this is not a very good method for solving this problem, since if we "vectorize" our code then we will get a diagonally dominant matrix that is sparse (i.e. mostly filled with zeros). Therefore, other matrix-based techniques will be more efficient for solving problems modeled with the finite difference method as we will explore later in the course.

Below is a Julia function which implements the Jacobi iteration for the 2D steady state heat equation on a N by N grid. The update formula and boundary conditions are as per the previous section (with some minor changes to make the pictures more interesting).

```
#sample call jacobi_heat(120, 100000)

function jacobi_heat(N, maxiter)
    #grid spacing
    h = 1 / (N + 1)

    #tolerance
    tolerance = 0.1 / (N+1)^2

    #x ad y coordinates of grid points
    x = [0:h:N+1]
    y = [0:h:N+1]

    #initialize U
    U = zeros(N+2, N+2)

    #top and right boundaries should be initialized to 1
    #U[:, N+2] = ones(N+2)
    #U[N+2, :] = ones(N+2)'

    #making the boundary temps more interesting
    for k = 1:N+2
        if 0.3 < x[k] < 0.7
            U[k, N+2] = 1
        end
        if 0.3 < y[k] < 0.7
            U[N+2, k] = 1
        end
    end

    iter = 0
    dumax = tolerance

    while iter < maxiter && dumax >= tolerance
        Uold = U
        for j = 2:N+1
            for i = 2:N+1
                U[i,j] = (1/4)*(Uold[i-1,j] + Uold[i+1,j] +
                          Uold[i,j-1] + Uold[i,j+1])
                dumax = max(dumax, abs(U[i,j] - Uold[i,j]))
            end
        end
        iter += 1
    end
    return U
end
```

Below is the pseudocolor plot for the resulting matrix U.



Homework 3 (C)

There are two components to homework 3 - first there is a conceptual question meant to encourage you to consider historically important PDE's in more detail. The remaining questions are not directly related to PDEs but is instead about benchmarking and performance in Julia, and should both give you further practice with the language as well as a better understanding of its performance characteristics (one of its main advantages relative to other languages oriented towards scientific computing).

Exercises:

1. (3 pts each) Choose two of the famous PDE's (e.g. the wave equation, the time-dependent heat equation, etc) listed in the beginning of this lecture. Look up their history and application, and write at least one paragraph about each of them (though more is encouraged) describing what makes them unique and important. You should have at least one source for each PDE you discuss, and you should cite it in MLA format. Commit your work in a file named `hw3PDE.tex` using appropriate markup (you may find `\footnote{footnote text}` and `\endnote{endnote text}` to be useful for citing sources).

2. (2 pts each) In this exercise you will benchmark Julia performance using simple mathematical functions. Your goal is to calculate how many millions of evaluations per second your computer performs for each of the following functions:

   (a) An arithmetic operator (addition, subtraction, multiplication, or division)
   (b) A trigonometric function (sine, cosine, etc.)
   (c) A logarithm
   (d) A function you wrote for a previous homework (should take one or two numbers as input, and can return whatever you want as output)

   To do this you will want to use either the `@time` or `@elapsed` macros. The former prints the time taken to execute but still returns the value from the computation, while the latter directly returns the time elapsed in evaluation instead of the regular return value. The Julia manual details these and other related methods.

   To get you started, here is code that draws a million random numbers and then performs a million additions with them, reporting how long it takes for each step:

```
N = 1000000
println("rand:  ",  1 / @elapsed x = rand(N))
println("+:     ",  1 / @elapsed x + x)
```

   Note: # of millions operations per second = 1 / time elapsed. For instance, if it takes 0.5 seconds to evaluate, then the number of operations per second is $1/0.5 = 2$ million.

   At a minimum you should execute the benchmark for each operator 10 times, and report the final average. Note that while you could do this manually, it's far wiser to take advantage of the programming language and build a loop around the benchmarking portion, and then sum up and average the overall time. Make sure you report the average of a million operations ran ten times, and not of e.g. ten millions operations run once, as that may be different.

   Check in a Julia notebook named `benchmark.ipynb` that performs the above described benchmarking, as well as a text file `benchmark_output.txt` that shows your results (both the raw output of running your code, and if necessary further clarification on how many operations per second each operator performs on average).

3. (1 pt) Interpret your benchmark results - how did the different functions perform? Why do you think they performed/ranked as they did? Did anything about the results surprise you? Feel free to include your ideas in the `benchmark_output.txt` file from exercise 2.

Lecture 4

Matrix Algebra

Notes coming soon...

Matrix Operations in Julia

Notes coming soon...

Homework 4 (P)

The purpose of this homework is to have you practice basic linear algebra concepts essential to the understanding of numerical algorithms that require matrix manipulation (which will be introduced later in the quarter). As this homework is to be completed using Julia primarily, it will also serve as practice for writing vectorized code. Complete all of the exercises below in an IJulia notebook using git to add to and revise your work.

Exercises:

1. (5 pts)

   (a) Consider the following system of linear equations:

   $$\begin{aligned} 2x_1 + x_2 + 5x_3 &= -1 \\ x_1 + 6x_3 &= 2 \\ -6x_1 + 2x_2 + 4x_3 &= 3 \end{aligned}$$

   On paper, convert this system of equations into a matrix equation of the form **Ax = b** (you do not have to turn this in). In your IJulia notebook, make a comment describing the dimensions of your matrix and vectors. Also explain what it means for us to solve for the vector **x**.

   (b) Solve for **x**.

   (c) Using Julia, what is **Ax - b**? Is that what you expected? Explain what you expected and why Julia didn't agree.

2. (5 pts)

   (a) Enter the following matrices in Julia:
   $A = \begin{bmatrix} 1 & 2 & 0 \\ 2 & 1 & 2 \\ 0 & 2 & 1 \end{bmatrix}, B = \begin{bmatrix} 3 & 0 & 3 \\ 1 & 5 & 1 \\ 1 & 1 & 3 \end{bmatrix}, x = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$
   And compute the following expression: $\mathbf{C} = (2\mathbf{A}^2\mathbf{B} + 3\mathbf{A}^T)^2$.

   (b) Now compute the following expressions: **xC**, **Cx** (store this one into a variable **b**), $(\mathbf{Cx})^T$, and $\mathbf{x}^T\mathbf{C}^T$. Write a few comments explaining the results of each expression and how they compare.

   (c) Use the built-in `inv` method to invert the matrix **C**, store it into `Inv_C`. Then compute **Inv_C*b**. Give a brief mathematical explanation of your result.

3. (5 pts) Suppose we have six cities with airports: Portland, San Francisco, Chicago, New York, Moscow, and Tokyo. We are interested in counting the number of ways we can travel from one city to another with at most n stopovers.

   Suppose for example that there are direct flights from:

   - Portland to San Francisco;
   - San Francisco to any of the remaining five cities;
   - Chicago to San Francisco, New York, and Moscow;
   - New York to Portland, San Francisco, Chicago, and Moscow;
   - Moscow to Chicago, New York, and Tokyo;
   - Tokyo to San Francisco, New York, and Moscow;

and we want to count the number of ways we can get from Portland to Moscow with at most 3 stopovers - e.g., the trip Portland - San Francisco - Tokyo - New York - Moscow is a trip with 3 stopovers. Also, note that for the sake of algorithmic simplicity, we shall include in our count ways that go back and forth between cities - e.g. the trip Portland - San Francisco - Portland - San Francisco - New York is another trip with 3 stop overs.

At first glance this problem seems to have little to do with matrices and linear algebra, but it turns out that there is a very easy and elegant way of solving problems of this type using something called an incidence matrix.

The first step is to number our cities in the order they are listed: Portland will be 1, San Francisco will be 2, and so on. Then, let us set up our incidence matrix **A** by the following rule: if we can get from the i^{th} city to the j^{th} city, then the entry $\mathbf{A}_{ij}$ of the matrix **A** will be set to 1. Otherwise we set that entry to 0. By convention we will set all of the diagonal entries, $\mathbf{A}_{ii}$ to 0, because you cannot take a flight from a city to itself. Thus we obtain the following matrix **A** for the given situation:

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 & 0 \end{bmatrix}$$

(a) Compute $\mathbf{A}^2$. What is the meaning of $\mathbf{A}^2$? Write a paragraph or so in your IJulia notebook comments. **Hint:** I recommend performing this computation on paper (in addition to doing it in Julia) and writing all the intermediary steps, in order to better understand its meaning (it will have to do with stopovers).

(b) Given the matrix **A** and your insights from part (a) above, solve the original problem of getting from Portland to Moscow with *at most 3* stopovers. Make sure that your solution agrees with your intuition (i.e. get a pencil and paper out and double check your answer). Note, you do not have to worry about eliminating routes that repeat cities, but if you do ... kudos!

Lecture 5 [1]

Solving Systems of Linear Equations

Systems of linear equations are ubiquitous in scientific computing. They can occur naturally, or they can be the result of approximating nonlinear or differential equations from a variety of different fields (we will look at some examples below). Therefore, understanding how such systems are solved and being able to come up with efficient and good solutions for these systems are important skills for computational scientists.

A system of linear equations is a collection of equations involving the same set of variables, e.g.

$$\begin{aligned} 3x_1 + 2x_2 - x_3 &= 1 \\ 2x_1 - 2x_2 + 4x_3 &= -2 \\ -x_1 + \frac{1}{2}x_2 - x_3 &= 0 \end{aligned}$$

Solving the system above means finding values for x_1, x_2, x_3 that satisfy each equation in the system. Note that for a system involving only a few variables (less than 10 or so), such as the one above, we could be persuaded to solve it by hand using the substitution method we've learned in Calculus. However, the more interesting systems will involve hundreds of variables and the above notation, as well as our knowledge of the substitution method, won't scale.

We need to model such problems with vector and matrices. Given the definition of matrix-vector multiplication from last week, we can re-write the above system as follows:

$$\begin{bmatrix} 3 & 2 & -1 \\ 2 & -2 & 4 \\ -1 & \frac{1}{2} & -1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 \\ -2 \\ 0 \end{bmatrix}$$

In general, for the system of m linear equations involving n unknowns below:

$$\begin{aligned} a_{1,1}x_1 + a_{1,2}x_2 + \ldots + a_{1,n}x_n &= b_1 \\ a_{2,1}x_1 + a_{2,2}x_2 + \ldots + a_{2,n}x_n &= b_2 \\ &\vdots \\ a_{m,1}x_1 + a_{n,2}x_2 + \ldots + a_{m,n}x_n &= b_m \end{aligned}$$

We have the following matrix equation:

$$\begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{n,2} & \cdots & a_{m,n} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix}$$

If we name the $m \times n$ matrix **A**, the vector of unknowns **x**, and the right-hand side vector **b**. We get the following concise vectorized notation for our linear system: **Ax = b**.

Note that the generalized system above allows us to assume that we have a different number of equations than unknowns, however for the sake of introduction we will only consider square systems (i.e. $m = n$) in this lecture.

[1] Sources for Lecture 5: Chapter 2 from Heath's *Scientific Computing: an Introductory Survey* and Wikipedia

Singular Matrices

A square $n \times n$ matrix **A** is singular if the following equivalent properties hold:

1. it has no inverse: that is, there does not exist a matrix **M** such that **AM** = **MA** = **I** (where **I** is the identity matrix);
2. its determinant is 0. The determinant of a matrix is a special number that tells us interesting things about the solution of a system (kinda like the discriminant tells us interesting things about the solution of a quadratic function). Basically, if the determinant of **A** is 0, then we either have no solution to the system of linear equations or infinitely many solutions;
3. its rank is less than n. The rank of a matrix is the maximum number of linearly independent rows or columns - that is, the maximum number of rows or columns that are not linear combinations (scalar multiples and addition) of other rows or columns in the matrix;
4. $\mathbf{Az} = \mathbf{0}$ for $\mathbf{z} \neq \mathbf{0}$, as this basically implies 3 above.

Otherwise, the matrix **A** is said to be *nonsingular*. If that's the case then the inverse of **A**, denoted $\mathbf{A}^{-1}$, exists and the equation $\mathbf{Ax} = \mathbf{b}$ has the unique solution $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$, regardless of the vector **b**. On the other hand, a system of linear equation whose coefficient matrix is singular, may or may not have solutions, and the number of solutions will be determined by the vector **b**.

Example 5.1 Consider the following system:

$$\begin{aligned} 2x_1 + 3x_2 &= b_1 \\ 5x_1 + 4x_2 &= b_2 \end{aligned}$$

We can translate it to vectorized form as follows:

$$\begin{bmatrix} 2 & 3 \\ 5 & 4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$$

Note that `A = [2 3; 5 4]` (try `det(A)` in Julia for a quick check) is nonsingular. Thus a unique solution exists and this fact does not depend on the value of the vector **b** (quick exercise: find the unique solution to the system above if `b = [8; 13]`).

Now consider the following system:

$$\begin{aligned} 2x_1 + 3x_2 &= b_1 \\ 4x_1 + 6x_2 &= b_2 \end{aligned}$$

with vectorized form:

$$\begin{bmatrix} 2 & 3 \\ 4 & 6 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$$

Right away we see that the second row of the matrix above has values that are twice the values of the first row for each corresponding value. Thus, whether a solution exists or not will depend on the vector **b**. If, for example $\mathbf{b} = \begin{bmatrix} 4 \\ 7 \end{bmatrix}$, then no solution exists. However, if $\mathbf{b} = \begin{bmatrix} 4 \\ 8 \end{bmatrix}$, then $\mathbf{x} = \begin{bmatrix} \alpha \\ \frac{4-2\alpha}{3} \end{bmatrix}$.

Note: Systems of linear equations have intuitive geometric interpretation. If we have two unknown variables, the equations are lines on the two dimensional plane. If we have three unknown variables, they are planes in the three dimensional space. And so on...

Gaussian Elimination

A simple way of solving a system of linear equations that is a bit more algorithmic and efficient than the good old substitution method is via Gaussian elimination of variable. Given a system of linear equations in vectorized for the algorithm sequentially performs elementary row operations on the matrix of coefficients until the lower-left hand corner of the matrix is filled with 0's (as far as possible anyway). The elementary row operations (these are operations that will not change the linear system) are:

1. Exchanging two rows;
2. Multiply a row by a non-zero scalar;
3. Adding a multiple of a row to another row.

Here is a detailed example of the algorithm being applied to a small linear system. This same technique can be use to compute the determinant of a square matrix or to find the inverse of an invertible (i.e. nonsingular) matrix, amongst other applications.

Here is the pseudocode for the Gaussian elimination algorithm. As an exercise, implement this algorithm in Julia:

```
for k = 1 ... m:
   //Find pivot for column k, that is the index i that maximizes |A[i,k]|:
   i_max  := argmax (i = k ... m, abs(A[i, k]))
   if A[i_max, k] = 0
     error "Matrix is singular!"
   swap rows(k, i_max)
   //Do for all rows below pivot:
    for i = k + 1 ... m:
     //Do for all remaining elements in current row:
      for j = k ... n:
       A[i, j]  := A[i, j] - A[k, j] * (A[i, k] / A[k, k])
     //Fill lower triangular matrix with zeros:
     A[i, k]  := 0
```

Matrix Factorization: LU decomposition

The standard algorithm for solving matrix equations using computers is based on the Gaussian elimination algorithm explained above with some modifications to ensure better efficiency and more accurate results. The commonly used method is called LU decomposition (Lower triangular - Upper triangular decomposition) introduced by Alan Turing.

As we've seen in the previous section, applying the Gaussian elimination process to a matrix transforms it into a upper triangular matrix, so that we can easily solve the problem with back-substitution. It can be shown with a bit of theory that there is a matrix $\mathbf{L}$ such that $\mathbf{A} = \mathbf{LU}$ where $\mathbf{A}$ is the original matrix of coefficients and $\mathbf{U}$ is the upper triangular matrix obtained from the Gaussian elimination process.

If we can figure out the LU-decomposition for a matrix $\mathbf{A}$, then the linear system $\mathbf{Ax} = \mathbf{b}$ can be re-written as $\mathbf{LUx} = \mathbf{b}$. This system is quick and easy to solve computationally, as we can let $\mathbf{y} = \mathbf{Ux}$ and solve the lower triangular system $\mathbf{Ly} = \mathbf{b}$ via forward-substitution, then the upper triangular system $\mathbf{Ux} = \mathbf{y}$ by back-substitution.

Example 5.2 Consider the following linear system in three dimensions:

$$\begin{aligned} 2x_1 + 4x_2 - 2x_3 &= 2 \\ 4x_1 + 9x_2 - 3x_3 &= 8 \\ -2x_1 - 3x_2 + 7x_3 &= 10 \end{aligned}$$

As usual, we can re-write it in matrix notation as such:

$$\mathbf{Ax} = \begin{bmatrix} 2 & 4 & -2 \\ 4 & 9 & -3 \\ -2 & -3 & 7 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 2 \\ 8 \\ 10 \end{bmatrix} = \mathbf{b}.$$

We can perform Gaussian elimination by subtracting two times the first row from the second row, and adding the first row to the third row - this will eliminate the sub-diagonal entries in first column. Then, we subtract the second row from the third row - this will eliminate the sub-diagonal entry in second column.

This series of operations can be represented by the following matrix multiplication:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -1 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 2 & 4 & -2 \\ 4 & 9 & -3 \\ -2 & -3 & 7 \end{bmatrix} = \begin{bmatrix} 2 & 4 & -2 \\ 0 & 1 & 1 \\ 0 & 0 & 4 \end{bmatrix} = U$$

The result is an upper triangular matrix which we will call **U**. Since we want to find two matrices **L** and **U**, such that **A= LU**, then given the above matrix multiplication we conclude that **L** is the following lower triangular matrix:

$$\left(\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -1 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix} \right)^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ -1 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ -1 & 1 & 1 \end{bmatrix}$$

(for more information on why the above equality makes sense, please read this chapter from Strang's Linear Algebra textbook)

Now set **Ux = y**, then we can easily solve for **y** in the system **Ly = b** using forward-substitution:

$$\begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ -1 & 1 & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} 2 \\ 8 \\ 10 \end{bmatrix}$$

$\Rightarrow y_1 = 2, y_2 = 4$ and $y_3 = 8$. Now we use back-substitution to solve for **x** in the system **Ux = y**:

$$\begin{bmatrix} 2 & 4 & -2 \\ 0 & 1 & 1 \\ 0 & 0 & 4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 2 \\ 4 \\ 8 \end{bmatrix}$$

$\Rightarrow x_3 = 2, x_2 = 2$ and $x_1 = -1$, so we get the solution $\mathbf{x} = \begin{bmatrix} -1 \\ 2 \\ 2 \end{bmatrix}$.

Homework 5 (C)

The purpose of this homework is to further improve your understanding of how the LU decomposition of a matrix **A** can be used to quickly and easily solve the matrix equation **Ax** = **b**. You will also gain a bit of historical knowledge on Gaussian elimination, as well as be introduced to the idea of pivot elements for obtaining more accurate results when performing Gaussian elimination computationally.

Exercises:

1. (5 pts) Write a Julia function that takes as input the matrices **L** and **U** (from the LU-decomposition of some matrix **A**) and a vector **b**, and outputs the solution vector **x** from the matrix equation **Ax** = **b**. You algorithm should mimic example 5.2 above in that it should perform forward-substitution followed by back substitution appropriately. Then compare your result with `A\b`.

2. (5 pts) Read this paper and write a two page summary using LaTeX.

3. (5 pts) Read this chapter and answer the following questions (your answers should be typeset):

   (a) What is a pivot element for a matrix?

   (b) Why is it important to use pivoting when performing Gaussian elimination?

   (c) Solve exercise 22 on page 541 as described there.

Commit and push your .ipynb, .tex, and .pdf files for this homework into your GitHub private repository by 5/12 at 11:59 PM.

Lecture 6

Linear Regression with One Variable

Linear regression is a statistical learning approach for quantitatively predicting an outcome $\mathbf{y}$ on the basis of predictor variables $\mathbf{x}_j$, such that the relationship between the outcome and the variables is linear. Mathematically we write

$$\mathbf{y} \approx \theta_0 + \theta_1\mathbf{x}_1 + \theta_2\mathbf{x}_2 + \ldots + \theta_n\mathbf{x}_n$$

that is $\mathbf{y}$ can be approximated by a linear combination of n predictor variables.

In this lecture we will focus our attention on single variable regression and leave the multivariable case for the next lecture. In the single variable case we have that the outcome $\mathbf{y}$ can be approximated by a linear relationship involving only one variable $\mathbf{x}$, or mathematically

$$\mathbf{y} \approx \theta_0 + \theta_1\mathbf{x}$$

Supervised Learning

There are different contexts in which we may discuss the topic of linear regression. In this lecture, we will study it in the light of machine learning and thus use vocabulary pertinent to this field. In machine learning there are two main sorts of problems: supervised learning and unsupervised learning problems. In this course we will only address supervised learning problems.

In supervised learning, we wish to infer a function that classifies or fits future data, based on a training data set (that is, data that we already have collected and know the "right" output for each input). Let us motivate our discussion with an example. Consider the following training data set below, where we have on the left column the number of years for which a person went to school and on the right column the income they now make, in thousands of dollars per year.

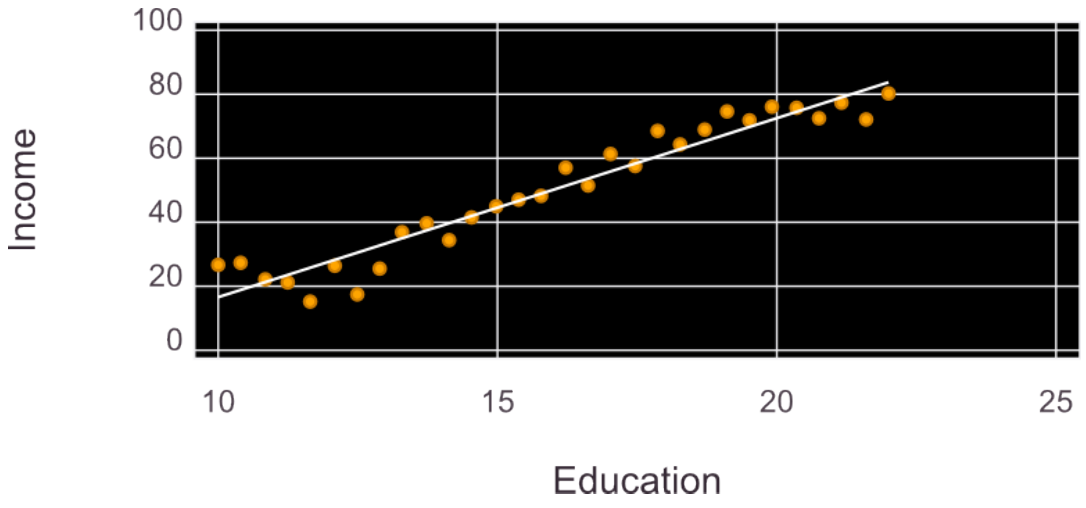
```
"Education"    "Income"
 10.0           26.6588
 10.4013        27.3064
 10.8428        22.1324
 11.2441        21.1698
 11.6455        15.1926
 12.087         26.399
 12.4883        17.4353
 12.8896        25.5079
 13.291         36.8846
 13.7324        39.6661
 14.1338        34.3963
 14.5351        41.498
   ⋮
```

Here our "x" or input is the education level of an individual, while our "y" or output is the income level of the same individual. Since we have many inputs for different observations (say m of them), then we put them all in a $m \times 1$ vector $\mathbf{x}$. Similarly, $\mathbf{y}$ is a $m \times 1$ vector of outputs. In linear regression, we wish to infer the function of the line that best fits data in a certain input-output relationship, that is, we wish to come up with a linear function

$$\mathbf{h}_\theta(\mathbf{x}) = \theta_0 + \theta_1\mathbf{x}$$

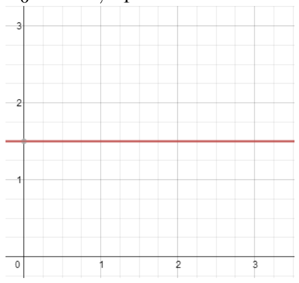
that best approximates our output vector $\mathbf{y}$.

In the formula above we have that θ_j's are the parameters or the weights of the linear map from $\mathbf{x}$ to $\mathbf{y}$. So we want to find the parameters (in this case they are the slope and y-intercept of our line) that give us a line of best fit:

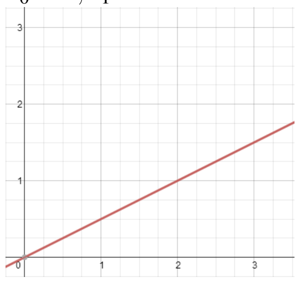


So how do we choose the parameters θ_j? In this lecture we will do that via the Gradient Descent Method. In the next lecture we will generalize linear regression and gradient descent for multiple variables (that is, multiple input vectors $\mathbf{x}_j$), and learn yet another technique we can use for solving for the parameters θ_j. But before we get into all that, let us get a better intuition for these parameters, simply by experimenting with a few different combinations of θ_0 and θ_1:

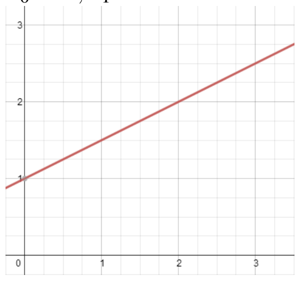
$h(x) = \theta_0 + \theta_1 x$,
$\theta_0 = 1.5, \theta_1 = 0$:



$h(x) = \theta_0 + \theta_1 x$,
$\theta_0 = 0, \theta_1 = 0.5$:



$h(x) = \theta_0 + \theta_1 x$,
$\theta_0 = 1, \theta_1 = 0.5$:



Note how changing the parameters θ_j changes the slope and position of our line in the plane. As an exercise, I encourage students to read different data sets in julia (with one input variable, where the output is quantitative), scatter plot the data and play around with manually picking the slope and intercept parameters for the line that best fits the data visually.

The Cost Function

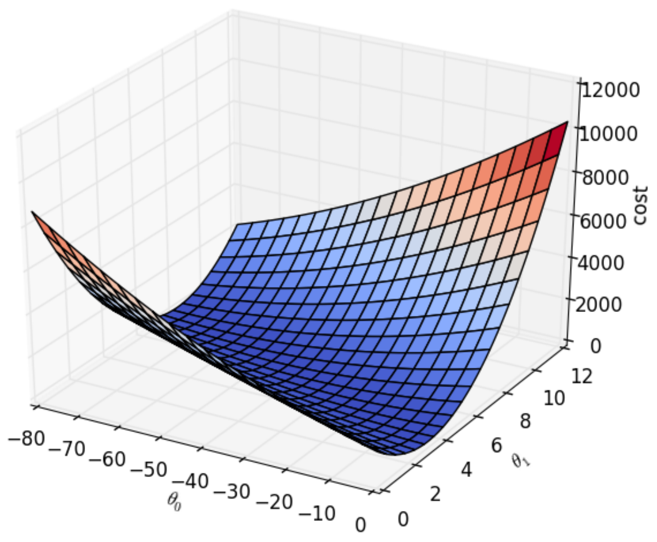
In order for us to be able to apply the gradient descent algorithm for solving our liner regression problem, we will need a function to minimize (as we will see later gradient descent is an algorithm for finding the input that minimizes the output value of a function). This function is called the cost function and it is defined as follows:

$$C(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2$$

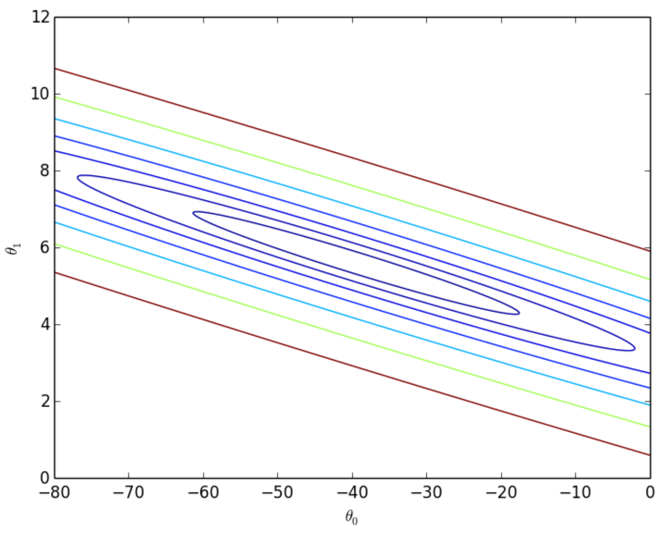
This function outputs the sum of the squared errors between the predicted outputs and the actual outputs, given inputs θ_0 and θ_1. Thus, minimizing this function means getting the θ_0 and θ_1 that causes C to be smallest. It seems intuitive to minimize the magnitude of the errors, but you may be wondering why minimize the squared errors versus just the absolute value of the errors. For our purposes, the explanation is simple - minimization means taking derivatives, and therefore the cost function must be continuous and differentiable. Also, Gauss proved that minimizing the cost function C defined above (aka the ordinary least squares method) gives the best estimate to the linear regression problem.

Note that the cost function is a multivariate quadratic function. If C were a quadratic function of one variable, then in a two dimensional space its graph would look like a concave-up parabola. Such a parabola would have a global minimum at the vertex. Analogously, in our case C's graph lives in a three dimensional space and it is a "bowl-up" shaped surface, with a global minimum at the base of the bowl.

The surface plot:



The contour plot:



Gradient Descent

In this section we will explore the gradient descent algorithm for optimizing the cost function defined earlier. If you are familiar with Newton's algorithm for finding the roots of a function, you may view the gradient descent algorithm as a generalization of Newton's algorithm. The gradient descent algorithm is a type of search algorithm, that is we start with an initial guess and then we update our guess iteratively by some rule. The rule for gradient descent is that from our previous guess we should step down (with some step size) toward the direction of steepest descent. If we iterate this rule in a "bowl-up" shaped surface with a not-to-large step size, we shall converge to a minimum.

Gradient Descent Algorithm for Bivariate Functions

```
repeat until convergence {
    for $j = 0$ and $j = 1$, simultaneously update:
        $\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} C(\theta_0, \theta_1)$
}
```

So we take a guess on each θ, then update each θ simultaneously by taking a step of size α in the respective direction of steepest descent as dictated by the partial derivative of the cost function with respect to the θ that is being updated. The intuition here is that the slope of the tangent line to our current point is the steepest slope we can get on a line that would still include the point.

In machine learning, we call the α parameter "the learning rate". An important note is that if α is too large our algorithm may diverge, as our steps will be so large as to "climb the walls" of our surface. On the other hand, if α is too small, our algorithm may take a long time to converge. Tweaking α can be seen as more of an art than a science and it will depend on your particular problem. The best way to check that you are picking a good α is to graph (or otherwise output) your cost and verify that it is indeed decreasing and the rate of decrease is not too slow.

Using basic Calculus you can easily verify that:

$$j = 0 \Rightarrow \frac{\partial}{\partial \theta_0} C(\theta_0, \theta_1) = \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})$$

$$j = 1 \Rightarrow \frac{\partial}{\partial \theta_1} C(\theta_0, \theta_1) = \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x^{(i)}$$

So now we have a powerful tool for determining the θ parameters in your linear regression model. Please see the accompanying IJulia notebook illustrating linear regression applied to the education versus income data for more details.

Homework 6 (P)

The purpose of this homework is to have you practice the method of ordinary least squares and the gradient descent algorithm for solving a linear regression problem. For best results completing this homework, please review the "Lecture 6" notes above and the IJulia notebook containing a detail example of linear regression.

Exercises:

The Advertising data set (found in "Advertising.csv" on Canvas or GitHub) has data pertaining to the sales in thousands of units versus the advertising budget in thousands of dollars of different advertising media such as TV, radio and newspaper.

1. (9 pts) Using linear regression with least squares and gradient descent, as outlined in lecture 6, determine the line that best fits the relationship between sales and advertising budget for each medium.
2. (2 pts) Plot your solution line against the data for each feature considered (that is, sales versus TV budget, radio budget and newspaper budget).
3. (2 pts) Plot the surface of your cost function for each feature considered.
4. (2 pts) Compare your results. Is there a medium for advertising that you would claim to be best above all others? Which one and why? Write a short report of your findings using LaTeX. Make sure to include your formulas and figures obtained in the process of solving this problem.

Commit and push your .ipynb, .tex, and .pdf files for this homework into your GitHub private repository by 5/19 at 11:59 PM.

Lecture 7[2]

Linear Regression with Multiple Variables

This lecture will continue our discussion of linear regression, the statistical learning approach for quantitatively predicting an outcome $\mathbf{y}$ on the basis of predictor variables $\mathbf{x}_j$. We have already studied in depth the single variable regression problem, now we will explore linear regression with multiple variables (or features, as per our machine learning vocabulary):

$$\mathbf{y} \approx \theta_0 + \theta_1\mathbf{x}_1 + \theta_2\mathbf{x}_2 + \ldots + \theta_n\mathbf{x}_n$$

that is $\mathbf{y}$ can be approximated by a linear combination of n predictor variables.

As an example we will consider the advertising training data set from homework 6, but instead of looking at how each feature affects sales individually, we will look at how they do so combined. Recall that sales is in thousands of units, and the different media advertising budgets are in thousands of dollars.

```
In [1]: ad = readcsv("Advertising.csv")

Out[1]:
201x4 Array{Any,2}:
    "TV"     "Radio"    "Newspaper"    "Sales"
 230.1    37.8        69.2           22.1
  44.5    39.3        45.1           10.4
  17.2    45.9        69.3            9.3
 151.5    41.3        58.5           18.5
 180.8    10.8        58.4           12.9
   8.7    48.9        75.0            7.2
  57.5    32.8        23.5           11.8
 120.2    19.6        11.6           13.2
   8.6     2.1         1.0            4.8
 199.8     2.6        21.2           10.6
  66.1     5.8        24.2            8.6
 214.7    24.0         4.0           17.4
   ⋮
```

Here our "x" values (or inputs, or features) are the different media advertising budgets - each row is an advertising campaign, while our "y" (or output) is the total number of units sold as a result of each of the advertising campaigns. Since we have many advertising campaigns (say m of them) with three features, then we put them all in a $m \times 3$ matrix $\mathbf{X}$. Similarly, $\mathbf{y}$ is a $m \times 1$ vector of outputs. As before, we wish to infer the function of the "line" (in this case a three-dimensional plane) that best fits data in a certain input-output relationship, that is, we wish to come up with a multivariate linear function

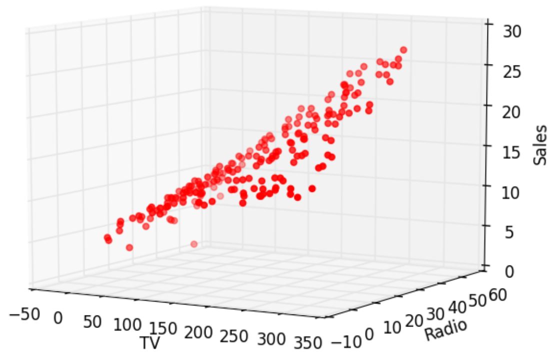
$$\mathbf{h}_\theta(\mathbf{x}) = \theta_0 + \theta_1\mathbf{x}_1 + \theta_2\mathbf{x}_2 + \ldots + \theta_n\mathbf{x}_n$$

that best approximates our output vector $\mathbf{y}$.

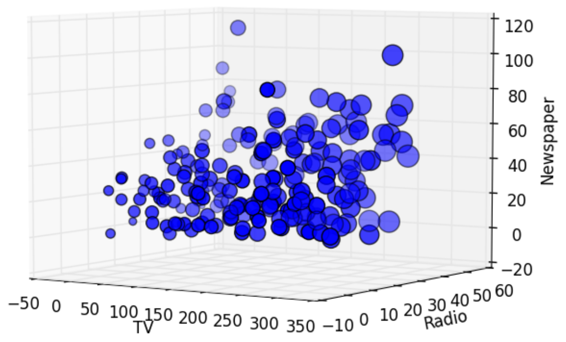
In the formula above we have that θ_j's are the parameters or the weights of the linear map from $\mathbf{X}$ to $\mathbf{y}$. Since we will want to vectorize our code for convenience, we add a column of ones in front of the matrix $\mathbf{X}$ to account for the θ_0 parameter, as such `X = [ones(m,1) X]`.

[2]Source for Lecture 7: Stanford's Machine Learning Lecture Notes

For the sake of showing a picture, here is the scatter plot for sales (on the z-axis) versus the advertisement budgets for TV and radio:



If the TV and Radio budgets were the only features in our advertising model, then the goal of linear regression would be for use to fit a two dimensional plane such that the squared errors are minimized, just as in the single variable case. So we can see that the ideas we learned last week can be generalized to multiple variable. The down side of having more than two features, however, is that we will have a hard time graphing our data! We can try some cleaver tricks (such as having TV, Radio and Newspaper on the x, y, and z axis respectively, while sales is represented by the size of the circles...), but at the end of the day we need to rely on our numbers.



In this lecture we will continue to explore the method of *gradient descent* to find the parameters θ_j. We will also learn about the *normal equation* method, an analytic way for exactly solving for the parameters θ_j, which works well if your data is not too large.

For using the gradient descent method we will still need the *cost function*, which will remain mostly the same as before. The only difference will be $\mathbf{h}_\theta(\mathbf{x})$ (recall here that each feature $\mathbf{x}_j$ is a $m \times 1$ vector representing different observed values):

$$\mathbf{h}_\theta(\mathbf{x}) = \theta_0 + \theta_1\mathbf{x}_1 + \theta_2\mathbf{x}_2 + \ldots + \theta_n\mathbf{x}_n$$

If we define $\mathbf{x}_0$ = `ones(m,1)`, that is a vector of m ones, then we can rewrite $\mathbf{h}_\theta(\mathbf{x})$ as such:

$$\mathbf{h}_\theta(\mathbf{x}) = \theta_0\mathbf{x}_0 + \theta_1\mathbf{x}_1 + \theta_2\mathbf{x}_2 + \ldots + \theta_n\mathbf{x}_n = \mathbf{X}\theta$$

where $\mathbf{X}$ is the $m \times n$ matrix $\begin{bmatrix} | & | & & | \\ \mathbf{x}_0 & \mathbf{x}_1 & \ldots & \mathbf{x}_n \\ | & | & & | \end{bmatrix}$, while θ is the $n \times 1$ column vector $\begin{bmatrix} \theta_0 \\ \theta_1 \\ \vdots \\ \theta_n \end{bmatrix}$.

Thus our "new" cost function looks a lot like the old one, however it's input now is the $n \times 1$ column vector θ:

$$C(\theta) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2$$

Gradient Descent for Multiple Variables

Since the cost function did not change too dramatically, the gradient descent recipe will also only change slightly:

Gradient Descent Algorithm

```
repeat until $C$ converges to a minimum {
    for $j = 0, 1, ..., n$, simultaneously update:
        $\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} C(\theta)$
}
```

where

$$\frac{\partial}{\partial \theta_j} C(\theta) = \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)}) \cdot x^{(i)}$$

And the interpretation is still the same as before: we take a guess on each θ, then update each θ simultaneously by taking a step of size α in the respective direction of steepest descent as dictated by the partial derivative of the cost function with respect to the θ that is being updated.

Good Practice: Feature Scaling

A good practice technique that becomes important when we have more than one feature in our model is called *feature scaling*. The idea here is that most of the time we have features that are on widely different ranges and scales from each other. For example, if we want to predict housing prices in Seattle, we might want to use the size of the house in square feet as a feature (say $\mathbf{x}_1$) and the number of bedrooms as another (say $\mathbf{x}_2$). So the values in $\mathbf{x}_1$ will range from 300 to 3000, while the values in $\mathbf{x}_2$ will range from 0 to 5.

You can imagine that his will cause the graph of our cost function to be narrow along the θ_1 dimension and wide along the θ_2 dimension. Thus, depending on where your initial θ finds itself, the gradient descent algorithm may take much longer to converge to the minimum value of $C(\theta)$, and in this case increasing α may not help (and may even cause a quicker divergence).

Mean Normalization and Scaling

There are different ways to fix the problem above. The idea in all of them is to make all of our features have values that lie in the same range roughly from -1 to 1 (times 2 or 3 is also ok). A preferred technique is *mean normalization* with standard deviation scaling, that is for each value $x^{(i)}$ in each feature vector $\mathbf{x}_j$, replace is with:

$$\frac{x^{(i)} - \mu_j}{\sigma_j}$$

where μ_j is the mean value of the values in $\mathbf{x}_j$, and σ_j is the standard deviation of the values in $\mathbf{x}_j$. Another common practice would be to scale with the size of the range of values in a particular feature vector (i.e. max - min) instead of the standard deviation.

And as before, the best way to check that your feature scaling is working and that you are picking a good α is to graph (or otherwise output) your cost versus the number of gradient descent iterations and verify that the cost is indeed decreasing and the rate of decrease is not too slow at each iteration.

The Normal Equation

$$A^T Ax = A^T b$$

So far we've been using the gradient descent method for performing linear regression on pretty small problems (gradient descent is meant for big problems, i.e. 10000 features or so). This is because we could not solve the matrix equation **Ax = b** with the ideas learned from lectures 4 and 5, since **A** is usually not square and we cannot invert non-square matrices.

The mathematical trick here is: multiply both sides of the matrix equation **Ax = b** on the left by $\mathbf{A}^T$. Thus we get the normal equation above and it turns out that the solution to the normal equation, that is,

$$x = (A^T A)^{-1} A^T b$$

is the *least-squares* solution to the original system of linear equation represented by **Ax = b** (i.e. the solution to the normal equation minimizes the sum of squared errors **b - Ax**). I will not go over the mathematical reason why the normal equation works, as that is beyond the scope of this course, but here is a resource for further exploration.

Given your expertise with matrices by now, you may be wondering what if $\mathbf{A}^T\mathbf{A}$ is singular? Then we cannot find the inverse. That is true...so we find the pseudoinverse (in julia we use the `pinv` function).

For more details on linear regression with multiple variables and the normal equation method for solving non-square linear systems, please see the accompanying IJulia notebook.

Homework 7 (C)

The purpose of this homework is to ensure your conceptual understanding of the topics presented in lecture 7. Please typeset your *complete solution* to each exercise below using LaTeX.

Exercises:

1. (3 pts) Suppose $m = 4$ students have taken some class, and the class had two midterm exams and a final exam. You have collected a data set of their scores on the three exams, which is as follows:

midterm 1	midterm 2	final exam
89	91	96
72	77	74
94	90	87
69	70	78

   You'd like to use linear regression to predict a student's final exam score from their midterm exams scores. Concretely, suppose you want to fit a model of the form $\mathbf{h_\theta(x)} = \theta_0 + \theta_1\mathbf{x}_1 + \theta_2\mathbf{x}_2$, where $\mathbf{x}_j$ is the vector of midterm j scores. Further, you plan mean normalization and scale the features using the size of the range of values. What is the normalized feature $x_1^{(1)}$? (Recall that the superscript indicates the particular observation, while the subscript indicates the particular feature). Please show your work.

2. (3 pts) You run gradient descent for 15 iterations with $\alpha = 0.3$ and compute $C(\theta)$ after each iteration. You find that the value of $C(\theta)$ decreases slowly and is still decreasing after 15 iterations. Based on this, what is your conclusion and course of action?

3. (3 pts) Why should we perform feature scaling?

4. (3 pts) Suppose you have a data set with $m = 50$ examples and $n = 15$ features for each example. You want to use multivariate linear regression to fit the parameters θ to your data. Should you prefer gradient descent or the normal equation? Why?

5. (3 pts) Find the least-squares solution of the matrix equation $\mathbf{Ax} = \mathbf{b}$ (for the given $\mathbf{A}$ and $\mathbf{b}$ below) using the normal equation $\mathbf{A}^T\mathbf{Ax} = \mathbf{A}^T\mathbf{b}$. You should show your work step by step. Including images or skipping steps is allowed, but typesetting it *completely* using LaTeX is worth +2 points.

$$A = \begin{bmatrix} 1 & -2 \\ -1 & 2 \\ 0 & 3 \\ 2 & 5 \end{bmatrix} \quad b = \begin{bmatrix} 3 \\ 1 \\ -4 \\ 2 \end{bmatrix}$$

Commit and push your .tex, **and** *.pdf files for this homework into your GitHub private repository by 5/26 at 11:59 PM.*

Lecture 8

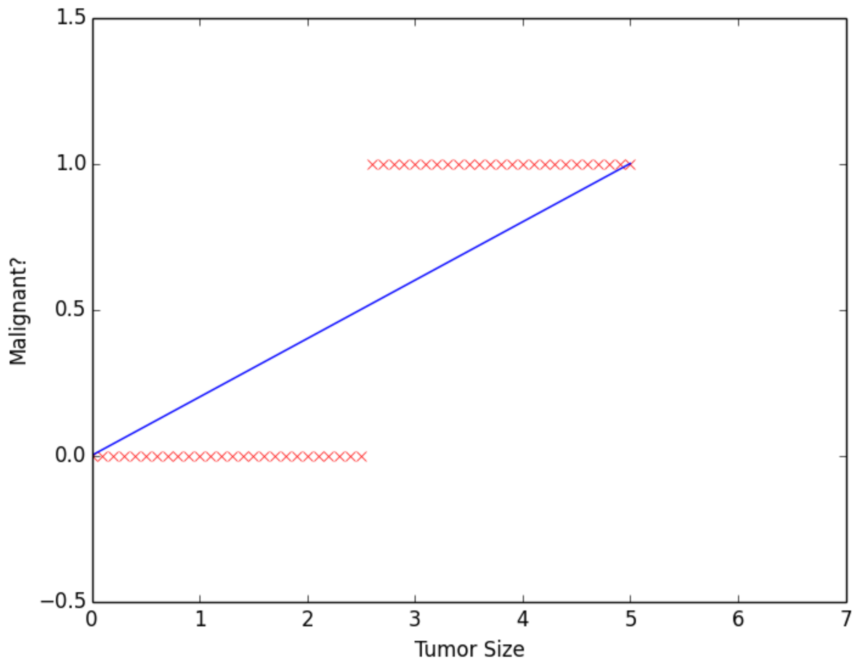
Logistic Regression

Last week we learned about linear regression, where our goal was to predict a quantitative outcome $\mathbf{y}$ given a linear relationship between multiple inputs $\mathbf{x}_j$. However for many different problems, when given an observation, we may wish to make a qualitative prediction instead. For example, we may wish to determine whether or not a patient has a high cancer risk or whether or not an email is spam. In this lecture we will study logistic regression, an approach used in statistics and machine learning for predicting qualitative outcomes.

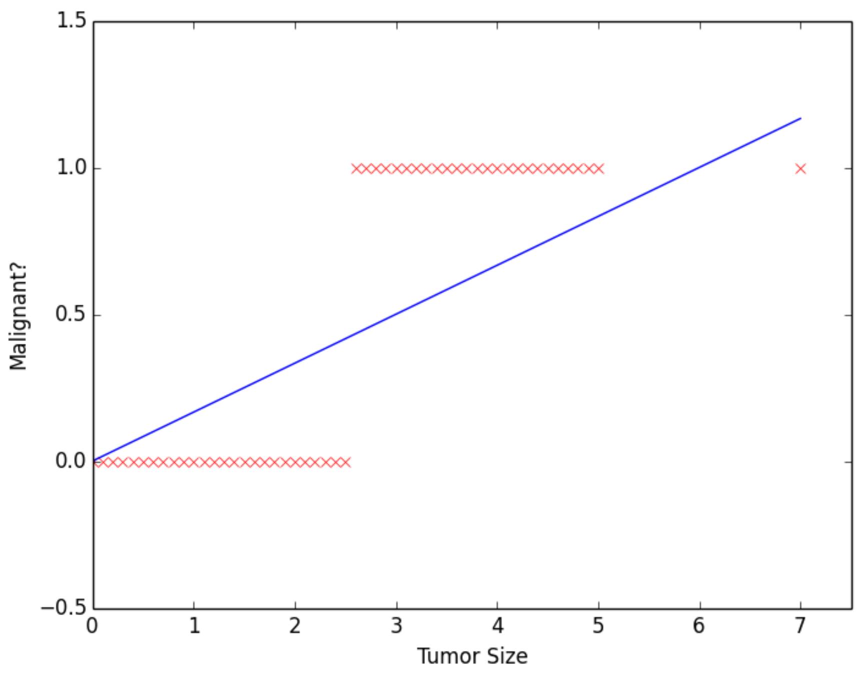
In machine learning, this sort of approach is more commonly known as *classification*, due to the fact that we seek to assign our observations (or training examples) to a category or class, thus classifying them. So now our outcome $\mathbf{y}$ is a vector of categories or classes. Here, since we will be studying logistic regression, we will consider two classes for the outcomes: 0 or the "negative" class (e.g. not high cancer risk, not spam); and 1 or the "positive" class (e.g. high cancer risk, spam).

Please note that there are other approaches than logistic regression for classifying data. Also note that there are other approaches and techniques that would accommodate predicting for more than two classes. We will briefly address the approach of applying logistic regression many times in order to classify data into more than two categories, however a deeper study of different classification techniques is left to the reader. At the end of this lecture, I will provide resources for further study of classification.

The first question we should ask ourselves is: why do we need a different technique than linear regression? Can't we just apply linear regression and if the outcome for an observation is below 0.5 we predict "negative", otherwise we predict "positive"?



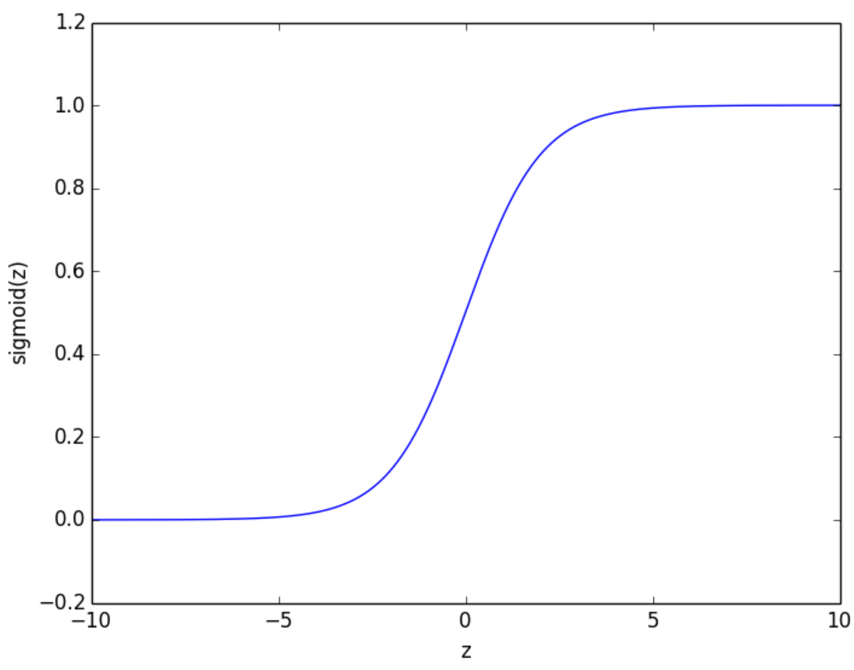
Well, let's think about how an outlier can change our predictions if we were to use linear regression:



In the above picture, this outlier will change the slope of our line of best fit and cause us to classify several positive cases as negative. Therefore, it looks like we cannot rely on linear regression alone to help us classify data. It turns out there is a function in which we can plug in the result of our linear regression in order to get a discretezation of categories that will not be affected by outliers - this function is called the *sigmoid*:

$$\text{sigmoid}(z) = \frac{1}{1+e^{-z}}$$

Here is its graph:



Thus, in our attempt to find a $\mathbf{h}_\theta(\mathbf{x})$ that best approximates the true observed outcome $\mathbf{y}$ (as we did with linear regression), if we were to interpret $h_\theta(x^{(i)})$ as the estimated probability that the outcome y is 1 for a given input $x^{(i)}$ (i.e. the i-th training example), then we can use the sigmoid to perform the classification step. That is, in vector notation for all training examples, we have:

$$\boxed{\mathbf{h}_\theta(\mathbf{x}) = \mathbf{g}(\theta^T \mathbf{x}) = \frac{1}{1+e^{-\theta^T \mathbf{x}}}}\ .$$

Thus, we can predict that $y = 1$ if $h_\theta(x^{(i)}) \geq 0.5$, or $y = 0$ if $h_\theta(x^{(i)}) < 0.5$. In the equation above, notice that $g(z) \geq 0$ when $z \geq 0$ and $g(z) < 0$ when $z < 0$. Also notice that z is defined to be a linear combination of the features $\mathbf{x}_j$ and that our goal is still to obtain the appropriate θ_j's. However, the meaning of this linear relationship is no longer to be the best linear fit to our data (as with linear regression), but instead be the best separator for our data. So now $\theta^T \mathbf{x}$ is the line that separates our data into two classes.

So once again, we must ask ourselves the question : how do we choose the parameters θ_j? Once again the answer is: via the Gradient Descent Method!

The Cost Function for Logistic Regression

In order for us to be able to apply the gradient descent algorithm for solving our logistic regression problem, we will need a function to minimize. Recall the linear regression cost function:

$$C(\theta) = \frac{1}{m}\sum_{i=1}^{m}\frac{1}{2}(h_\theta(x^{(i)}) - y^{(i)})^2\ ,$$

unfortunately simply plugging in the sigmoid for $h_\theta(x^{(i)})$ will yield a non-convex function, and as discussed in previous lectures, the gradient decent algorithm may not converge to the global minimum if the function being minimized is non-convex. Therefore, we may not re-use this cost function for logistic regression.

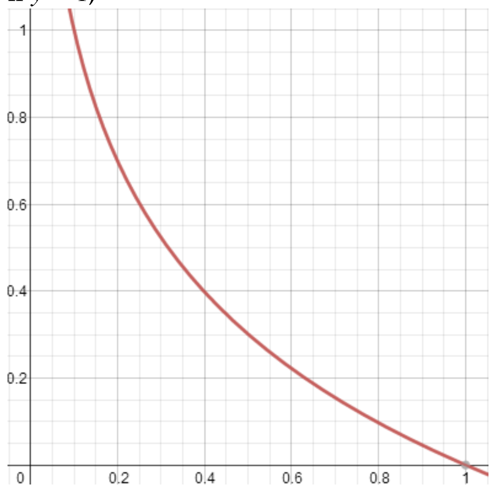
Instead, we will define a different cost function that is convex so that gradient descent can converge to the global minimum:

$$C(\theta) = \frac{1}{m}\sum_{i=1}^{m}\mathrm{Cost}(h_\theta(x^{(i)}), y^{(i)})\ , \text{ where}$$

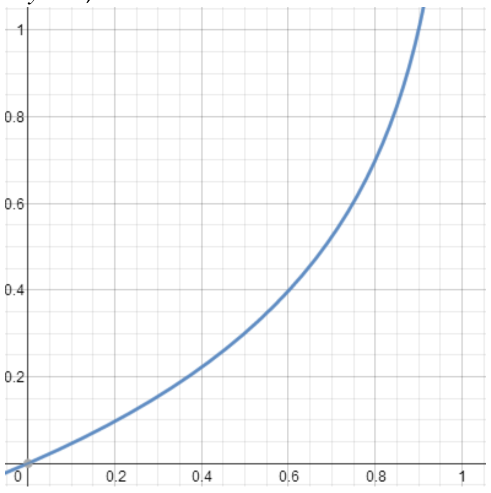
$$\mathrm{Cost}(h_\theta(x^{(i)}), y^{(i)}) = \begin{cases} -\log(h_\theta(x^{(i)})) & \text{if } y = 1 \\ -\log(1 - h_\theta(x^{(i)})) & \text{if } y = 0 \end{cases}$$

If we look at the graph of $\mathrm{Cost}(h_\theta(x^{(i)}), y^{(i)})$, then we get that:

if $y = 1$,



if $y = 0$,



Notice that this cost function does a good job at giving the error between the predicted value and the true value. For example, if we predict that $h_\theta(x^{(i)}) = 1$ and $y = 1$, then by the graph on the left we see that the error is 0. On the other hand, if we predict that $h_\theta(x^{(i)}) = 0$ but $y = 1$, then the error diverges to infinity. Thus this function penalizes the learning algorithm with a very large cost on wrong predictions. The same argument applies to the graph on the right for $y = 0$.

There is a clever way to re-write $\text{Cost}(h_\theta(x^{(i)}), y^{(i)})$ not as a piecewise function. Since either $y = 0$ or $y = 1$, then:

$$\text{Cost}(h_\theta(x^{(i)}), y^{(i)}) = -y^{(i)} \log(h_\theta(x^{(i)})) - (1 - y^{(i)}) \log(1 - h_\theta(x^{(i)})) \; .$$

This we have that the *logistic regression cost function* can be simply written as follows:

$$\boxed{C(\theta) = -\frac{1}{m} \sum_{i=1}^{m} \left[y^{(i)} \log(h_\theta(x^{(i)})) + (1 - y^{(i)}) \log(1 - h_\theta(x^{(i)})) \right]} \; .$$

Now, with the correct cost function we can apply the gradient descent algorithm exactly as before:

Gradient Descent Algorithm - We want to find the vector of parameters θ such that the cost function is minimized:

```
repeat until $C$ converges to a minimum {
    for $j = 0, 1, \ldots, n$, simultaneously update:

        $\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} C(\theta)$

}
```

where

$$\frac{\partial}{\partial \theta_j} C(\theta) = \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x^{(i)}$$

and

$$h_\theta(x^{(i)}) = \frac{1}{1 + e^{-x^{(i)}\theta}}$$

Notice that the derivative of the new cost function turns out to be exactly the same as the derivative of our old cost function (to see how this derivative was derived, please consult Professor's Ng lecture notes). Also, as before you may need to scale and normalize the features in order for gradient descent to perform efficiently.

Brief Note: One-vs-All

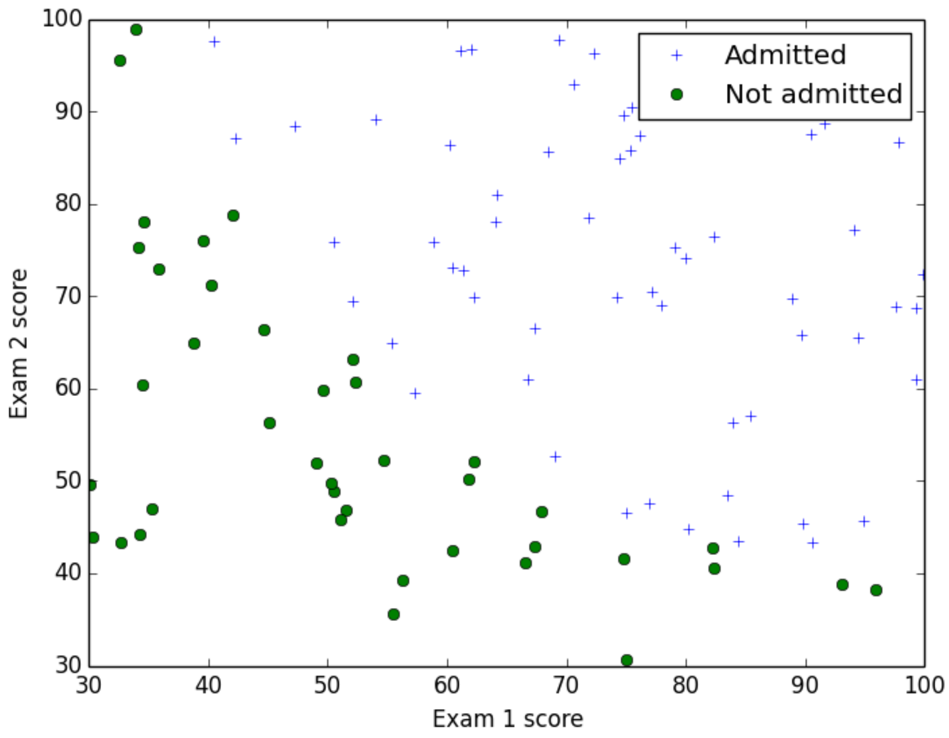
We have just learned about how to solve for 0-1 classification problems, but there are many classification problems out there that are not of the 0-1 type. For example, we may wish to implement email tagging that goes beyond spam versus ham and include other categories such as work, family, friends, etc.

As we have seen the way logistic regression works, we may feel restricted to only using it in cases where we wish to predict outcomes for two classes only. However, that is not the case. Along with a technique called "one-vs-all", we can use logistic regression to predict outcome for more than two classes.

One-vs-all is simply the technique of running multiple logistic regressions where in each run of logistic regression we pick only one of our classes to be the "positive" class (a different one each time), while the rest become the "negative" class. The logistic regression that outputs the *highest* value of $h_\theta(x^{(i)})$ for some input $x^{(i)}$ "wins", that is we classify $x^{(i)}$ into the class k where k is the positive class in the respective logistic regression run.

An Example: Data Classification

We now turn our attention to the companion IJulia notebook for Lecture 8. There we shall explore the application of logistic regression to classify a group of candidates for admission at an university into two classes: accepted versus rejected. Using historical data from previous admission rounds, we will build a model to predict whether a an individual gets admitted based on their grades on two exams. Before we try to build our model, it may be useful to visualize our data:



Lecture 9

Recommender Systems

In this lecture we are going to explore a further application of linear regression: Recommender Systems. These are systems that allows businesses (or otherwise content providers) to more intelligently provide their content based on their customers preferences. For example, your "movies-watched" history on Netflix is input data for a recommender system that tries to predict what else you would like to watch, as it is in Netflix interest to keep you satisfied and watching. Another example is amazon.com, as they have a vast number of products and so it is in their interest to intelligently recommend you things that you are actually likely to buy.

There are two main types of recommender systems:

1. *Content based systems:* these are systems that base recommendations on the content of the media or information. For example, if you watch actions movies on Netflix and rate those positively, chances are that you will enjoy other action movies.

2. *Collaborative filtering systems:* these are systems that base recommendations on similarities amongst users or customers. For example, if you bought a book on amazon that another user also bought along with other books, then maybe you would like those other books too.

Most recommender systems in use are based on these two main types or on some sort of hybrid of these types. There are many techniques from scientific computing and machine learning that can be applied to implement these different types of recommender systems - such as singular value decomposition and clustering analysis (which are not covered in this course).

Here, we will discuss simplified implementations of both content based and collaborative filtering systems that are basically applications of linear regression. We will focus on algorithms that use gradient-based optimization (such as gradient descent or some more advanced optimization algorithm like L-BFGS) to find the least squares solution. However, before we "jump in", let us take a detour through the topic of *regularization*.

Regularization

Regularization refers to the technique of adding extra information to our linear (or logistic) regression in other to give preferential treatment to features that seem more relevantly correlated and penalize features whose correlation may not be that relevant to solving the regression (it can also be use to prevent over-fitting in polynomial regression - which we have not studies, but it is simply trying to find the polynomial of best fit, i.e. the degree of your polynomial solution can be greater than 1).

There are many ways to perform regularization depending on the regression problem one is trying to solve. The regularization that we will use for our linear regression is this lecture is know as Tikhonov regularization. I will leave the more conceptual and theoretical details to the reader to explore and simply state how our cost function and its derivative will be changed in the case of regularization.

Regularized Cost Function and its derivative:

$$C(\theta) = \frac{1}{2m}\left[\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2 + \lambda\sum_{j=1}^{n}\theta_j^2\right]$$

for $j = 0$:

$$\frac{\partial}{\partial\theta_0}C(\theta) = \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_0^{(i)}$$

for $j > 0$:

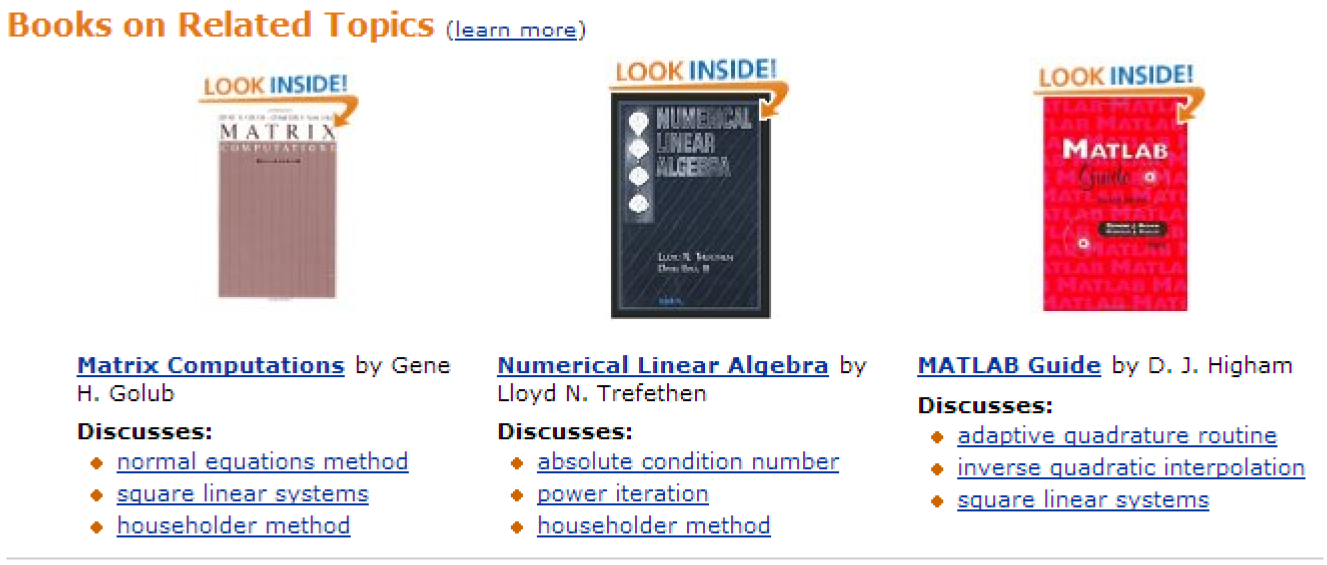
$$\frac{\partial}{\partial\theta_j}C(\theta) = \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)} + \frac{\lambda}{m}\theta_j$$

λ is called the *regularization parameter* - it is a parameter we should tweak in order to reward (or penalize) certain features. Note that we do not regularize θ_0 since it pertains to the "ones" feature and it is not affected by the weight of the features in consideration.

Content Based Systems

In content based systems, the features which describe the content of something being recommended to users are known. However, the user weights for these features are not known. These features are information like amount of action or romance in a movie, or the different categories in which a particular product may fall. For example, if you search for "scientific computing" on amazon.com, you will get a list of books that fall in that category, but not only that, you will get a list in order of relevance.

Clicking one of the results and scrolling down a bit, we find yet another example of content based recommendations:



Books on Related Topics (learn more)

Matrix Computations by Gene H. Golub

Discusses:
- normal equations method
- square linear systems
- householder method

Numerical Linear Algebra by Lloyd N. Trefethen

Discusses:
- absolute condition number
- power iteration
- householder method

MATLAB Guide by D. J. Higham

Discusses:
- adaptive quadrature routine
- inverse quadratic interpolation
- square linear systems

In order to introduce the topic and formulate the problem, let us work on a simplified example:

Book	Alice	Bob	Charlie	Dave
Advanced Math in Greek	5	0	5	0
The Book of Proofs	?	2	5	0
Down-to-Earth Math	0	5	1	4
Everyday Math	0	?	?	?
Intermediate Crazy Math	5	0	?	?

Say we have a matrix as above, where rows correspond to book tiles and columns correspond to readers (i.e. users of this system). An entry $a_{i,j}$ in this matrix is a number ranging from 0 through 5, representing the rating user j gave to book i. If we do not have a rating from a particular user on a particular book, then the entry is represented by a "?".

<u>Definitions:</u>

- n_b is the number of books;
- n_u is the number of users;
- $r(i,j) = 1$ if user j has rated book i (otherwise 0);
- $y^{(i,j)}$ is the rating by user j of book i if $r(i,j) = 1$;
- $\theta^{(j)}$ is the parameter vector for user j;
- $x^{(i)}$ is the feature vector for book i; and
- $(\theta^{(j)})^T(x^{(i)})$ is thus, the predicted rating for user j, for book i.

Furthermore, in this case we know the feature vectors being considered: x_1 = amount of abstract mathematics and x_2 = amount of practical mathematics. That is, someone read all of these books and "graded" them on these features, say on a scale from 0.0 to 1.0.

In order for us to learn $\theta^{(j)}$ (the parameter vector for user j - or the weights user j gives to the features considered) we must solve the following optimization problem:

$$\boxed{\min_{\theta^{(j)}} \left(\frac{1}{2} \sum_{i:r(i,j)=1} ((\theta^{(j)})^T(x^{(i)}) - y^{(i,j)})^2 + \frac{\lambda}{2} \sum_{k=1}^{n} (\theta_k^{(j)})^2 \right)}$$

Note that the expression inside the parenthesis looks just like the regularized cost function we mentioned earlier, with the exception of the division by m. Recall from our first discussion of linear regression that the $\frac{1}{m}$ constant multiplying our cost function was there for mathematical convenience and that it did not affect the minimum. Here, the analogous constant factor would be $\frac{1}{m^{(j)}}$ where $m^{(j)}$ is the number of books rated by user j. Since we will have to optimize for these $\theta^{(j)}$'s across all users j, then the constant factor $\frac{1}{m^{(j)}}$ is no longer convenient.

Thus to learn the parameters for all users j, that is $\theta^{(1)}, \theta^{(2)}, ..., \theta^{(n_u)}$, we need to solve:

$$\boxed{\min_{\theta^{(1)},\theta^{(2)},...,\theta^{(n_u)}} \left(\frac{1}{2} \sum_{j=1}^{n_u} \sum_{i:r(i,j)=1} ((\theta^{(j)})^T(x^{(i)}) - y^{(i,j)})^2 + \frac{\lambda}{2} \sum_{j=1}^{n_u} \sum_{k=1}^{n} (\theta_k^{(j)})^2 \right)}$$

If we are using a gradient based optimization algorithm to solve for the above problem, then we provide the derivative of the cost function:

For $k = 0$ (since j has a different meaning in our current formulation):

$$\frac{\partial}{\partial \theta_0^{(j)}} C(\theta^{(j)}) = \sum_{i:r(i,j)=1} ((\theta^{(j)})^T (x^{(i)}) - y^{(i,j)}) x_0^{(i)}$$

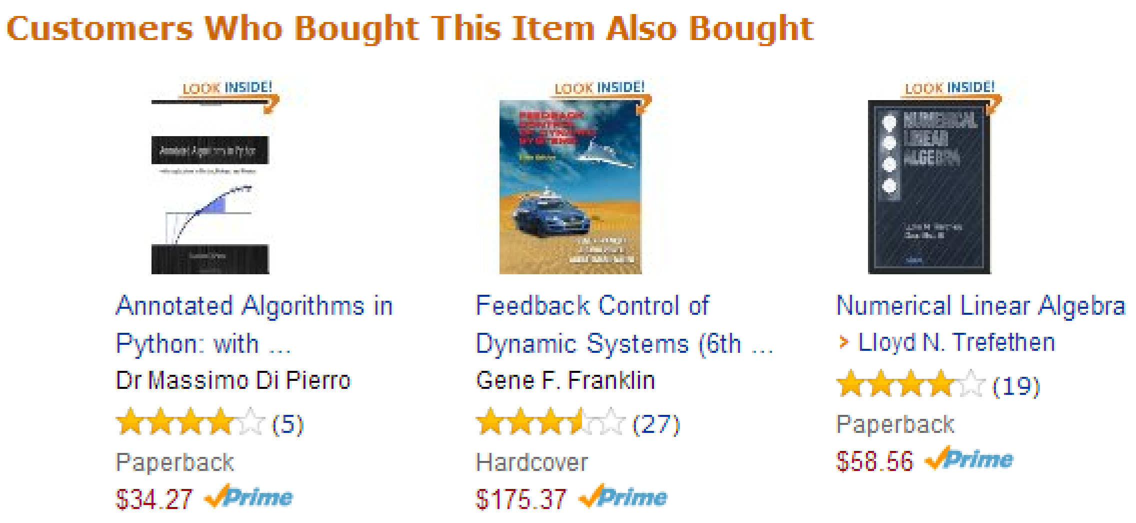
and for $k > 0$:

$$\frac{\partial}{\partial \theta_k^{(j)}} C(\theta^{(j)}) = \sum_{i:r(i,j)=1} ((\theta^{(j)})^T (x^{(i)}) - y^{(i,j)}) x_k^{(i)} + \lambda \theta_k^{(j)}$$

Collaborative Filtering Systems

In collaborative filtering systems, the features which describe the content of something being recommended to users are not known. However, we do know the weights users give to particular features. For example, we know how Sally weighs her genre preferences when it comes to movies - we know that she really likes action movies, then "contains action" is a likely feature of the movies she watches.

On the same amazon.com page I was before, we find an example of collaborative filtering:



Customers Who Bought This Item Also Bought

Annotated Algorithms in Python: with ...
Dr Massimo Di Pierro
★★★★☆ (5)
Paperback
$34.27 ✓Prime

Feedback Control of Dynamic Systems (6th ...
Gene F. Franklin
★★★½☆ (27)
Hardcover
$175.37 ✓Prime

Numerical Linear Algebra
› Lloyd N. Trefethen
★★★★☆ (19)
Paperback
$58.56 ✓Prime

Once again, let us work on a simplified example:

Book	Alice	Bob	Charlie	Dave
Advanced Math in Greek	5	0	5	0
The Book of Proofs	?	2	5	0
Down-to-Earth Math	0	5	1	4
Everyday Math	0	?	?	?
Intermediate Crazy Math	5	0	?	?

This is the same matrix as before, however now we do not have any tags on these books, i.e. there is no "grade" for the amount of abstract mathematics versus practical mathematics. However, we do have the following weight vectors for each user, say:

$$\theta^{(1)} = \begin{bmatrix}0\\5\\0\end{bmatrix}, \theta^{(2)} = \begin{bmatrix}0\\1\\5\end{bmatrix}, \theta^{(3)} = \begin{bmatrix}0\\5\\0\end{bmatrix}, \theta^{(4)} = \begin{bmatrix}0\\0\\5\end{bmatrix}$$

The definitions for the notation we will use here are the same as before. So given vectors $\theta^{(j)}$'s for all n_u users, we wish to learn the vector $x^{(i)}$ (the feature vector for book i). Thus, we must solve the following optimization problem:

$$\boxed{\min_{x^{(i)}} \left(\frac{1}{2} \sum_{j:r(i,j)=1} ((\theta^{(j)})^T (x^{(i)}) - y^{(i,j)})^2 + \frac{\lambda}{2} \sum_{k=1}^{n} (x_k^{(i)})^2 \right)}$$

In order to learn the features for all books i, that is $x^{(1)}, x^{(2)}, ..., x^{(n_b)}$, we need to solve:

$$\boxed{\min_{x^{(1)}, x^{(2)}, ..., x^{(n_b)}} \left(\frac{1}{2} \sum_{i=1}^{n_b} \sum_{j:r(i,j)=1} ((\theta^{(j)})^T (x^{(i)}) - y^{(i,j)})^2 + \frac{\lambda}{2} \sum_{i=1}^{n_b} \sum_{k=1}^{n} (x_k^{(i)})^2 \right)}$$

If we are using a gradient based optimization algorithm to solve for the above problem, then the derivative of the cost function here should be with respect to $x^{(i)}$, however it should not look too different from before and working the differentiation is left as an exercise to the reader.

So once we solve the above problem and get a good approximation of $x^{(1)}, x^{(2)}, ..., x^{(n_b)}$, there is no reason why we can't use this new information to estimate $\theta^{(1)}, \theta^{(2)}, ..., \theta^{(n_u)}$ - as we might not have all of them, or there may be new users over time. But there may also be new books over time, and we can go back and forth in this manner estimating features and parameters.

If you study the two minimization problems below, you will see that it is possible to combine them into one:

In order to learn the features for all books i, that is $x^{(1)}, x^{(2)}, ..., x^{(n_b)}$, we need to solve:

$$\min_{x^{(1)}, x^{(2)}, ..., x^{(n_b)}} \left(\frac{1}{2} \sum_{i=1}^{n_b} \sum_{j:r(i,j)=1} ((\theta^{(j)})^T (x^{(i)}) - y^{(i,j)})^2 + \frac{\lambda}{2} \sum_{i=1}^{n_b} \sum_{k=1}^{n} (x_k^{(i)})^2 \right)$$

In order to learn the parameters for all users j, that is $\theta^{(1)}, \theta^{(2)}, ..., \theta^{(n_u)}$, we need to solve:

$$\min_{\theta^{(1)}, \theta^{(2)}, ..., \theta^{(n_u)}} \left(\frac{1}{2} \sum_{j=1}^{n_u} \sum_{i:r(i,j)=1} ((\theta^{(j)})^T (x^{(i)}) - y^{(i,j)})^2 + \frac{\lambda}{2} \sum_{j=1}^{n_u} \sum_{k=1}^{n} (\theta_k^{(j)})^2 \right)$$

... combined, becomes:

$$\min_{x^{(1)}, x^{(2)}, ..., x^{(n_b)}, \theta^{(1)}, \theta^{(2)}, ..., \theta^{(n_u)}} \left(\frac{1}{2} \sum_{(i,j):r(i,j)=1} ((\theta^{(j)})^T (x^{(i)}) - y^{(i,j)})^2 + \frac{\lambda}{2} \sum_{i=1}^{n_b} \sum_{k=1}^{n} (x_k^{(i)})^2 + \frac{\lambda}{2} \sum_{j=1}^{n_u} \sum_{k=1}^{n} (\theta_k^{(j)})^2 \right)$$

So to implement collaborative filtering:

1. Initialize $x^{(1)}, x^{(2)}, ..., x^{(n_b)}, \theta^{(1)}, \theta^{(2)}, ..., \theta^{(n_u)}$ to small random numbers.
2. Minimize the cost function

   $$C(x^{(1)}, x^{(2)}, ..., x^{(n_b)}, \theta^{(1)}, \theta^{(2)}, ..., \theta^{(n_u)}) = \frac{1}{2} \sum_{(i,j):r(i,j)=1} ((\theta^{(j)})^T (x^{(i)}) - y^{(i,j)})^2 + \frac{\lambda}{2} \sum_{i=1}^{n_b} \sum_{k=1}^{n} (x_k^{(i)})^2 + \frac{\lambda}{2} \sum_{j=1}^{n_u} \sum_{k=1}^{n} (\theta_k^{(j)})^2$$

   using gradient descent or another optimization algorithm.
3. For a user j with parameters $\theta^{(j)}$ and for a book i with features $x^{(i)}$, predict a rating of $(\theta^{(j)})^T (x^{(i)})$.

The algorithm just described here is also known as *low-rank matrix factorization*, due to the vectorized implementation of the predicted ratings $X\Theta^T$ being a matrix of low-rank.

We can use the results to find the top five similar books to some book i as follows: find the five books l with the smallest distance from book i, that is $||x^{(i)} - x^{(l)}||$.

Lecture 10

Activity and Conclusion

Note: The activity will be available in class only, during the last class.