

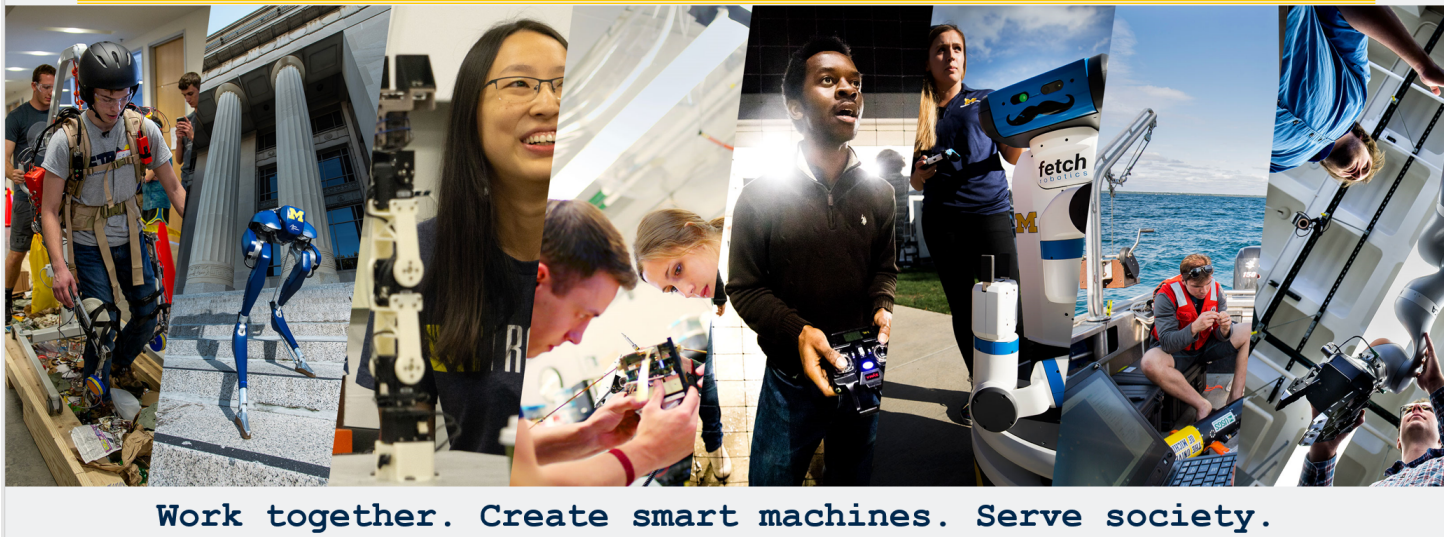
Computational Linear Algebra

Math and Programming
at the Scale of Life

Author
Jessy Grizzle, Director

Lab Manual for Julia Instruction

Inspiration
Chad Jenkins, Associate Director of Undergraduate Programs



Cover design by Dan Newman, Head of Communications, Michigan Robotics

© 2022 Jessy Grizzle, Professor of Robotics, University of Michigan
JERRY W. AND CAROL L. LEVIN PROFESSOR OF ENGINEERING
ELMER G. GILBERT DISTINGUISHED UNIVERSITY PROFESSOR

This manual is based on the Julia lab exercises that were developed in F-21 by Lu Gan, John Pye, and Jessy Grizzle. Over Su-22, Ms. Yining (Maya) Yuan contributed numerous improvements to the exercises and the editing of the manual.

First release, August 31, 2022. Second Release, January 4, 2023.

Contents

Preface	7
Philosophy of the Course	9
0 Julia Lab 0: Introduction to Coding in Julia	11
0.1 Welcome to ROB 101 and the Julia Programming Language!	13
0.2 Vocabulary	14
0.3 Accessing and Submitting Julia Assignments	15
0.4 Learning and Keeping Track of Programming Skills	15
0.5 LAB 0: Julia as a Calculator	16
0.5.1 Rounding Error	16
0.5.2 Assignment Operator is Denoted with an Equals Sign	18
0.5.3 Arithmetic Operations and PEMDAS	19
0.5.4 Reading an Error Message	20
0.6 Standard Built-in Functions	20
0.7 How to “Reboot” your Jupyter Notebook	22
0.8 Pro Tip: Hit the Save Button!	24
0.9 Julia is not Working. What can I do?	24
0.10 My Submit Button has Disappeared! What do I do?	24
0.11 (Optional Read) Additional Julia Resources	25
0.12 (Optional Read) Why was Julia Created?	25
1 Julia Lab 1: Variable Types, Plotting a Function, and Creating Arrays	27
1.1 Variable Types	28
1.2 Creating Simple Functions	29
1.3 Very Basic Plotting	30
1.4 Creating Arrays	33
1.5 Commenting out Lines of Code or Providing Explanations	35
1.6 Applying Functions to Arrays via Broadcasting	36
1.7 Debugging	38
2 Julia Lab 2: Vectors, Matrices, and Indexing	41
2.1 Friendly Checks	42
2.2 New Ways to Create Matrices	44
2.2.1 Using Built-in Functions	44
2.2.2 Concatenation or Adding Columns and Rows to Matrices	46
2.3 Indexing or Slicing Vectors	48
2.4 Indexing or Slicing Matrices	50
2.5 More Advanced Remarks on Indexing	51
2.6 Advanced Methods for Creating Vectors and Matrices	54
2.7 Length of a Vector and Size of a Matrix	57
2.8 Debugging	59
2.8.1 Errors with the <code>size</code> command	59
2.8.2 Bounds errors due to improper indexing	60

3	Julia Lab 3: For Loops and 1-Element Vectors	63
3.1	Basic For Loops	64
3.2	Incrementing the Counter by Something other than 1	65
3.3	Summation Symbol in Math Equals a For Loop in Programming	67
3.4	A Brief Intro to Matrix-Vector Multiplication and the Unnerving Fact that 1-Element Vectors in Julia are not Scalars	68
3.5	Back Substitution as a Backward For Loop	70
3.6	Hidden Variables in For Loops (aka Scope of a Variable)	75
3.7	Debugging	77
3.7.1	Dimension Mismatch	77
3.7.2	More on 1×1 Matrices are not Numbers	79
3.7.3	Hidden Variables	81
3.8	(Optional Read) While Loops and an Open Problem in Mathematics	82
3.9	(Optional Read) Double For Loops	84
3.10	(Optional Read) A Less Brief Intro to Matrix-Vector Multiplication and the Unnerving Fact that 1×1 Matrices and 1-Element Vectors in Julia are not Scalars	85
4	Julia Lab 4: If Statements, Function Creation, and “Peeling the Onion”	89
4.1	If Statements aka Conditional Statements	90
4.2	Writing Better Functions	92
4.3	A Function for Multiplying Matrices	94
4.4	Forward Substitution	95
4.5	Peeling the Onion: the Base Step for LU Factorization	96
4.6	Debugging	101
4.6.1	Misusing operations designed for scalar variables	101
4.6.2	Bounds Error	102
4.6.3	Using the <code>display()</code> command to find an error buried in a function	104
4.7	(Optional Read) “Hero 60 AD” or Heron’s Formula for the Area of a Triangle	106
4.8	(Optional Read) More Examples Combining Functions and Flow Control	109
5	Julia Lab 5: Linear Independence and LDLT Factorization, a Souped-up Version of LU	113
5.1	Checking Linear Independence with LU	114
5.2	LU is a Suboptimal Tool for Determining the Number of Linearly Independent Vectors in a Set	116
5.3	LDLT Factorization: A One-shot Means to Find Linearly Independent Vectors in a Set	119
5.4	Debugging	123
5.5	(Optional Read) Finding Counterexamples via Search	127
6	Julia Lab 6: Matrix Null Space and Linear Regression	129
6.1	Null Space of a Matrix	130
6.2	Linear Regression or Least Squares Fit to Data	134
6.3	Debugging	139
6.4	(Optional Read) Fitting a Surface to Data	145
6.5	(Optional Read) Lab 6.5 A Potpourri of Interesting Commands	150
6.5.1	Using the Benchmark Tool: Which is Faster for Solving $Ax=b$?	150
6.5.2	Printing with Style and Macros @	152
6.5.3	Multiple Dispatch and Function Overloading	153
7	Julia Lab 7: Gram-Schmidt Algorithm, Orthogonal (Basis) Vectors, and Computing the Null Space	157
7.1	Orthogonal Vectors	158
7.2	The Span Condition	160
7.3	Graphical Interpretation of Orthogonal Vectors and Another way to Understand the Span Condition	161
7.4	Gram-Schmidt Algorithm on a Set of four Vectors	164
7.5	Gram-Schmidt Algorithm for a General Number of Linearly Independent Vectors	166
7.6	Gram-Schmidt Algorithm without Assuming Linear Independence of the Starting Vectors	168
7.7	Orthogonal Complement of a Set of Vectors	170
7.8	Null Space of a Matrix using the Orthogonal Complement	172
7.9	Debugging: the Worst Case is When We Have No Error Messages	173
7.10	(Optional Read) Comparing Julia’s Native Null Space Command with Our Own	179

7.11 (Optional Read) Null Space of a Matrix without Mentioning the Orthogonal Complement	180
8 Julia Lab 8: Basis Vectors, Dimension, Coordinates, Eigenvectors and Eigenvalues	185
8.1 Basis Vectors, Linear Independence, and Span	186
8.2 Vector Space Coordinates and Vector Representations	189
8.3 Eigenvectors and Eigenvalues	193
9 Julia Lab 9: Modeling, Simulating, and Controlling a Mobile Robot	201
9.1 Motivation	202
9.2 What is an ODE?	203
9.3 Euler’s Method for Numerically Solving ODEs	204
9.4 Discrete-time Approximations of Continuous-time Linear ODEs with Control Variables	209
9.5 Predicting the State Vector at Future Instants of Time	213
9.6 (Optional Read:) For What Values of N are the Rows of M Linearly Independent?	216
9.7 Feedback Control of Discrete-time Linear Models via Least Squares for Underdetermined Systems of Equations	217
9.7.1 Getting the Pendulum Upright is not Enough	217
9.7.2 Keeping the Pendulum Upright is Our Goal	218
9.7.3 Relevant Code for MPC	219
9.7.4 Balancing the Double Inverted Pendulum with MPC	221
10 Julia Lab 10: The Joy of Doing Calculus with Julia!	223
10.1 Symbolic Differentiation	224
10.2 Numerical Differentiation	227
10.3 Automatic Differentiation	229
10.4 Which one is fastest? And by How Much?	231
A Summary of Key Julia Commands	233
A.1 Creating Vectors and Matrices	233
A.2 Indexing or Slicing Vectors and Matrices	235
A.3 1-Element Vectors and 1 x 1 Matrices	237
A.4 Size vs Length Commands, Which to use for Vectors and Matrices	238
A.5 Flow Control: If-then-else and Looping	241
A.6 Functions	243
A.7 Applying Functions to Arrays via Broadcasting	244
A.8 Plotting	245
A.9 Suggested Resources	247
B Building Better Functions Through Structured Returns or Named Tuples	249
B.1 Building a Structured Return of Your Own	250
B.2 Your First Named Tuple, an Alternative to Structured Returns	254
B.3 Tools to Solve Linear Equations	259
C From MATLAB to Julia	267
D From C++ to Julia	269
E Creating Your Own Local Julia Installation	271

Preface

This Julia lab manual goes with the notes for ROB 101, Computational Linear Algebra. Julia is used to transform the theory learned in the course into effective algorithms, or code! The course assumes no prior experience with programming. For those with prior knowledge of MATLAB or C++, Appendices C and D provide hints to help you acquire more quickly some fluency in Julia. A summary of the key Julia commands used in the course is given in Appendix A.

Philosophy of the Course

Please see the ROB 101 textbook, which is available online, at no cost <https://github.com/michiganrobotics/rob101/tree/main/Fall%202021>.

Jessy Grizzle Ann Arbor, Michigan USA

Chapter 0

Julia Lab 0: Introduction to Coding in Julia

Learning Objectives

- What is coding?
- Getting started with coding at the level of a calculator.

Outcomes

- The Good, The Bad, and the Ugly about Julia
- What is rounding error
- Assigning variables
- Basic arithmetic
- Our first dive into an error message
- A list of built-in functions
- Rebooting: what to do when all else fails
- How to learn and keep track of programming skills

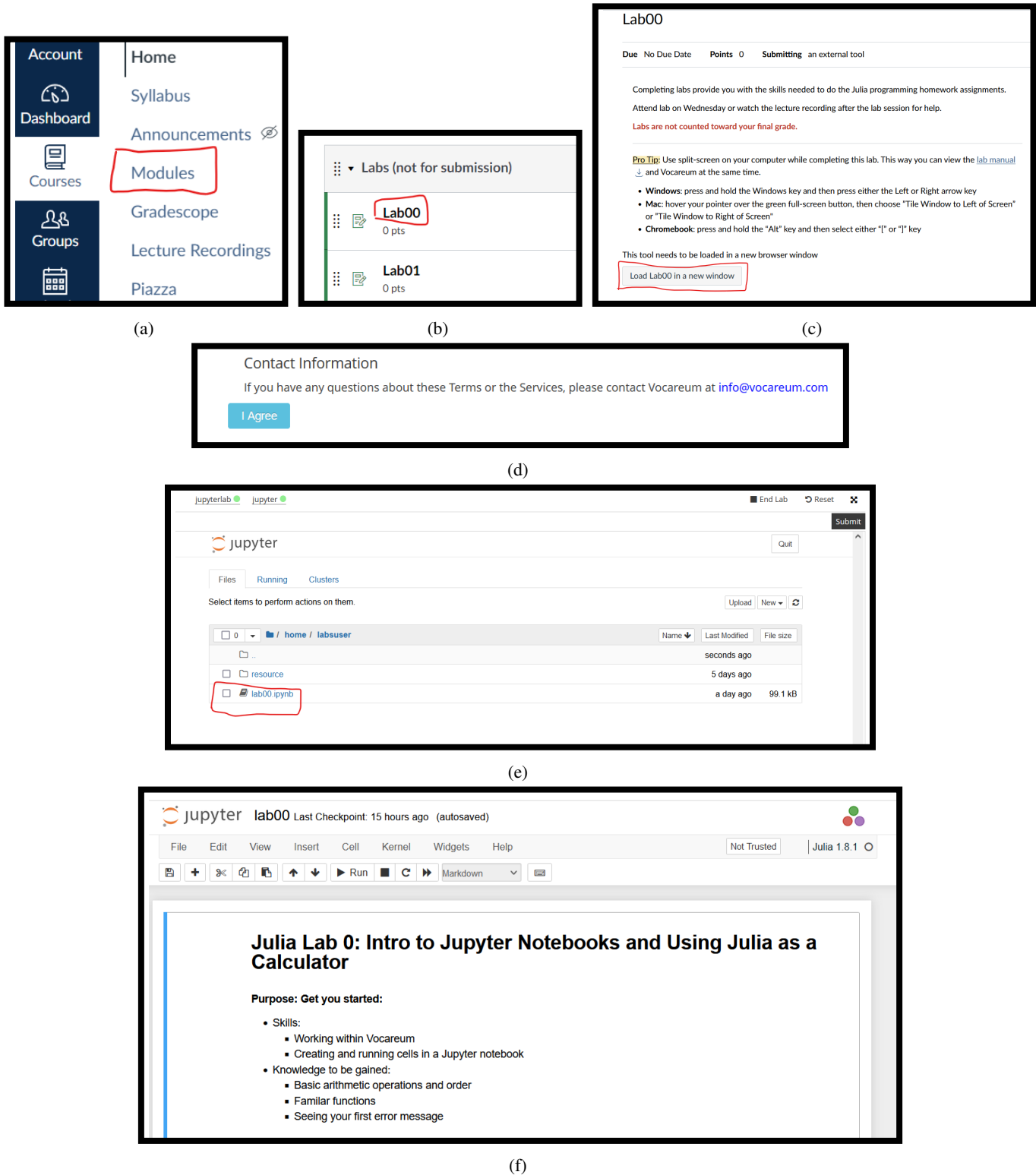


Figure 1: Sequence of steps for accessing your Julia assignments. (a) You start in Canvas, where you click the Modules link on the left hand side of the page. (b) If necessary, scroll down the page until you see the Labs tab or the assignment for the appropriate week. Here we will click on Lab00. (c) Fetch the assignment by clicking on the button that says, Load Lab00 in a new window. (d) Accept the terms of service for Vocareum. You only have to do this once in the term. (e) Click on the name of the assignment. **For Labs, you never use the submit button. Labs are not graded. Labs are meant to help you with the Julia HW sets and the Projects, which are graded.** (f) View of a Jupyter notebook.

Remark 1 *Hitting the save icon and/or Cntrl-S is completely different than hitting the submit button. You can submit as many times as you want. The last submission is graded. For Julia HW only, if you forget to submit, we automatically grab your latest SAVED copy at 11:59 pm, the day it is due. We do NOT automatically collect your Projects. Why? Because we allow late submissions with Projects. If you submit your Project on time and then hit the submit button some other time after the due date, your last submission will be graded and will receive a late penalty.*

0.1 Welcome to ROB 101 and the Julia Programming Language!

ROB 101 assumes no prior experience with Calculus or programming. These notes are meant for the novice programmer. Learning to program can be fun and frustrating at the same time. Our goal with these notes is to increase the amount of time you have fun and reduce the frustration to the minimum necessary for learning to take place. We all had to touch a hot stove once in our lives!

If you took programming in High School (HS) or you've already taken ENGR 101 at Michigan, then you have a head start, perhaps. I say perhaps, because Julia is similar to MATLAB, and being a pretty good MATLAB programmer, I expected to pick up Julia in a flash. Wrong! There are just enough maddening differences that I was super frustrated for the first month to six weeks. The same will be true of those of you who know C++. To help you out, in the Appendix, you will find helpful hints to speed your transition into Julia. You can find many sites like this one for additional help <https://github.com/brenhinkeller/JuliaAdviceForMatlabProgrammers>. When you find even better sites, please let us know.

If you did not take programming in High School, these notes are meant for you! You'll find that learning math and programming at the same time is a much richer and more meaningful experience than learning math alone. My colleague Prof Chad Jenkins coined the phrase, "Programming = Believing", and I think he is right. Our goal is to turn into code almost every equation we see in lecture. Things we would never ever dream of doing by hand become "cake" in Julia.

Why are we using Julia? For several reasons:

- Robotics is working alongside Historically Black Colleges and Universities (HBCUs) https://en.wikipedia.org/wiki/Historically_black_colleges_and_universities, and eventually other Minority Serving Institutions (MSIs), to offer Robotics instruction either remotely or in house. To support remote instruction, we wanted a programming language that we could deliver via a browser. While most students at engineering-intensive universities can download MATLAB to a laptop for free, that is not true at most institutions of higher learning.
- Julia is open source and hence free to everyone! Robotics is a huge supporter of the open-source movement. Even as a Michigan student with free access to MATLAB, when you leave Michigan, there is no guarantee that your company will have MATLAB because it is quite expensive (multiple thousands of dollars per user (seat)). Because Julia is free, cost is not a limiting factor for its adoption.
- Julia is blazingly fast. When you run a block of code for the first time, it is compiled. Thereafter, you essentially have the speed associated with C++.
- Julia is modern and forward looking. As someone who was taught FORTRAN back in the day, it feels good to be sharing with you a language that is likely to stand a significant test of time.

Why should we have chosen anything but Julia for ROB 101? The error messages in Julia are horrible. These will be fixed in Julia 2.0. The most current release is Julia 1.8 and thus we'll have to teach you how to suffer through the error messages and find your mistakes. A second reason would be that Julia is *strongly Typed*, which is the proverbial "blessing" and a "curse".

For the super nerds, you must know about the official website for the Julia programming language <https://julialang.org/>. It's promotion of Julia is designed for professional programmers. There is a blog that is kind of fun <https://julialang.org/blog/2022/02/10years/> where a group of Julia founders and pioneers look back on the language.

All of the Julia work for ROB 101 can be completed in the cloud via a browser. **You do not need to install Julia on your personal computer! You can complete the assignments with as little as an iPad and a keyboard.** Nevertheless, some of you may want to have your own Julia installation; see Appendix E for how to do that.

Warning

If you pursue your own installation, **we cannot provide IT support. We do not have the person-power to do it. Moreover, you cannot submit HW or Project assignments from your personal Julia installation.** While you can transfer files from our web-based Julia installation to a personal installation, do the work there, and then copy your work **cell by cell** back to the web-based environment, you may find that very tedious.

Further Warning: If you change the name of a web-based assignment, it will not be processed by the autograder and you will score a ZERO on your assignment.

You might, however, create your own installation so that you can work on Julia when you are “off the grid”, for example, or because “you like the feel of being in control of your computing environment”. Your author uses both: the class site and a personal installation.

0.2 Vocabulary

- **Programming** is the act of writing instructions for a computer to follow. **Coding** is similar to programming...if there is a difference, it would be that you do not “code” a computer, you “program” it, while the action of writing or composing computer instructions can be called either “coding” or “programming”.
- **Syntax** is the set of formal rules that govern a language, whether a human language such as Spanish or English, or a computer language, such as Julia or C++. Syntax prescribes the allowed symbols in a language, how they can be combined/ordered, and the role of punctuation. Where a pair of square brackets [] is used in Julia to “index” or “slice” an array of numbers, in MATLAB, one uses instead parentheses (). These differences can be maddening, just like “false friends” between English and French, such as the word “special” having a positive connotation in English and “speciale” having a negative connotation in French. How can that be? Or French inserting a space before a question mark at the end of a sentence, while in English, we do not. The challenge with learning a computer language is that it is **unforgiving of syntax errors**. If you are lucky, it “throws an error flag” and identifies the line on which the error occurred. If you are unlucky, what you entered was still legal syntax and the computer does something you never intended, without you suspecting that your instruction actually meant something else.
- A list of **punctuation symbols** in Julia <https://docs.julialang.org/en/v1/base/punctuation/>. We will only encounter a few of them in ROB 101.
- **Debugging** computer code is the process of finding and fixing errors in the code. It could be that the code does not run because the errors were “fatal” or the code runs but produces flawed output. You can read more here about the etymology of the term debugging <https://en.wikipedia.org/wiki/Debugging>.
- **Variables** store information that can be referenced or changed in a program; see [https://en.wikipedia.org/wiki/Variable_\(computer_science\)](https://en.wikipedia.org/wiki/Variable_(computer_science)).
- A variable’s **Type** determines the values it can hold and the operations that can be performed on the variable. If a variable’s type is `char` then it can hold characters, such as letters of the alphabet, or common symbols such as `@` or `&`. If its type is `char` and you try to assign a real number to it, you will generate an error. In the computer, a variable’s type is used to economize on storage space, because in general it takes more zeros and ones (**bits**) to specify a real number than it does a letter of the alphabet. Hence allocating a ton of space in memory and then putting the letter D in it is wasteful. Types are also used to indicate what operations can be performed on a variable. For example, letters in the alphabet (variables of `char` type) can be concatenated to form `strings`, whereas it does not make so much sense to concatenate 3.14159 with 1.41421, because what would you do with an object that had two decimal points? Similarly, it makes sense to add two real numbers (type `Real`), but not so much the addition of two letters A and M. By assigning `Types` to all variables, Julia helps you to avoid “stupid mistakes”. We’ll also see that Julia can sometimes be so strict with its imposition of `Types` that it’s maddening. More on this later.
- **Jupyter Notebook** (spelling is correct) “is an open-source web application that allows data scientists to create and share documents that integrate live code, equations, computational output, visualizations, and other multimedia resources, along with explanatory text in a single document¹.” Jupyter is sort of a contraction built from the programming languages Julia, Python, and R. Jupyter Notebooks consist of a sequence of cells, where each cell can hold code, text, plots, typeset equations, images, etc.

¹<https://odsc.medium.com/why-you-should-be-using-jupyter-notebooks-ea2e568c59f2>

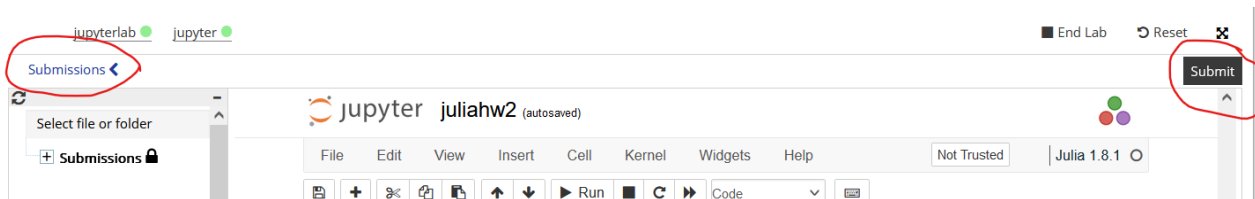
- **Vocareum** is the company supporting our web access to Julia and Jupyter notebooks; you can learn more at <https://www.vocareum.com/>.
- A more extensive **glossary of coding terminology** is available here <https://code.org/curriculum/docs/k-5/glossary>.

0.3 Accessing and Submitting Julia Assignments

Accessing Labs, HWs, and Projects All of the Jupyter notebooks used in the course are accessed via the ROB 101 Canvas page; hence you need to be a member of the course’s Canvas site to reach it. Once you login to Canvas, you want to

- Click on the Modules link in the left hand panel.
- Scroll down until you see what you are after: a Lab, a Julia HW, or a Project. They are mostly listed by the week that an assignment becomes available to you, though Labs can be accessed at any time.
- Click on the appropriate link.
- Hit the tan box that says, **Load XXXX in a new window**.
- Click on the appropriate link.
- You should now be in a Jupyter notebook; see Fig. 1.

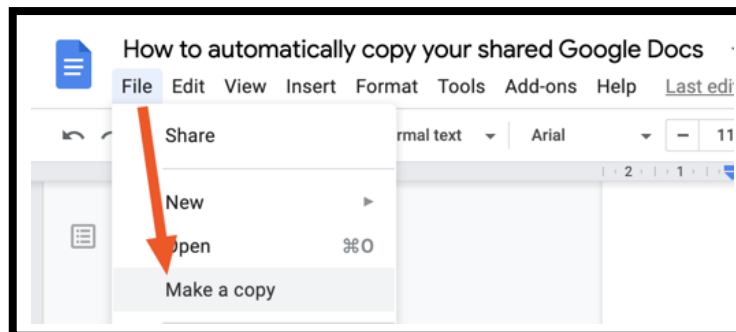
Julia HW Submission: While Labs are not submitted, you should submit your Julia HW and your Projects. You do that by hitting the **BLACK** submit button in the Upper Right corner. You can see all of your submissions by hitting the **Submission >** link in the Upper Left corner. After you hit the link, the arrow will change directions **Submission <**.



0.4 Learning and Keeping Track of Programming Skills

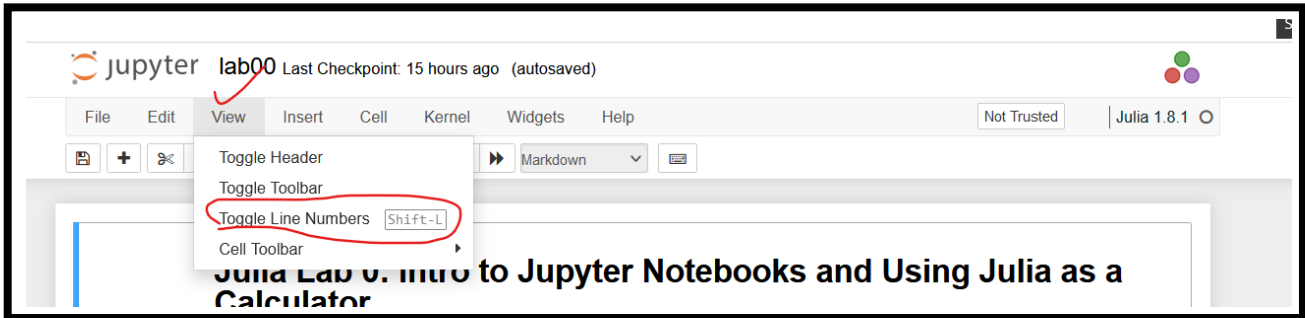
While we have summarized the most frequently used commands in Appendix A, you will learn more quickly if you create your own record of frequently used commands and encountered problems. It is suggested that you make a copy of the following Google Doc https://docs.google.com/document/d/1NnB0dFUKbLAOJ8CIB9YpXj71XhAPFwmTAYOFDDHR_w/edit?usp=sharing and add commands to it as you learn them in the course. You must make your own personal copy before you can edit the file. Once you copy the file, you can rename it and add content.

Can a group of you share a Google Doc filled with Julia tips and tricks? Sure. It’s OK, but you’ll probably learn more if you make your own “cheat sheet”.



0.5 LAB 0: Julia as a Calculator

This lab gets you started with Julia. You should follow the steps in Chapter 0.3 and open the lab00 Julia notebook, or in your own installation of Julia, open a Julia notebook and try the various basic commands introduced here.



Remark 0.1 (Enable Line numbering) *View > Toggle Line Numbers.* In labs and on homework assignments, you will come across instructions which tell you to alter certain lines of your code. You will also need to debug your code based on the errors reported by Julia, which will tell you the line number at which your program failed.

If you are in a Jupyter notebook and want to **create a new cell**, hit the plus sign **+** in the top banner. The cell will open below the cell in which you last clicked with your cursor (play around with it). This works for HW, LABs, and empty Jupyter notebooks. If you want to try something out, such as rewriting a piece of code without erasing your current code, just open a new cell below the current one, copy and paste in whatever you wish and go to work. To **remove a cell**, click on it with your cursor, and then click on the scissor symbol **✂**, which is right next to the plus sign **+**. To **run a cell**, you hit the run **▶** Run button on the top banner. On a PC, you can hold **Shift** **Enter**. It is worth getting to know the family of buttons on the top banner. Experiment a little! Google a little.



0.5.1 Rounding Error

```
1 #in a code cell, julia will compute and return operations you do inside it
2 (1 + 2 + 3 + 4 + 5 + 6)/2
3 #here is some basic addition followed by division
```

Output

10.5

Computers do arithmetic with a **finite number** of zeros and ones. That's easy to accept. Some of the consequences of doing arithmetic with zeros and ones may be hard to accept when you first start programming. Let's take a look, because **you probably never noticed what is generally called "rounding error" when you used a calculator.**

```
1 # in a code cell, julia will compute and return operations you do inside it
2 4+7.62
3 # basic addition
```

Output

11.620000000000001

Wait a minute, we're pretty good at arithmetic, "Julia²" got the wrong answer! It's off by $0.000000000000001 = 1e-15$ or 1×10^{-15} . In fact, the actual error is

```
1 # rounding error
2 (4+7.62) - 11.62
```

Output

1.7763568394002505e-15

You don't need to understand the source of the error to be successful in ROB 101, but you do need to accept that it occurs.

(Optional read:) *In case you are interested, the **rounding error** comes from the fact that digital computers do calculations in base 2 using a finite number of zeros and ones.*

- By way of background, you are comfortable with the fraction $\frac{1}{3}$ having the infinite decimal expansion $0.3333333\dots$. This means that

$$\frac{1}{3} = \sum_{k=1}^{\infty} \frac{3}{10^k} = \frac{3}{10} + \frac{3}{100} + \frac{3}{1000} + \frac{3}{10000} + \frac{3}{100000} + \dots,$$

where the \dots means it goes on forever! Hence, it is not possible to represent the number $\frac{1}{3}$ in decimal notation using a finite number of digits between zero and nine.

- In base 3, however, the decimal fraction $\frac{1}{3}$ has the finite expansion 0.1 , while $\frac{1}{3} + \frac{2}{3^2} = \frac{5}{9} = 0.555555$ in base 10 has the base 3 expansion 0.12 . Finally, $\frac{1}{3} + \frac{2}{3^2} + \frac{1}{3^3} = \frac{16}{27} = 0.592592592$ in base 10, but has the base 3 expansion 0.121 .
- If you have forgotten or never mastered representing numbers in different bases, then don't worry about it. You can simply accept that a number having a finite (exact) expansion in base 10 does not mean it has a finite (exact) expansion in base 2, the number system used in digital computers.
- The number 7.62 requires three digits to express in base 10 (decimal numbers), but it takes an *infinite number of digits* in base 2 (binary numbers). Julia is using something like 111.10011110101110000101 as the binary approximation of 7.62 , which is roughly 7.61999988555908203125 in decimal, and is "very close" to 7.62 , but not equal to it. Indeed,

$$\begin{aligned} \underbrace{111}_7 \cdot \underbrace{.10011110101110000101}_{\approx 0.62} &= \overbrace{1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0}^{4+2+1=7} + \overbrace{1 \times 2^{-1} + 0 \times 2^{-2} + 0 \times 2^{-3} + 1 \times 2^{-4} + 1 \times 2^{-5} + 1 \times 2^{-6} + 1 \times 2^{-7} + 0 \times 2^{-8} + 1 \times 2^{-9} + 0 \times 2^{-10} + 1 \times 2^{-11} + 1 \times 2^{-12} + 1 \times 2^{-13} + 0 \times 2^{-14} + 0 \times 2^{-15} + 0 \times 2^{-16} + 0 \times 2^{-17} + 1 \times 2^{-18} + 0 \times 2^{-19} + 1 \times 2^{-20}}_{\text{fraction starts here}} \\ &\approx 7.61999988555908203125 \\ &= 7 \times 10^0 + 6 \times 10^{-1} + 1 \times 10^{-2} + 9 \times 10^{-3} + 9 \times 10^{-4} + 9 \times 10^{-5} + \\ &\quad \vdots \\ &\quad 2 \times 10^{-15} + 0 \times 10^{-16} + 3 \times 10^{-17} + 1 \times 10^{-18} + 2 \times 10^{-19} + 5 \times 10^{-20}, \end{aligned}$$

where the symbol \approx means "approximately equal to". This explains the error, in case you care. Understanding "rounding error" is not important for ROB 101, but eventually, it becomes important for all students of STEM.

- If a number has a finite representation in either base 2 or base 5, then it will have a finite representation in base 10, simply because 2 and 5 are factors of 10. For example,

$$\frac{1}{2^k} = \frac{5^k}{2^k 5^k} = \frac{5^k}{10^k}.$$

If you stare at this for awhile, you will convince yourself that if in binary, a number has k digits after the "decimal point" (binary point?), then in decimal, it will have at most k digits after the decimal point.

²The error has nothing to do with the Julia programming language and everything to do with the binary numbers used in a digital computer.

- Because 3 is not a factor of 10, there is no “reciprocity” on finite expansions, as we saw with the fraction $1/3$.

Notation 0.2 In case you are curious, the general name for the point “.” that separates the integer part of a number from the fractional part is **radix point** https://en.wikipedia.org/wiki/Decimal_separator. Not everyone uses points to indicate the separator. The French use a comma to separate the integer part of a number from the fractional part and a period to separate powers of ten. And they are not alone in this practice! Your author did a post-doc in Paris back in the 80’s, and had a checking account, of course. Getting the commas and periods correct was slightly important when I wrote a check.

Remark 0.3 This web page documents how fractions are handled in various programming languages, including Julia, Matlab, and C++, <https://0.3000000000000004.com/>.

0.5.2 Assignment Operator is Denoted with an Equals Sign

In mathematics, when we write $x + y + 2 = 3$, we mean the value of the sum of x , y , and 2 equals the number 3. We can “rearrange” the equation and write $x + y = 1$ or $x = 1 - y$ without changing what the equation means.

In programming, it is very important to understand that the symbol $=$ means **assignment**. For example, $x = (1 + 2 + 3 + 4 + 5 + 6) / 2$ **assigns** the value 10.5 to the variable x . You can now use x in calculations that follow it. It will hold the value 10.5 until you assign something else to it.

```
1 # assigning a value to x
2 x=(1 + 2 + 3 + 4 + 5 + 6) / 2
```

Output

10.5

```
1 # because we have already assigned x, we can use it in further calculations
2 y = 3*x
```

Output

31.5

```
1 # We can update x, using its past value
2 x = 3*x
```

Output

31.5

The assignment operator does not work like an equals sign.

```
1 x=x-0.5
```

Output

31.0

In “regular” arithmetic, we could subtract x from both sides of the equation $x = x - 0.5$ and arrive at the nonsensical equation, $0 = -0.5$. In the computer, you need to think of the left hand side of $x = x - 0.5$ as the new value that x will hold. When x appears on the right hand side as well, the computer will use the current value of x , which in our case, was 31.5, perform whatever operations you have specified, and store the new value in x . **In a programming language, $=$ is an assignment operation and not an equals sign.**

Remark 0.4 (Hindsight is 20/20, or the Equals Sign Blues) When that first computer programming language was written, there was probably no \leftarrow on the keyboard and hence the notation

$$x \leftarrow x + 3$$

was not adopted. The left arrow \leftarrow would make a perfectly good assignment operator. In fact, it has been adopted when writing pseudo code https://en.wikibooks.org/wiki/GCSE_Computer_Science/Pseudocode, which is an informal language, between a human language and a computer language, used for giving the key steps of an algorithm without committing to any particular coding paradigm.

0.5.3 Arithmetic Operations and PEMDAS

PEMDAS stands for **P**arentheses, **E**xponentials, **M**ultiplication & **D**ivision (left to right), **A**ddition and **S**ubtraction, the order in which Julia performs operations. While it is fine to memorize this, it is usually better to employ more parentheses than is necessary to make your code easily readable.

- Parentheses
- Exponentials (right to left)
- Multiplication and Division (left to right)
- Addition and Subtraction (left to right)

```
1 # The symbol ^ means ``raise to the power``  
2 10^(2 + 1)
```

Output

```
1000
```

```
1 # whereas, without parentheses  
2 10^2 + 1
```

Output

```
101
```

```
1 # you can take the power of variables  
2 x = 10 # just to be sure, we set x to a value  
3 x = x^(x^0.5)
```

Output

```
1453.0403018990432
```

Exercise 0.5 Place one (1) set of parentheses in the expression below to make w equal to 30

```
1 #place your parentheses here  
2 w = 2 + 1 + 2 + 7 * 3
```

Solution:

```
1 w = 2 + 1 + (2+7) * 3
```

Output

```
30.0
```

Remark 0.6 With no parentheses, the result of $w = 2 + 1 + 2 + 7 * 3$ is $w = 26$ because, Julia first does $7 * 3 = 21$ and then performs the addition left to right, $2 + 1 + 2 + 21 = 26$.

Exercise 0.7 Place two (2) sets of parentheses in the expression below to make z equal to 7

```
1 #place your parentheses here
2 z = 2 + 1 + 2 + 7 * 3 + 2 / 10 + 2
```

Solution:

```
1 z=2 + (1 + (2+7)*3 + 2)/10 + 2
```

Output

7.0

Remark 0.8 Adding parentheses to someone else's equation is no fun! Your instructor hates those kinds of problems. The take-home point is that parentheses allow you to control the order of computations. If you did not find the solution to the problem on your own, no worries.

Remark 0.9 With no parentheses, the result of $z = 2 + 1 + 2 + 7 * 3 + 2/10 + 2$ is $z = 28.2$ because, Julia first does $7 * 3 = 21$ and $2/10 = 0.2$, before then performing the addition left to right, $2 + 1 + 2 + 21 + 0.2 + 2 = 28.2$.

0.5.4 Reading an Error Message

You are going to make lots of errors. Let's make an error on purpose to see what happens. It is assumed that you have not yet defined a value to the variable r . If you have used r , replace it in the following with something else, like rr .

```
1 # A deliberate error
2 r = r+2
```

Output

UndefVarError: r not defined

Stacktrace:

```
[1] top-level scope
@ In[15]:2
[2] eval
@ ./boot.jl:360 [inlined]
[3] include_string(mapexpr::typeof(REPL.softscope), mod::Module, code::String,
filename::String)
@ Base ./loading.jl:1094
```

Julia reported that r is an **undefined variable** via the statement `UndefVarError: r not defined`. That's cool, you now know the name of the problematic variable and the nature of the problem. You were next told in which cell of your Jupyter notebook the error occurred, here it was input cell 15, and the line number of the cell in which the error occurred, here it was line 2. You can see all of this from the line `@ In[15]:2`. The other parts of the error message are helpful for a Julia professional, but not so much for us. Once again, the key parts of the error message are

Stacktrace:

[1] top-level scope

@ In[15]:2 ← gave the Jupyter Cell number and the line number in the cell where the error occurred. When you compose more complicated notebooks, the Cell number may be further up in your jupyter notebook, possibly where you define a new function. Julia's error messages can be called lot's of things, cryptic and arcane would be two of the nicer ones. Oh well! **Always look for the line that has In[some cell number]: some line number to get started on locating your error.** And of course, have line numbering turned on.

0.6 Standard Built-in Functions

As you can imagine, Julia has many built-in functions, more than a standard calculator will have. Here are a few,

- `abs()` #absolute value

- `sin()`, `cos()`, `tan()`, `asin()`, `acos()`, `atan()` #many more trigonometric functions
- `sqrt()` #square root
- `exp()` #natural exponential e^x
- `log()` #natural logarithm
- `round()` #rounds to nearest integer
- `ceil()` #rounds up to the nearest integer
- `floor()` #rounds down to the nearest integer
- `trunc()` #chops off anything beyond the decimal point

and so many more <https://docs.julialang.org/en/v1/manual/mathematical-operations/>. The key thing with a programming language is that you can expand the language by writing your own functions. In later labs, we will definitely build our own functions in ROB 101.

It is enough to scan the following. If you want to enter them into the lab0 notebook, feel free. Recall that LABS are not graded and you do not submit them to us. LABS are to help you learn Julia for use in the HW sets and the Projects.

```
1 cos(3*pi/4) #It assumes radians and not degrees
```

Output

```
-0.7071067811865475
```

```
1 0.5-cos(3*pi/4)^2 #In perfect math, the answer should be zero
```

Output

```
1.1102230246251565e-16
```

Remark 0.10 $1.1102230246251565e-16 \approx 2^{-53}$. Hence, in binary, that is 52 zeros after the “radix point” followed by a 1. That’s pretty close to zero!

```
1 exp(log(2)) # remember e ^ log(x) = x
```

Output

```
2.0
```

```
1 log10(10^3) # log base 10 instead of the natural log, which is base e
```

Output

```
3.0
```

```
1 round(1.5) #Goes to the nearest integer. Which way will 1.5 go? It's quite arbitrary.
```

Output

```
2.0
```

```
1 # ceil is short for ceiling, which is above you, typically,
2 # while floor is below you.
```

```
3
4 ceil(1.5) #note that ceil and floor always round "predictably"
```

Output

2.0

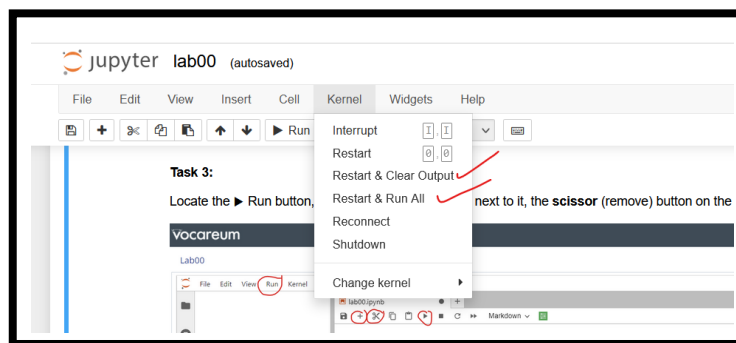
```
1 floor (1.5) #for instance, floor always returns a number less than or equal
2 #to the input argument, while
3 #ceil is greater or equal to the its input argument
```

Output

1.0

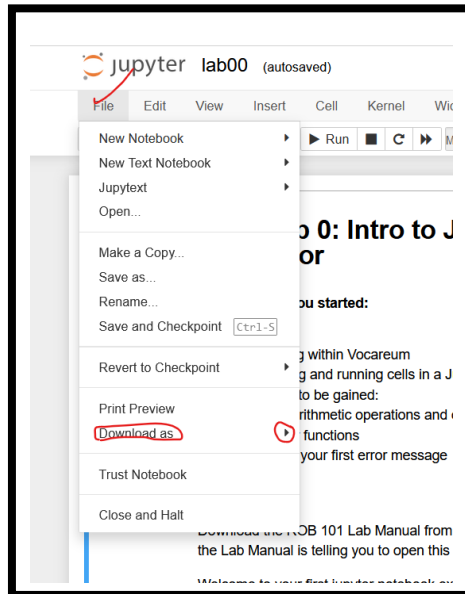
0.7 How to “Reboot” your Jupyter Notebook

From time to time, your jupyter notebook will freeze up or code will not run properly. In these cases, you need to do the equivalent of “rebooting your laptop”. Here is how to do it.

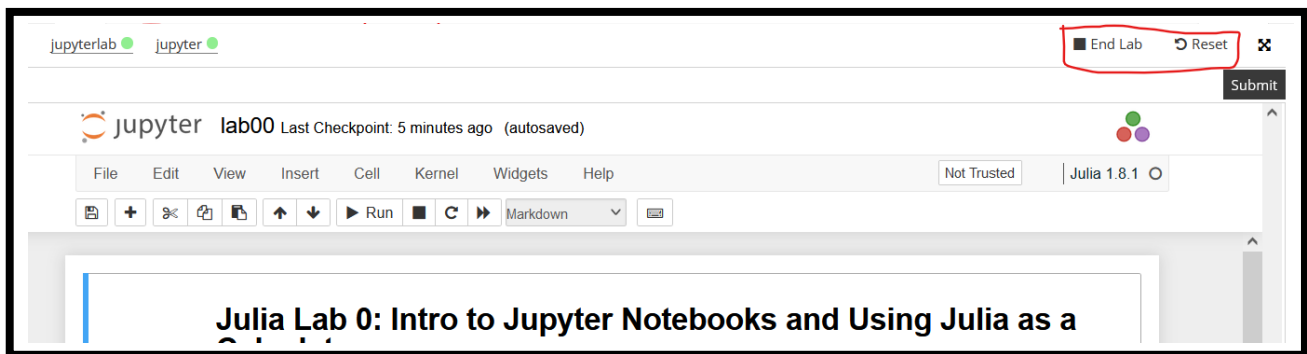


- Click on the Kernel tab, to get the above dropdown menu. Click on the Restart & Clear Output **or** Restart & Run All.
- **Neither of these erases any code in your Input Cells. Your work is safe!**
- Restart & Clear Output Stops and restarts the program that is running your notebook and clears all of the output cells. You then need to start from the top cell in your notebook (or other relevant location) and re-run each cell to redefine your variables because **the value of each variable has been “erased”**. Once again, all of your code is still there.
- Restart & Run All does all of the above and re-runs every cell in your notebook automatically (not a typo). You’ll have to experiment with these two options to see which one you like best.

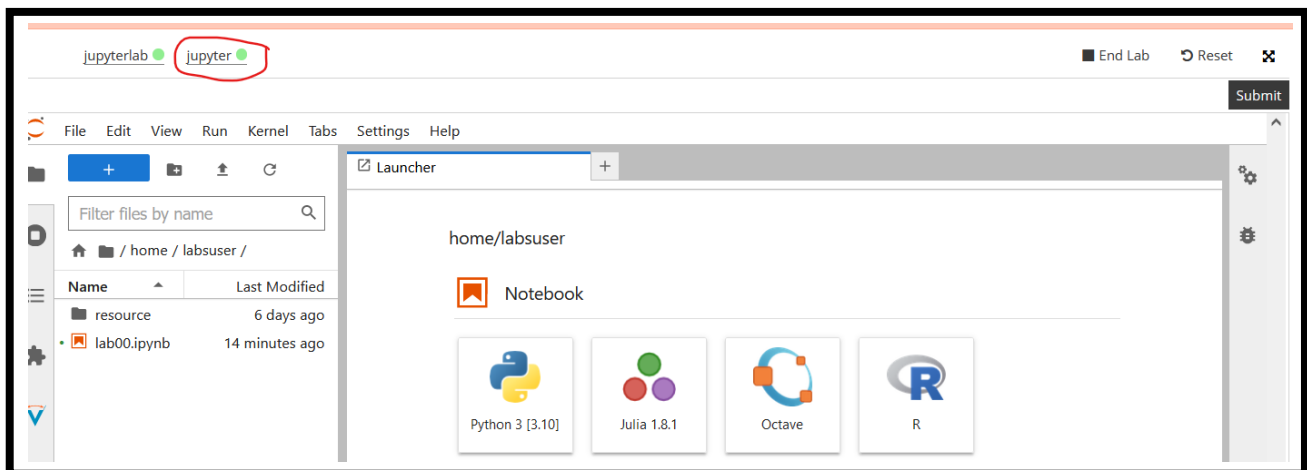
Sometimes the above is not enough and you need a “nuclear option”. The following will create a brand-new notebook, with all of your work erased. Yes. You have to start all over again! Step 1, if the system is not hung up too badly, is to hit the file tab, look for Download as, and select Save as notebook. If the system is hung up badly and you cannot download your current notebook, then in a different browser (e.g., if you are using Chrome, go to Edge) and access the ROB 101 site and fetch a fresh notebook.



Step 2 is to hit the **End Lab and Reset** buttons. Be patient. It can take some time to close all of the related processes. This will open up a fresh assignment, with any previous work **Completely Erased**. Copy and paste in relevant work from your previous file, if possible. **Personally, I do not close my previously opened file until I am sure that I do not need to copy any lines of code from it.** You'll arrive at your own file management etiquette!

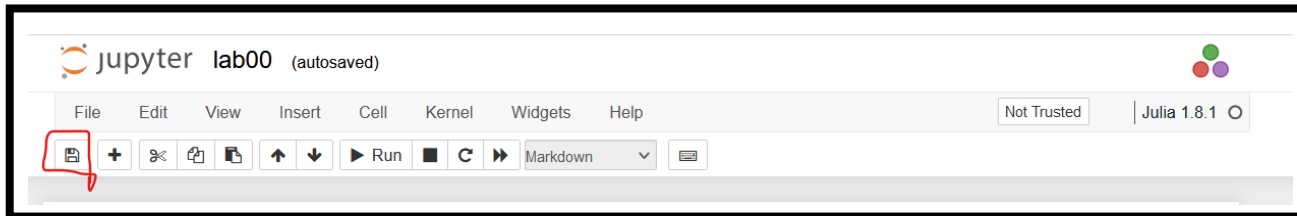


Sometimes, you may find yourself with a screen that is different than what you expect. Don't panic, just hit the `jupyter` button at the top and everything will return to normal!



0.8 Pro Tip: Hit the Save Button!

It's a good idea to ensure that your work is saved as you go. Your author hits the `CNTRL-S` key frequently on a Window's Machine. You can also use the `Disc` icon as shown below.



0.9 Julia is not Working. What can I do?

Here are some suggestions, in the recommended order of use.

- Hit the **End Lab** button in the upper right corner. That is supposed to give you a new session with your solutions intact, meaning, none of your coding is erased. You will have to re-run your cells as the results of your answers are erased. Once again, no work is lost. **Do not yet hit the reset button.**
- Next, check on Piazza to see if anyone else is having the same problem. A fix may be posted.
- If not, **a common reason that Julia stops working is that your Canvas login has expired.** Say you had juliahwXX open at 1 AM, go to bed, and start again at noon. Canvas is likely to have blocked your credentials without closing the open window/tab. When Canvas does that, all tools connected to Canvas are no longer accessible. The solution is
 - Keep the current copy of juliahwXX open.
 - Exit Canvas and log back in or just open up a fresh Canvas login to ROB 101 in a new tab, or better yet, a different browser (if you were in Chrome, try Edge or Firefox).
 - Re-open juliahwXX
 - Copy over any work that was not saved. Usually, that will be very little because you will have followed the Pro-Tip and hit the save button frequently!
- If the above does not work, please post on Canvas under help!help! and under Julia. The IAs and Instructors can post on the Vocareum forum that we need HELP and we need it NOW!
- It's late at night and you are desperate. No one is around to respond to your Piazza post. You are frustrated and all alone. It's good to try once again the **End Lab** button in the upper right corner. Wait two minutes and launch Canvas in a different browser, go to modules, invoke any magic incantations that have saved you in the past, maybe the one that got you accepted into Michigan, and hope for the best. When this fails, well, then, the "Nuclear Option" in Chapter 0.7 is likely your best bet. **It will erase all of your work. You will be given a pristine notebook.** User discretion is advised.

0.10 My Submit Button has Disappeared! What do I do?

This happens rarely, but it can be very disconcerting. Fortunately, there are a few easy things to try.

- If you can see the **stop lab** button at the top right, hit it. This will NOT erase your work. Wait a bit. You should be given a chance to re-open the notebook. Hit submit as normal. If that does not work...
- Close your HW or Project. Logout of Canvas. Log back into Canvas. Open your jupyter notebook. Normally, this does it, but if not ...
- Go to <https://labs.vocareum.com/main/main.php> and log in to your account on Vocareum using your email address. You will likely not know your password, click on **forgot password**, enter the email address being used by Canvas, and have Vocareum send you a password reset. After logging in, click on *ROB 101 Computational Linear Algebra*, click on the jupyter file in question, and then click the submit button as normal.

0.11 (Optional Read) Additional Julia Resources

- [Complete Julia Documentation](#)
- [Linear Algebra](#)
- [Noteworthy Differences from Matlab](#)
- [Noteworthy Differences from C++](#)
- [FAQ for Advanced Users of Julia](#)

0.12 (Optional Read) Why was Julia Created?

The following material is from https://juliadatascience.io/julia_accomplish: “We are greedy: we want more. We want a language that’s open source, with a liberal license. We want the speed of C with the dynamism of Ruby. We want a language that’s homoiconic, with true macros like Lisp, but with obvious, familiar mathematical notation like Matlab. We want something as usable for general programming as Python, as easy for statistics as R, as natural for string processing as Perl, as powerful for linear algebra as Matlab, as good at gluing programs together as the shell. Something that is dirt simple to learn, yet keeps the most serious hackers happy. We want it interactive and we want it compiled.”

You can find many more reasons [here](#).

Chapter 1

Julia Lab 1: Variable Types, Plotting a Function, and Creating Arrays

Learning Objectives

- Some basic computing constructs
- Moving beyond a graphical calculator
- Getting started with debugging code in Julia

Outcomes

- Types of variables in Julia
 - Int64
 - Float64
 - Char
 - String
- Building arrays
- Declaring and plotting functions
- Applying functions to arrays via broadcasting
- First appearance of n -element `Vector{Type}` and $n \times m$ `Matrix{Type}`
- Commenting out lines of code

Either download Lab1 from our Canvas site or open up a Jupyter notebook so that you can enter code as we go. It is suggested that you have line numbering toggled on.

1.1 Variable Types

A data **Type** or simply type is an attribute of data which tells the [computer] how the programmer intends to use the data https://en.wikipedia.org/wiki/Data_type". Computers have a limited amount of memory (zeros and ones) available to represent a number. If the computer knows that the number does not require a decimal point (as in -7, 0, 1, 45, etc.) then it can store it very compactly and hence store enormously large numbers of the `Int64` TYPE. If the number requires a decimal to express it properly (such as pi, sqrt(2), 0.6) then Julia will store it in a different format. Mostly, Julia is able to handle the memory needs of our code without our help, because we tell Julia in advance the memory needs of our variables.

Everything in Julia has a Type. We will mostly use variables with these 4 Type

- (a) `Int64`
- (b) `Float64`
- (c) `Char`
- (d) `String`

Int64 is used for integers or whole numbers, in other words, ...-3, -2, -1, 0, 1, 2, 3, ... The 64 refers to the number of 'bits', 0s and 1s, that are used to represent the number and its sign. The sign takes 1 bit, leaving 63 for the number. It is possible to represent numbers from -2^{63} to $2^{63} - 1$. These are enormous values.

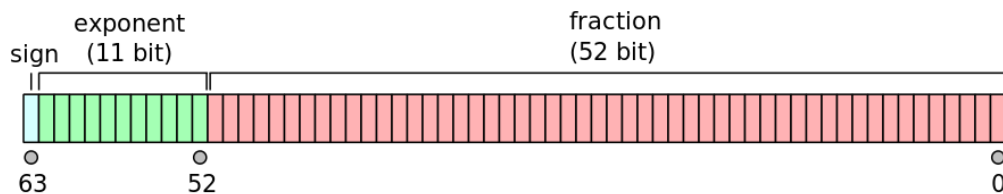


Figure 1.1: (Optional Read) Explanation from https://en.wikipedia.org/wiki/Double-precision_floating-point_format on how the bits used to represent a double-precision Floating Point number are organized in a computer.

Float64 Floating point is computer speak for numbers that require decimal points, such as 1.414. Figure 1.1 explains (optional read) why larger numbers can be stored in `Int64` than in `Float64`. We mostly do not care about this in ROB 101. We will primarily use floating point variables. But sometimes Julia really wants an integer variable and will throw a “Type Error” when you get it wrong. If you are not aware that all variables have a Type, then you will be at a loss to understand your mistake.

Char is used to store characters, such as letters, numbers, or symbols. If you dig into Julia, you can learn how to use Chinese characters, Hebrew characters, Greek letters, and more. You place characters between single ‘ ’ marks, such as `x = 'A'`.

String is used for a sequence of characters in a single line. This could be a word, or a sentence, or even a number that is placed between double quote marks “ ”, such as `myPlotTitle = "My #1 Plot in Julia"`.

Remark 1.1 Julia only prints out the last line in a cell that produces a result (it does not count a comment sign # as a result). To cause other values to print to the screen, use the `@show` command. To verify (or learn) the Type of a variable, use the `typeof` command.

```
1 # Run me
2 w = 33
3 x = sqrt(2)
4 y = 'A'
5 z = "Oh oh, what I have I done!"
```

Output

```
"Oh oh, what I have I done!"
```

```
1 # Run me
2 @show w = 33
3 @show x = sqrt(2)
4 @show y = 'A'
5 z = "Oh oh, what I have I done!"
```

Output

```
w = 33 = 33
x = sqrt(2) = 1.4142135623730951
y = 'A' = 'A'
"Oh oh, what I have I done!"
```

You could have added an @show command on *z* as well.

```
1 @show typeof(w)
2 @show typeof(x)
3 @show typeof(y)
4 @show typeof(z)
```

Output

```
typeof(w) = Int64
typeof(x) = Float64
typeof(y) = Char
typeof(z) = String
```

String

Remark 1.2 While Julia <https://docs.julialang.org/en/v1/base/numbers/#Standard-Numeric-Types> has a lot of number TYPES, it does not have all of the possible categories of numbers that mathematicians use, <https://youtu.be/5TkIe60y2GI>. In addition to *Int64* and *Float64*, we'll use from to time *Complex64*.

1.2 Creating Simple Functions

One of the main points of programming is the creation of new functions. Here, we'll learn the simplest way to create a function in Julia and then how to plot a graph of the function.

```
1 f(x) = 5x^2 + 2x - 4 # this is too easy
```

Output

```
f (generic function with 1 method)
```

Note that we used **integer** coefficients when we composed the function. We can now evaluate the function at any point. Let's see what Type of variable is returned

```
1 @show f(3)
2 @show f(3.0)
3 @show f(-pi)
```

Output

```
f(3) = 47
f(3.0) = 47.0
f(-pi) = 39.06483669826721
39.06483669826721
```

The first one is clearly *Int64* (there is no decimal point) and the other two are *Float64*. Next, we'll modify the function and make at least one of the coefficients a decimal number. All of the outputs are now type *Float64*.

```
1 f(x) = 5.0x^2 + 2x - 4 # this is way too easy
2 @show f(3)
3 @show f(3.0)
4 f(-pi)
```

Output

```
f(3) = 47.0
f(3.0) = 47.0
39.06483669826721
```

1.3 Very Basic Plotting

Rich source of plotting examples: <https://gist.github.com/gizmaa/7214002>

“Tutorial” for plotting in Julia: <https://docs.juliaplots.org/latest/tutorial/>

Julia attempts to avoid filling the memory of your computer with functionality that you do not intend to use. It does that through the use of **packages**, which are collections of useful functions designed around a single theme, such as plotting (package is called Plots), random numbers (package is called Random), or linear algebra (package is called LinearAlgebra). We have pre-loaded a bunch of packages into the ROB 101 Julia installation. The process of accessing packages is a bit different if you are working in a personal Julia installation versus the course site; we’ll give both methods. In the course site,

```
1 # Turning on the plotting functions
2 using Plots
3 gr()
```

Output

```
Plots.GRBackend()
```

while if you are in a personal installation of Julia, do this

```
1 # For a personal installation the first time you use a package
2 using Pkg
3 Pkg.add("Plots")
4 # After the first time, you just need to run these two commands
5 using Plots
6 gr()
```

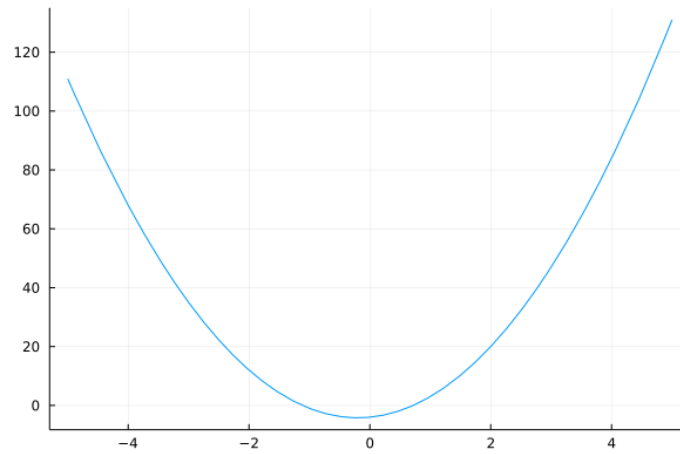
Output

```
Plots.GRBackend()
```

We assume the function f has already been defined. Here we are using $f(x) = 5.0x^2 + 2x - 4$

```
1 # Run me
2 # now, we can use the plot() function
3 plot(f, -5, 5, legend=false) #f will be plotted for x varying from -5 to 5
```

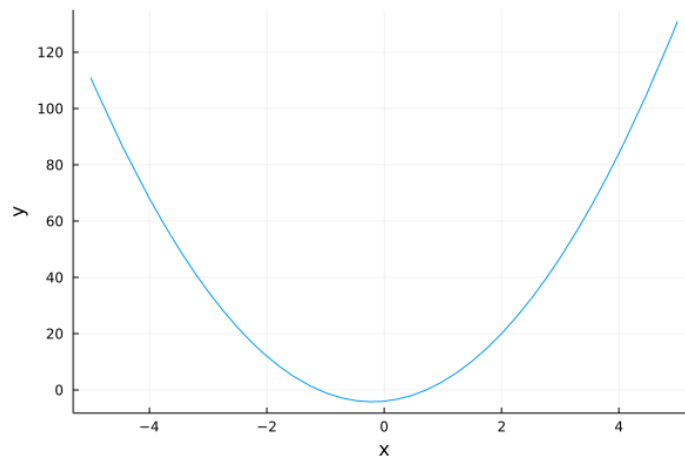
Output



We can add labels to the plot

```
1 plot!(xlabel = "x", ylabel = "y") # plot! modifies the previous plot
```

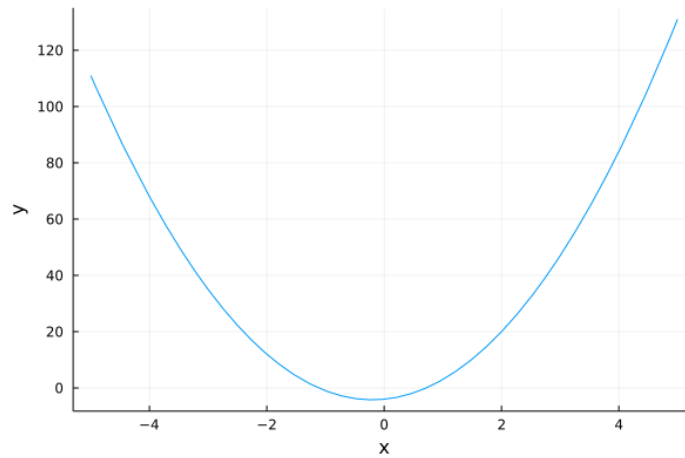
Output



We can put all the commands in one line if we want

```
1 # Run me
2 xmin=-5
3 xmax = 5
4 # You can add a bunch of stuff in one command line, if you want....
5 plot(f, xmin, xmax, xlabel = "x", ylabel = "y", legend = false)
```

Output



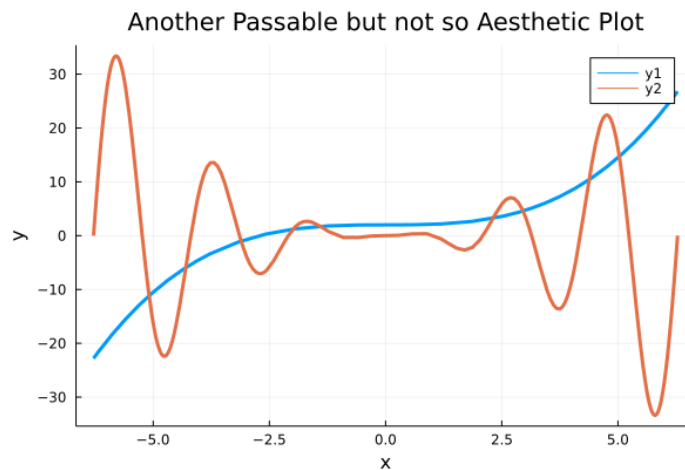
```

1 # Run me to see how to have two functions on the same plot
2 f(x) = .1x^3 + 2
3 g(x) = x^2*sin(3*x)
4 xmin = -2*pi
5 xmax = 2 *pi
6 titre = "Another Passable but not so Aesthetic Plot"
7 plot(f, xmin,xmax, titre=titre, linewidth=3)
8 # If you leave off the ! (bang), then the previous plot is erased and
9 # a new one is started. Try it.
10 plot!(g, xmin,xmax, titre=titre, linewidth=3) #plot bang, it's kind of joyous!
11 println("Every point where the graphs of f and g cross is a solution of the equation f(x) - g(x) = 0")
12 plot!(xlabel = "x", ylabel = "y") # plot! modifies the previous plot
13 # png("Lab01_TwoFunctionsSameGraph") #Creates a png of the graph, uncomment to use

```

Output

Every point where the graphs of f and g cross is a solution of the equation $f(x) - g(x) = 0$



```

1 # Run me to see how to place two functions on the same plot
2 f(x) = .1x^3 + 2
3 g(x) = x^2*sin(3*x)
4 xmin = -2*pi
5 xmax = 2 *pi
6 titre = "Another Passable but not so Aesthetic Plot"
7 plot([f,g], xmin,xmax, titre=titre, linewidth=3)

```



```
8 println ("Every point where the graphs of f and g cross is a solution of the equation f(x) - g(x) = 0")
9 plot!(xlabel = "x", ylabel = "y") # plot! modifies the previous plot
```

Output Same as from the previous cell.

Plotting in Julia is a bit awkward, in your instructor's opinion. We will mostly give you the plotting commands throughout the term. We'd rather you focus on other aspects of coding that are common to most programming languages.

To learn more about plotting in Julia: <https://docs.juliaplots.org/latest/tutorial/>

1.4 Creating Arrays

An **array** “is a data structure, which can store a fixed-size collection of elements of the same data type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.” https://www.tutorialspoint.com/computer_programming/computer_programming_arrays.htm. See also <https://press.rebus.community/programmingfundamentals/chapter/arrays-and-lists/>.

We will mostly be creating arrays with elements that are one of the following Types:

- (a) Int64
- (b) Float64
- (c) String

We'll first build an array with elements in a single row.

```
1 # Run me to create an array of 10 numbers
2 # a Matrix is one kind of array
3 # 1 x 10 Matrix{Int64} means it has one row and ten columns,
4 # with each entry of TYPE Int64
5 myArray = [1 2 3 4 5 6 7 8 9 10]
```

Output

```
1×10 Matrix{Int64}:
 1  2  3  4  5  6  7  8  9 10
```

Note that in Julia, what we call a **row vector** in lecture is being called a $1 \times n$ matrix, where in this case $n = 10$ because there are 10 entries in the vector. Note that Julia is confirming the data Type for us, 1×10 Matrix{Int64}. Each entry of the array has data type Int64. However, if we change even one of the entries to Float64, then **all of the entries** become Float64, as shown below.

```
1 # Run me to create an array of 10 numbers
2 # We change one of the values to a number of TYPE Float64,
3 # and leave the rest as TYPE Int64. Watch what happens
4 #
5 # 1 x 10 Matrix{Float64} means it has one row and ten columns,
6 # with each entry of TYPE Float64
7 println ("All elements assume TYPE Float64")
8 myArray = [1.0 2 3 4 5 6 7 8 9 10]
```

Output

```
All elements assume TYPE Float64
```

```
1 x 10 Matrix{Float64}:
 1.0  2.0  3.0  4.0  5.0  6.0  7.0  8.0  9.0 10.0
```

In Julia, arrays that are arranged in a column are called **Vectors**. What we call a **column vector** in lecture is simply a vector in Julia because Julia does not have two kinds of vectors. Julia only has column vectors and because of that, it is redundant to use the adjective “column”. OK! In lecture, it is not redundant to specify column vs row when talking about a vector.

In Julia, when you use **commas** or **semicolons** to separate the elements of an array to form a “column vector”, it is automatically declared as a **Vector**. Below, we build a 5-element `Vector{Float64}`. We used semicolons. You can replace the semicolons with commas and check that you get the same result. You cannot mix, however, commas and semicolons in the same array.

```
1 # Run me to create an array of 5 numbers
2 # arranged as a column
3 # 5-element Vector{Float64} means it has five elements,
4 # each with value of TYPE Float64
5 #
6 println("You can separate the elements by commas OR by semicolons. This is different than MATLAB.")
7 myArray = [1.0; 2; 3; 4; 5]
```

Output

You can separate the elements by commas OR by semicolons. This is different than MATLAB.

```
5-element Vector{Float64}:
 1.0
 2.0
 3.0
 4.0
 5.0
```

You can build vectors with elements that have type `String`.

```
1 # Run me to create an array with names in it
2 # 6-element Vector{String} means it is a column vector with
3 # elements of TYPE String. Note that we used commas this time.
4 animalsVector = ["lemur", "elephant", "tiger", "panda", "zebra", "cuttlefish"]
```

Output

```
6-element Vector{String}:
 "lemur"
 "elephant"
 "tiger"
 "panda"
 "zebra"
 "cuttlefish"
```

You can also create $1 \times n$ matrices with elements that have type `String`.

```
1 # Run me to see what happens when one uses spaces to separate the elements
2 animalsVector = ["lemur" "elephant" "tiger" "panda" "zebra" "cuttlefish"]
```

Output

```
1×6 Matrix{String}:
 "lemur" "elephant" "tiger" "panda" "zebra" "cuttlefish"
```

Exercise 1.3 Build three arrays, each with five elements

1. `myArray01` is a 1×5 matrix with entries of TYPE `Float64`
2. `myArray02` is a 5-element vector with entries of TYPE `Float64`
3. `myArray03` is a 5-element vector with entries of TYPE `String`

Solution

```
1 # Place your answers here
2 @show myArray01 = [1/3 11 4 2^(0.5) 7.62]
3 @show myArray02 = [1/3; 11; 4; 2^(0.5); 7.62]
4 myArray03 = ["un"; "deux"; "trois"; "quatre"; "cinque"]
```

Output

```
myArray01 = [1 / 3 11 4 2 ^ 0.5 7.62] = [0.3333333333333333 11.0 4.0 1.4142135623730951
7.62]
```

```
myArray02 = [1 / 3; 11; 4; 2 ^ 0.5; 7.62] = [0.3333333333333333, 11.0, 4.0,
1.4142135623730951, 7.62]
```

```
5-element Vector{String}:
```

```
"un"
"deux"
"trois"
"quatre"
"cinque"
```

1.5 Commenting out Lines of Code or Providing Explanations

```
1 # The pound sign comments out a single line
2 # x = 3
```

Output

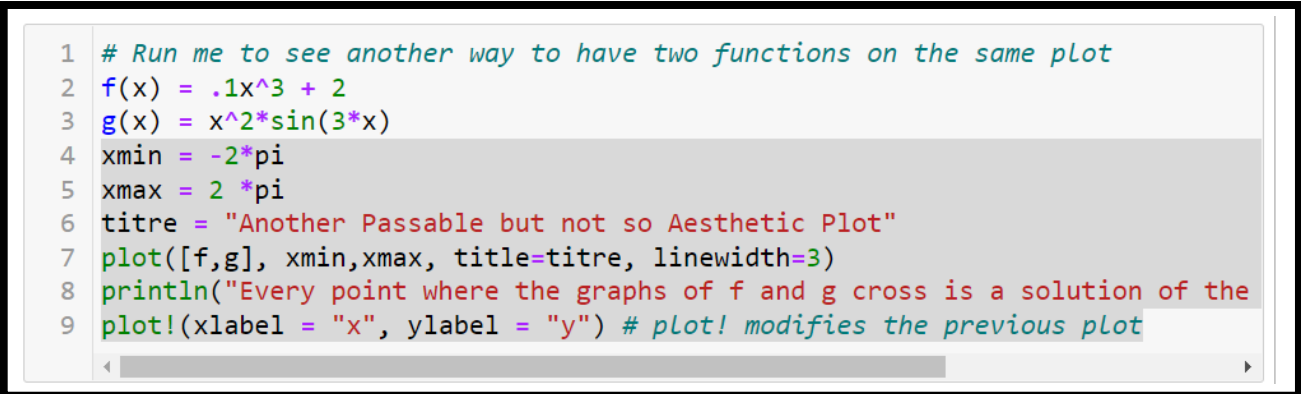
```
(nothing)
```

```
1 #
2 # Below is how you comment out multiple lines at one time
3 #=
4 enter comments or code here
5 it can take as many
6 lines as you want
7 =#
```

Output

```
(nothing)
```

At least on a PC, you can highlight a section of code with your mouse



```
1 # Run me to see another way to have two functions on the same plot
2 f(x) = .1x^3 + 2
3 g(x) = x^2*sin(3*x)
4 xmin = -2*pi
5 xmax = 2 *pi
6 titre = "Another Passable but not so Aesthetic Plot"
7 plot([f,g], xmin,xmax, title=titre, linewidth=3)
8 println("Every point where the graphs of f and g cross is a solution of the
9 plot!(xlabel = "x", ylabel = "y") # plot! modifies the previous plot
```

and then hit `Ctrl /` / control-forward-slash (simultaneously) to comment out or un-comment multiple lines at a time.

```
1 # Run me to see another way to have two functions on the same plot
2 f(x) = .1x^3 + 2
3 g(x) = x^2*sin(3*x)
4 # xmin = -2*pi
5 # xmax = 2 *pi
6 # titre = "Another Passable but not so Aesthetic Plot"
7 # plot([f,g], xmin,xmax, title=titre, linewidth=3)
8 # println("Every point where the graphs of f and g cross is a solution of th
9 # plot!(xlabel = "x", ylabel = "y") # plot! modifies the previous plot
```

1.6 Applying Functions to Arrays via Broadcasting

Julia has a special syntax for applying functions to individual elements of arrays. You need to add a “dot” after the function and before the argument to the function. It’s best understood by doing it.

```
x = [0 pi/4 pi/2 3pi/4 pi 5pi/4]
```

Output

```
1×6 Matrix{Float64}:
 0.0  0.785398  1.5708  2.35619  3.14159  3.92699
```

```
sin.(x) # note the dot
```

Output

```
1×6 Matrix{Float64}:
 0.0  0.707107  1.0  0.707107  1.22465e-16  -0.707107
```

This also works with functions that you write yourself! It pretty awesome.

```
f(x) = 1 + 2x + sin(x^2)
```

Output

```
f (generic function with 1 method)
```

```
f.(x) # note the dot
```

Output

```
1×6 Matrix{Float64}:
 1.0  3.14927  4.76586  5.04438  6.85288  9.13678
```

For an array, $x = [x_1 \ x_2 \ \dots \ x_n]$, the returned value is $f.(x) = [f(x_1) \ f(x_2) \ \dots \ f(x_n)]$. This feature is called “broadcasting”. If you leave out the “dot”, this is what you get

```
f(x)
```

Output

```
DimensionMismatch("A has dimensions (1,6) but B has dimensions (1,6)")
```

```
Stacktrace:
```

```

[1] gemm_wrapper!(C::Matrix{Float64}, tA::Char, tB::Char, A::Matrix{Float64},
B::Matrix{Float64}, _add::LinearAlgebra.MulAddMul{true, true, Bool, Bool})
    @ LinearAlgebra /buildworker/worker/package_linux64/build/usr/share/julia/stdlib/
v1.6/LinearAlgebra/src/matmul.jl:643
[2] mul!
    @ /buildworker/worker/package_linux64/build/usr/share/julia/stdlib/v1.6/
LinearAlgebra/src/matmul.jl:169 [inlined]
[3] mul!
    @ /buildworker/worker/package_linux64/build/usr/share/julia/stdlib/v1.6/
LinearAlgebra/src/matmul.jl:275 [inlined]
[4] *
    @ /buildworker/worker/package_linux64/build/usr/share/julia/stdlib/v1.6/
LinearAlgebra/src/matmul.jl:160 [inlined]
[5] power_by_squaring(x_::Matrix{Float64}, p::Int64)
    @ Base ./intfuncs.jl:261
[6] ^
    @ /buildworker/worker/package_linux64/build/usr/share/julia/stdlib/v1.6/
LinearAlgebra/src/dense.jl:442 [inlined]
[7] macro expansion
    @ ./none:0 [inlined]
[8] literal_pow
    @ ./none:0 [inlined]
[9] f(x::Matrix{Float64})
    @ Main ./In[8]:1
[10] top-level scope
    @ In[10]:1
[11] eval
    @ ./boot.jl:360 [inlined]
[12] include_string(mapexpr::typeof(REPL.softscope), mod::Module, code::String,
filename::String)
    @ Base ./loading.jl:1094

```

You can also use the “dot” with addition or multiplication. Below, we add one to each entry of the array x .

```
1 x .+ 1
```

Output

```
1×6 Matrix{Float64}:
 1.0  1.7854  2.5708  3.35619  4.14159  4.92699
```

The following code multiplies the entries of x times the corresponding entries in y ,

```
1 x = [1 2 3 4 5]
2 y = [5 4 3 2 1]
3 x.*y # note the dot
```

Output

```
1×5 Matrix{Int64}:
 5  8  9  8  5
```

The same applies to squaring the entries of an array, such as

```
1 x.^2 # note the dot
```

Output

```
1×5 Matrix{Int64}:  
 1  4  9 16 25
```

The following applies the function $f(x)$ to each entry of x and then multiplies each entry of the resulting array times the corresponding entry in y .

```
z=f.(x).*y # we used two dots
```

Output

```
1×5 Matrix{Float64}:  
19.2074 16.9728 22.2364 17.4242 10.8676
```

The following applies the function $f(x)$ to each entry of x , then multiplies each entry of the resulting array times the corresponding entry in y , and finally adds 2 to each entry.

```
z=f.(x).*y .+ 2 # we used three dots
```

Output

```
1×5 Matrix{Float64}:  
21.2074 18.9728 24.2364 19.4242 12.8676
```

We note that $z = [z_1 \ z_2 \ \dots \ z_n]$, where for $1 \leq i \leq n$, the components of z are given by $z_i = f(x_i) * y_i + 2$.

1.7 Debugging

We created our first function in Chapter 1.3. Here we'll show a common programming error that you might think is a bug.

```
g=2 # set the variable g to a constant
```

Output

```
2
```

That's pretty simple! Now, imagine that you used the variable g 15 cells back and have completely forgotten about it. You then want to create a function and call it g .

```
g(x) = 5x + 2
```

Output

```
cannot define function g; it already has a value
```

```
Stacktrace:
```

```
[1] top-level scope  
  @ none:0  
[2] top-level scope  
  @ In[4]:1  
[3] eval  
  @ ./boot.jl:360 [inlined]  
[4] include_string(mapexpr::typeof(REPL.softscope), mod::Module,  
code::String, filename::String)  
  @ Base ./loading.jl:1094
```

In the example, Cell [4] is the cell where we defined $g(x)$. There are two cures for this:

- Clear your Kernel, **which will erase the variables in your Jupyter notebook without erasing your code**. You can then run the cell above again and it will work fine.
- Choose a different name for the function, which we illustrate below.

```
g2(x) = 5x + 2
```

Output

```
g2 (generic function with 1 method)
```

Debugging

- Taken From the Pro-tip page at Georgia Tech for a graduate-level course: https://cse6040.gatech.edu/fa22/pro_tips.html
- Debugging is an iterative process where you must trace the undesired behavior back to the root cause. Sometimes it's simple; other times it can be frustratingly complex.
- Identify what is wrong. This needs to be as precise as possible. If you don't understand the error in the traceback - Google it!. Check the syntax, parameters, and inputs of any function calls.
- Identify where the wrong thing got set. This will usually be an assignment or a function call.
- Rinse and repeat until you have found and corrected the root cause.

Chapter 2

Julia Lab 2: Vectors, Matrices, and Indexing

Learning Objectives

- More ways to create vectors and matrices
- Indexing as a means to access the elements of vectors and matrices
- Debugging, or finding and fixing errors in your code

Outcomes

- Vectors
- Matrices
 - `zeros(n,m)`
 - `ones(n,m)`
 - using `Random`
 - `rand(n,m)` and `randn(n,m)`
- Adding columns or rows to matrices, concatenation
- Indexing into row vectors (aka $1 \times n$ matrices in Julia) and column vectors (aka simply a vector in Julia)
- Indexing into matrices
 - Extracting columns is easy
 - Extracting rows requires concentration
 - Extracting sub-matrices
 - Help command in Julia
- Creating “empty” matrices via `Vector{Float64}(undef, n)` and `Matrix{Float64}(undef, n, m)`, and
- `B = copy(A)` and how that is different than `B=A`
- Creating an identity matrix in Julia is a bit odd.

Either download Lab2 from our Canvas site or open up a Jupyter notebook so that you can enter code as we go. It is suggested that you have line numbering toggled on.

At the end of Lab 1, we saw how to create (small) row vectors, columns vectors and $n \times m$ matrices. You may want to review that material before starting here.

2.1 Friendly Checks

In your lab assignments, Julia HWs, and Projects, we are inserting here and there a means to detect errors in your code before you get too deep into the assignments. **Mostly, we are not checking that you are correct, we are instead checking that you are not obviously wrong.** In mathematical language, we are checking “necessary conditions” but not “sufficient conditions” for you to be correct. In simple terms, you can pass our tests, and still be wrong. You will just not be wrong in a way that we anticipated, which is kind of a bummer for you.

Remark 2.1 *We have messed up in the past by updating the numbers in a problem and failing to update the associated tests! If you and your classmates are all failing a test, let us know and we'll double check our test. In the end, we are trying to decrease your frustration and uncertainty. Sometimes, we add to it! C'est la vie !*

Here is an example of how the tests work.

```
1 # Warm up work for you: # Create a column vector (called almost_zero_vec)
2 # that has 4 entries: the 2nd entry is -2.5, and others are zeros
3 #
4 almost_zero_vec = NaN      #Replace NaN with your answer
```

Output

NaN

Here is the solution:

```
1 # Warm up work for you: # Create a column vector (called almost_zero_vec)
2 # that has 4 entries: the 2nd entry is -2.5, and others are zeros
3 #
4 almost_zero_vec = [0, -2.5, 0, 0]      #Replace NaN with your answer
```

Output

```
4-element Vector{Float64}:
 0.0
-2.5
 0.0
 0.0
```

Here is an example test, **designed to not give away the answer and yet, try to let you know if you are wrong:**

```
1 # friendly check
2 test1 = almost_zero_vec' * almost_zero_vec
3 ind = [1 3 4]
4 test2 = sum(abs.(almost_zero_vec[ind]))
5 test3 = length(almost_zero_vec)
6
7 is_it_correct_check1 = (test1 == 6.25) ? "Yes" : "No"
8 is_it_correct_check2 = (test2 == 0.0) ? "Yes" : "No"
9 is_it_correct_check3 = (test3 == 4) ? "Yes" : "No"
10
11 @show is_it_correct_check1;
12 @show is_it_correct_check2;
13 @show is_it_correct_check3;
```

Output

```
is_it_correct_check1 = "Yes"
is_it_correct_check2 = "Yes"
is_it_correct_check3 = "Yes"
```

In this case, you're good to go. Suppose you had instead posed the solution as

```
1 # Warm up work for you: # Create a column vector (called almost_zero_vec)
2 # that has 5- entries: the 2nd entry is -2.5, and others are zeros
3 #
4 almost_zero_vec = [0, 2.5, 0, 0, 0] #Replace NaN with your answer
```

Output

```
5-element Vector{Float64}:
 0.0
 2.5
 0.0
 0.0
 0.0
```

Then the friendly test let's you know that something is wrong via the line `is_it_correct_check3 = "No"`. **That you failed test3 is not meant to be particularly helpful to your debugging process.** You can ignore that part.

```
1 @show is_it_correct_check3;
```

Output

```
is_it_correct_check1 = "Yes"
is_it_correct_check2 = "Yes"
is_it_correct_check3 = "No"
```

Most of our tests do not specify the nature of your mistake. You need to find it yourself. Our goal is simply to prevent you from compounding errors as you proceed with an assignment. You may be able to study our error-checking code to see, as in this case, we were checking the `length` of the vector (a command we have not yet taught you), but normally we try to hide what we are checking.

Below is a wrong answer that passes our test because we are not checking the sign of the non-zero number. We just assume you will get that part correct.

```
1 # Warm up work for you: # Create a column vector (called almost_zero_vec)
2 # that has 4 entries: the 2nd entry is -2.5, and others are zeros
3 #
4 almost_zero_vec = [0, 2.5, 0, 0] #Replace NaN with your answer
```

Output

```
4-element Vector{Float64}:
 0.0
 2.5
 0.0
 0.0
```

The output of the test code is

```
is_it_correct_check1 = "Yes"
is_it_correct_check2 = "Yes"
is_it_correct_check3 = "Yes"
```

From the Friendly check, it looks like you are good to go, even though you got the sign wrong. On the other hand, you had the right number of elements, the magnitude of the second value was correct, and you had the zero elements in the correct locations. We checked for a lot of things, but not everything. In the pilot offering of ROB 101, there were no friendly checks at all!

2.2 New Ways to Create Matrices

2.2.1 Using Built-in Functions

We'll quickly illustrate the following methods for creating matrices. The `rand` command is very useful for creating examples that have hundreds of variables (for later in the course).

- `zeros(n,m)`
- `ones(n,m)`
- using `Random`
- `rand(n,m)` (uniform) and with an extra “n” `randn(n,m)` (normal, aka Gaussian or Bell Curve)

```
1 # Creates a matrix of zeros with n-rows and m-columns
2 A = zeros(3,5)
```

Output

```
3×5 Matrix{Float64}:
 0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0
```

```
1 # Creates a matrix of ones with n-rows and m-columns
2 B = ones(5,3)
```

Output

```
5×3 Matrix{Float64}:
 1.0  1.0  1.0
 1.0  1.0  1.0
 1.0  1.0  1.0
 1.0  1.0  1.0
 1.0  1.0  1.0
```

```
1 # Install the package Random
2 using Random
```

Output

(none)

The command `rand(n)` or `rand(n,m)` produces numbers that are “uniformly distributed between zero and one”. This roughly means that each $x \in [0, 1]$ is equally likely; a more precise statement would be that, for all $0 \leq a < b \leq 1$, the probability that $x \in [a, b]$ is equal to $b - a$. The related command `randn(n)` or `randn(n,m)` produces numbers that are “normally distributed between minus infinity and plus infinity”. This means that the random number comes from a Bell curve.

```
1 # Run me
2 Random.seed!(4321) # Set the seed so that each of you get the same results.
3 @show myVector = rand(9) # column vector
4 myMatrix = rand(3,4) # 3 x 4 matrix with random entries in [0,1]
```

Output

```
myVector = rand(9) = [0.1619496289112512, 0.1385032717390522, 0.6355055732000539,
0.36326986232801595, 0.3700559311804976, 0.8332539255123794, 0.6910169607353331,
0.08205602210008567, 0.6441883306325207]
```

```
3×4 Matrix{Float64}:
 0.873045  0.413895  0.0578154  0.55376
 0.705639  0.492289  0.914811  0.544223
 0.528017  0.24885  0.917523  0.478742
```

```
1 # Run me
2 Random.seed!(4321) # Set the seed so that each of you get the same results.
3 myMatrix = randn(6,5) # 6 x 5 matrix with random entries in (-inf, inf)
```

Output

```
6×5 Matrix{Float64}:
-0.229071 -0.991273 -0.850184 -1.06297  0.609128
 0.321924 -0.158205  0.592798 -0.509626 -0.547334
 0.74283  1.02869 -0.358643  0.673963  0.24726
 0.948465 -1.94117 -0.0688075  0.317936 -0.860824
-0.540755 -1.00753  1.51188 -1.13022  1.45335
-1.95979  0.825056 -0.718068 -0.0391889  0.453098
```

```
1 # Run me
2 # if we do not set the seed, each time we get a new set of random numbers.
3 myMatrix = randn(6,5) # 6 x 5 matrix with random entries in (-inf, inf)
```

Output

```
6×5 Matrix{Float64}:
 0.498376  0.411742 -0.753106 -0.984998  0.53562
-0.670348 -0.826093 -0.219619 -0.582682  0.151646
-0.52135  0.680463  0.885311 -0.703102  1.64057
-0.71363 -0.322208 -0.991709 -1.44334 -1.41471
 0.249046  0.373825 -1.04189 -0.254509 -0.0348657
 0.851102 -0.044526 -1.34575  2.70248  0.666966
```

We'll use the ability to generate random matrices of arbitrary size when we start testing our own algorithms. Let's try $A = \text{randn}(100, 100)$. Julia cannot print out all of its entries. Julia is showing here the first three columns and last three columns, of the first 13 rows and the last 12 rows.

```
1 # Run me
2 A = randn(100,100)
```

Output

```
100×100 Matrix{Float64}:
-1.21507  1.51014  -0.358913  ...  0.202398  0.92658  0.300415
-0.0293227 -0.643004  0.941623  ...  0.326798 -0.847375  0.174174
 1.20972 -0.745284 -1.57119  ...  1.13692 -2.35792 -0.191441
-0.36017 -0.400969  0.398202  ... -0.194262 -0.785111 -0.764176
 0.243962  0.102421 -1.74552  ...  0.0195486 -0.475647  1.23968
-0.586099 -0.481138 -0.396337  ...  2.16696  0.145727  0.348194
 0.810478 -1.41851  0.661778 -0.110213  0.397295  0.908824
-0.173821 -0.0334777  0.593349  1.00495 -1.38393 -1.6448
-1.13339 -0.911688  1.52026 -0.293288  0.612828  0.45864
 0.493715 -2.1743 -1.14563  1.00688 -1.27257 -0.0470864
 1.03548  1.01407 -0.586865  ... -1.33328  0.683025  0.215578
-0.208806  0.89081  1.14522  1.05247  0.55675 -0.0393328
-0.730545 -0.51702  0.642981  0.429845 -0.869449  0.75665
  :
  :
```

```

-0.421939  -0.66485  -1.33336  -1.06668  0.0644415  0.0635774
-1.18934  -0.465795  0.461657  0.247114  -1.71652  -1.9573
-0.129762  0.434988  0.23351  ...-0.875235  -0.963982  -0.247793
 1.2493    0.646449  0.236439  -0.861302  -0.764389  0.858528
-0.516579  0.0443446  -1.08403  -1.23918  -1.24305  -0.386032
-0.916162  -0.0758758  -0.421691  1.52287  0.527025  1.24463
-0.684109  -0.107738  -0.461977  -0.110118  1.00922  0.489418
-1.11484   -0.38536  -0.631  ... 1.24931  -0.290083  1.79908
-0.0738447  1.76986  1.21057  0.575938  -0.275458  0.324595
 0.0713201  0.190427  -1.85452  1.33801  -0.0978082  0.34016
 0.238514   1.22612  0.177986  -1.08624  1.82018  -0.47512
 0.249298   0.00947162  -1.33509  -0.728613  -0.992987  0.0453849

```

2.2.2 Concatenation or Adding Columns and Rows to Matrices

Another way to build a matrix is by concatenating columns or rows. We illustrate that now.

Suppose that v_1, \dots, v_4 are row vectors (what Julia calls a $1 \times n$ matrix). Then we can use them as the rows of a matrix, as in

$$M = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \end{bmatrix}$$

Similarly, suppose that v_1, \dots, v_4 are column vectors (what Julia calls a vector). Then we can use them as the columns of a matrix, as in

$$M = [v_1 \ v_2 \ v_3 \ v_4]$$

```

1 vRow1 = [1 2 3]
2 vRow2 = [4 5 6]
3 vRow3 = [7 8 9]
4 vRow4 = [10 11 12]
5 #
6 # We will concatenate the row vectors to form a 4 x 3 matrix
7 #
8 M = [vRow1; vRow2; vRow3; vRow4] # separated by semicolons

```

Output

```

4×3 Matrix{Int64}:
 1  2  3
 4  5  6
 7  8  9
10 11 12

```

```

1 vCol1 = [1, 2, 3]
2 vCol2 = [4, 5, 6]
3 vCol3 = [7, 8, 9]
4 vCol4 = [10, 11, 12]
5 #
6 # We will concatenate the column vectors to form a 3 x 4 matrix
7 #
8 M = [vCol1 vCol2 vCol3 vCol4] # separated by spaces

```

Output

```
3×4 Matrix{Int64}:
 1  4  7 10
 2  5  8 11
 3  6  9 12
```

We can even build matrices from other matrices, as long as they are **size compatible**. We illustrate. Suppose that A_{11} is $n_1 \times m_1$ and A_{12} is $n_1 \times m_2$, where both A_{11} and A_{12} have the same number of rows. Then we can form

$$M = [A_{11} \quad A_{12}].$$

M now has $m_1 + m_2$ columns. Hence, if B is $n_2 \times (m_1 + m_2)$, then it has the same number of columns as M and thus we can add B to M as additional rows, like this,

$$N = \begin{bmatrix} M \\ B \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ & B \end{bmatrix}.$$

we illustrate this now.

```
1 A11 = [1 2; 3 4]
2 A12 = [5; 6]
3 M = [A11 A12]
4 display (M)
5
6 B = [7 8 9; 10 11 12]
7 display (B)
8
9 N = [M; B] # Build the matrix piece by piece
10 display (N)
11
12 N2 = [A11 A12; B] # Build the matrix all at once. Both are fine.
```

Output

```
2×3 Matrix{Int64}:
 1  2  5
 3  4  6
```

```
2×3 Matrix{Int64}:
 7  8  9
10 11 12
```

```
4×3 Matrix{Int64}:
 1  2  5
 3  4  6
 7  8  9
10 11 12
```

```
4×3 Matrix{Int64}:
 1  2  5
 3  4  6
 7  8  9
10 11 12
```

```
1 # With practice, you do this
2 A11 = [1 2; 3 4]
3 A12 = [5; 6]
4 B = [7 8 9; 10 11 12]
5
6 C = [A11 A12; B]
```

Output

```
2×3 Matrix{Int64}:
```

```
 1  2  5
 3  4  6
```

```
1×3 Matrix{Int64}:
```

```
 7  8  9
```

```
3×3 Matrix{Int64}:
```

```
 1  2  5
 3  4  6
 7  8  9
```

If A_{11} , A_{12} , A_{21} , and A_{22} are size compatible matrices, we can do this $M = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$.

What do the sizes have to be?

- A_{11} and A_{12} must have the same number of rows, so that we can form $[A_{11} \ A_{12}]$.
- A_{21} and A_{22} must have the same number of rows, which can be different from A_{11} and A_{12} . This allows the formation of $[A_{21} \ A_{22}]$.
- $[A_{11} \ A_{12}]$ and $[A_{21} \ A_{22}]$ must have the same number of columns.
- In other words, if A_{ij} is $n_{ij} \times m_{ij}$, then we need
 - * $n_{11} = n_{12}$ (rows match)
 - * $n_{21} = n_{22}$ (rows match)
 - * $m_{11} + m_{12} = m_{21} + m_{22}$ (column sums match)

```
1 A11 = [1 2; 3 4]
2 A12 = [5; 6]
3 A21 = [7; 8]
4 A22 = [9 10; 11 12]
5
6 M = [A11 A12; A21 A22]
```

Output

```
4×3 Matrix{Int64}:
```

```
 1  2  5
 3  4  6
 7  9 10
 8 11 12
```

2.3 Indexing or Slicing Vectors

Remark 2.2 Julia uses ****1-based indexing**** which means that the index starts at 1 and not 0. Be aware that 0-based indexing is used in some other programming languages, such as C++. If you are familiar with MATLAB, it uses 1-based indexing.

```
1 # Run me to define some vectors and matrices
2 @show row_vec = [1 3 5 7 9] # spaces make me a row vector
3 almost_zero_vec=[0; 0; -pi; 0; 0; 0; 0] # the semicolons (commas also work)
4                                     # make me a column vector
```

Output


```
row_vec = [1 3 5 7 9] = [1 3 5 7 9]
```

```
7-element Vector{Float64}:  
 0.0  
 0.0  
 -3.141592653589793  
 0.0  
 0.0  
 0.0  
 0.0
```

```
1 # Select the 1st entry from row_vector  
2 num = row_vec[1]  
3 #= Can you tell that the type of num is Int64?  
4 If you are unsure, uncomment and run the next command  
5 =#  
6 # typeof(num)
```

Output

```
1
```

```
1 # Select the non-zero number from almost_zero_vec  
2 num = almost_zero_vec[3]  
3 # Note: because a vector has only one dimension,  
4 # only one index is needed and it corresponds to  
5 # the location where the number is stored
```

Output

```
-3.141592653589793
```

You can select multiple entries at one time.

```
1 # Select the first, second, and fourth entries of the  
2 # vector row_vec and return them as a row vector  
3 #  
4 ind = [1 2 4] # note the use of spaces instead of commas  
5 #           or semicolons so as to return a row vector.  
6 result = row_vec[ind]
```

Output

```
1×3 Matrix{Int64}:  
 1  3  7
```

```
1 # You can also do it like this, but it's kind of ugly  
2 result = row_vec[[1 2 4]]
```

Output

```
1×3 Matrix{Int64}:  
 1  3  7
```

```
1 # Select the first, second, and fourth entries of the  
2 # vector row_vec and return them as a column vector  
3 #
```

```
4 ind = [1, 2, 4] # note the use of commas (semicolons also work)
5 #           so as to return a column vector.
6 result = row_vec[ind]
```

Output

```
3-element Vector{Int64}:
 1
 3
 7
```

2.4 Indexing or Slicing Matrices

Remark 2.3 We recall once again that Julia uses ***1-based indexing*** which means that the index starts at 1 and not 0. Be aware that 0-based indexing is used in some other programming languages, such as C++. If you are familiar with MATLAB, it uses 1-based indexing.

```
1 # Run me to define a matrix with random numbers
2 #
3 using Random # Using an external package called Random
4 Random.seed!(1234) # Set the seed so that each of you gets the same results.
5 rand_matrix = rand(4, 6)
```

Output

```
4×6 Matrix{Float64}:
 0.590845  0.794026  0.246837  0.066423  0.276021  0.950498
 0.766797  0.854147  0.579672  0.956753  0.651664  0.96467
 0.566237  0.200586  0.648882  0.646691  0.0566425  0.945775
 0.460085  0.298614  0.0109059  0.112486  0.842714  0.789904
```

```
1 # Select the entry in the 2nd row and 3rd column of rand_matrix
2 #
3 myNum = rand_matrix[2, 3]
4 #
5 # Note: 2 is the index for the row, and 3 is the index for the column
6 #
7 # Can you tell that the type of myNum is Float64?
8 # If you are unsure, uncomment and run the next command
9 # typeof(myNum)
```

Output

```
0.5796722333690416
```

```
1 # Select the entire 4th column of rand_matrix
2 #
3 myVect = rand_matrix[:, 4]
4 #
5 # Note: when we want to select the whole column,
6 # we use `:` as the index for row, and vice versa.
7 # The symbol ":" stands for "all entries"
```

Output

```
4-element Vector{Float64}:
 0.06642303695533736
 0.9567533636029237
 0.646690981531646
 0.11248587118714015
```

Remark 2.4 *Selecting a row of a matrix is trickier than selecting a column. Pay attention here. This is very different than MATLAB.*

```
1 # Select the 2nd row of rand_matrix
2 #
3 myVect = rand_matrix[2:2, :] # note the use of "2:2" instead of simply "2"
```

Output

```
1×6 Matrix{Float64}:
 0.766797 0.854147 0.579672 0.956753 0.651664 0.96467
```

Remark 2.5 *Pay attention here. This is very different than MATLAB. Julia hates row vectors so much that it calls a row vector a $1 \times n$ Matrix. Every chance it gets, Julia creates a column vector and then calls it an n -element Vector. You need to be aware of this peculiarity of the language. Forgetting it will lead to frustration and heartbreak!*

```
1 # If you only place a "2" instead of "2:2" you get a column vector
2 #
3 myVect = rand_matrix[2, :]
```

Output

```
6-element Vector{Float64}:
 0.7667970365022592
 0.8541465903790502
 0.5796722333690416
 0.9567533636029237
 0.6516642063795697
 0.9646697763820897
```

```
1 # What if you use the m:m trick on a column?
2 #
3 myVect = rand_matrix[:, 4:4]
4 #
5 # You get a 4 x 1 Matrix. While this is not a Vector in Julia,
6 # for most uses, it is perfectly fine.
```

Output

```
4×1 Matrix{Float64}:
 0.06642303695533736
 0.9567533636029237
 0.646690981531646
 0.11248587118714015
```

2.5 More Advanced Remarks on Indexing

We'll use the matrix `rand_matrix` to show how to select "sub-matrices" from it.

```
1 # Run me
2 rand_matrix
```

Output

```
4x6 Matrix{Float64}:
 0.590845  0.794026  0.246837  0.066423  0.276021  0.950498
 0.766797  0.854147  0.579672  0.956753  0.651664  0.96467
 0.566237  0.200586  0.648882  0.646691  0.0566425  0.945775
 0.460085  0.298614  0.0109059  0.112486  0.842714  0.789904
```

We will now select out the rows in red,

$$\begin{bmatrix} 0.5908 & 0.7940 & 0.2468 & 0.0664 & 0.2760 & 0.9505 \\ 0.7668 & 0.8541 & 0.5797 & 0.9568 & 0.6517 & 0.9647 \\ 0.5662 & 0.2006 & 0.6489 & 0.6467 & 0.0566 & 0.9458 \\ 0.4601 & 0.2986 & 0.0109 & 0.1125 & 0.8427 & 0.7899 \end{bmatrix} \quad (2.1)$$

```
1 indRow = [2, 4] # Note that we wanted all the columns,
2 mySlice=rand_matrix[indRow, :] # so we use the colon : to do that
```

Output

```
2x6 Matrix{Float64}:
 0.766797  0.854147  0.579672  0.956753  0.651664  0.96467
 0.460085  0.298614  0.0109059  0.112486  0.842714  0.789904
```

We will now select out the columns in red,

$$\begin{bmatrix} 0.5908 & 0.7940 & 0.2468 & 0.0664 & 0.2760 & 0.9505 \\ 0.7668 & 0.8541 & 0.5797 & 0.9568 & 0.6517 & 0.9647 \\ 0.5662 & 0.2006 & 0.6489 & 0.6467 & 0.0566 & 0.9458 \\ 0.4601 & 0.2986 & 0.0109 & 0.1125 & 0.8427 & 0.7899 \end{bmatrix} \quad (2.2)$$

```
1 indCol = [3, 5, 6] # Note that we wanted all the rows,
2 mySlice=rand_matrix[:, indCol] # so we use the colon : to do that
```

Output

```
4x3 Matrix{Float64}:
 0.246837  0.276021  0.950498
 0.579672  0.651664  0.96467
 0.648882  0.0566425  0.945775
 0.0109059  0.842714  0.789904
```

We will now select out the entries in red,

$$\begin{bmatrix} 0.5908 & 0.7940 & 0.2468 & 0.0664 & 0.2760 & 0.9505 \\ 0.7668 & 0.8541 & 0.5797 & 0.9568 & 0.6517 & 0.9647 \\ 0.5662 & 0.2006 & 0.6489 & 0.6467 & 0.0566 & 0.9458 \\ 0.4601 & 0.2986 & 0.0109 & 0.1125 & 0.8427 & 0.7899 \end{bmatrix} \quad (2.3)$$

```
1 # Select rows 2 and 3 of columns 3 and 4
2 indRow = [2, 3] # Yes, commas are used. Try it without the commas here
3 indCol = [3, 4]
4 mySlice=rand_matrix[indRow, indCol]
```

Output

```
2x2 Matrix{Float64}:
 0.579672  0.956753
 0.648882  0.646691
```

We will next select out the entries in red,

$$\begin{bmatrix} 0.5908 & 0.7940 & 0.2468 & 0.0664 & 0.2760 & 0.9505 \\ 0.7668 & 0.8541 & 0.5797 & 0.9568 & 0.6517 & 0.9647 \\ 0.5662 & 0.2006 & 0.6489 & 0.6467 & 0.0566 & 0.9458 \\ 0.4601 & 0.2986 & 0.0109 & 0.1125 & 0.8427 & 0.7899 \end{bmatrix} \quad (2.4)$$

```
1 # Select rows 1,2, and 4 of columns 1 and 6
2 indRow = [1, 2, 4] # Yes, commas are used. Try it without the commas here
3 indCol = [1, 6]
4 mySlice=rand_matrix[indRow, indCol]
```

Output

```
3×2 Matrix{Float64}:
 0.590845  0.950498
 0.766797  0.96467
 0.460085  0.789904
```

We will select out the entries in red,

$$\begin{bmatrix} 0.5908 & 0.7940 & 0.2468 & 0.0664 & 0.2760 & 0.9505 \\ 0.7668 & 0.8541 & 0.5797 & 0.9568 & 0.6517 & 0.9647 \\ 0.5662 & 0.2006 & 0.6489 & 0.6467 & 0.0566 & 0.9458 \\ 0.4601 & 0.2986 & 0.0109 & 0.1125 & 0.8427 & 0.7899 \end{bmatrix} \quad (2.5)$$

```
1 # Select from column 3 to the end of rows 2 and 3
2 indRow = [2, 3] # Yes, commas are used. Try it without the commas here
3 mySlice=rand_matrix[indRow, 3:end] # note 3:end is the same as 3:6
```

Output

```
2×4 Matrix{Float64}:
 0.579672  0.956753  0.651664  0.96467
 0.648882  0.646691  0.0566425  0.945775
```

Here is another way to do the same thing,

```
1 # Select from column 3 to the end of rows 2 and 3
2 indRow = [2, 3] # Yes, commas are used. Try it without the commas here
3 indCol = collect(3:6) # look up the command ``collect``
4 @show indCol
5 mySlice=rand_matrix[indRow, indCol]
```

Output

```
indCol = [3, 4, 5, 6]
```

```
2×4 Matrix{Float64}:
 0.579672  0.956753  0.651664  0.96467
 0.648882  0.646691  0.0566425  0.945775
```

Remark 2.6 To get help on a function in Julia, use a question mark, followed by a space, followed by the name of the function. We illustrate here for the function `collect`

```
1 ? collect
```

Output

```
search: collect
```

```
collect(element_type, collection)
```

Return an Array with the given element type of all items in a collection or iterable. The result has the same shape and number of dimensions as collection.

Examples

```
julia> collect(Float64, 1:2:5)
```

```
3-element Vector{Float64}:
```

```
1.0
3.0
5.0
```

```
collect(collection)
```

Return an Array of all items in a collection or iterator. For dictionaries, returns Pair{KeyType, ValType}. If the argument is array-like or is an iterator with the [HasShape](@ref IteratorSize) trait, the result will have the same shape and number of dimensions as the argument.

Examples

```
julia> collect(1:2:13)
```

```
7-element Vector{Int64}:
```

```
1
3
5
7
9
11
13
```

Remark 2.7 *At least in the ROB 101 version of Julia, any call to a help function must be the only thing in the cell. Even inserting a comment breaks the help command.*

```
1 # see what happens when you lead with a comment
2 ? collect
```

Output

```
syntax: invalid identifier name "?"
```

Stacktrace:

```
[1] top-level scope
    @ In[11]:2
[2] eval
    @ ./boot.jl:360 [inlined]
[3] include_string(mapexpr::typeof(REPL.softscope), mod::Module, code::String,
filename::String)
    @ Base ./loading.jl:1094
```

2.6 Advanced Methods for Creating Vectors and Matrices

We'll illustrate the commands `Vector{Float64}(undef, n)` and `Matrix{Float64}(undef, n, m)` for creating vectors and matrices. They allow you to define a vector or matrix of a specified size without specifying its entries. We'll also illustrate the `copy` command for creating a copy of a matrix and how to make an identity matrix.

- `Vector{Float64}(undef, n)`
- `Matrix{Float64}(undef, n, m)`

- where `undef` is a Julia keyword to create a value that is “undefined”, and
- n defines the number of rows and m the number of columns.
- `B=copy(A)`
- How to create an identity matrix `Id = zeros(n,n) + I`.

Notice that when Julia creates a vector or matrix using the `Vector{Float64}(undef, n)` and `Matrix{Float64}(undef, n, m)` commands, it fills its entries with tiny numbers. How tiny? Well, the age of the universe in seconds is approximately 4.2×10^{16} . Hence, each second is approximately 2.3×10^{-17} -th of the age of the universe. Let’s say a typical ant that shows up in your home is half a centimeter long. Then that ant is 1.25×10^{-10} of the circumference of the earth (which is approximately 40,000 km). Julia is using numbers with 10^{-310} . I hope these comparisons were both helpful and interesting!

```
1 Vector{Float64}(undef, 5)
```

Output

```
5-element Vector{Float64}:
 0.0
 5.0e-324
 6.9094653095626e-310
 1.0e-323
 5.0e-324
```

```
1 Matrix{Float64}(undef, 3, 4)
```

Output

```
3×4 Matrix{Float64}:
 6.90947e-310  6.90947e-310  6.90947e-310  6.90947e-310
 6.90947e-310  6.90947e-310  6.90947e-310  6.9094e-310
 6.90947e-310  6.90947e-310  6.90947e-310  6.90947e-310
```

The Copy command is more important than you might think. The command `C=A` creates the matrix `C` and sets all of its entries equal to those of `A`. So far so good. It also links `C` to the matrix `A` so that any changes made to `C` are also made in `A` and vice versa. The command `B=copy(A)` creates an **independent** copy of `A`. Changes made in `B` do not show up in `A` and vice versa. Study the following examples and create a few of your own:

```
1 A=[1.9 2.0; 3.0 4.0]
2 C=A
```

Output

```
2×2 Matrix{Float64}:
 1.0  2.0
 3.0  4.0
```

```
1 C[1,2]=pi
2 C
```

Output

```
2×2 Matrix{Float64}:
 1.0  3.14159
 3.0  4.0
```

Now, look at the matrix `A`.

```
1 A
```

Output

```
2×2 Matrix{Float64}:
 1.0  3.14159
 3.0  4.0
```

Next, we illustrate the copy command. We make a change in B and show that it does not show up in A.

```
1 # Redefine A
2 A=[1.0 2.0; 3.0 4.0]
3 @show A
4 # Use the copy command
5 B=copy(A)
6 @show B
7 println("Now we change B[1,1]")
8 B[1,1]=sqrt(2)
9 @show B
10 println("and show there is no change in A")
11 A
```

Output

```
A = [1.0 2.0; 3.0 4.0]
B = [1.0 2.0; 3.0 4.0]
Now we change B[1,1]
B = [1.4142135623730951 2.0; 3.0 4.0]
and show there is no change in A

2×2 Matrix{Float64}:
 1.0  2.0
 3.0  4.0
```

Identity matrix: There is a special square matrix denoted I , or sometimes I_n to emphasize that it is an $n \times n$ matrix, which has ones on its diagonal and zeros everywhere else,

$$I_1 = [1], I_2 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, I_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, I_4 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \text{ etc.}$$

While MATLAB has the command `eye(n)` to create an identity matrix, Julia does not. Here is how you do it.

```
1 # how to make an identity matrix in Julia
2 using LinearAlgebra
3 n=4
4 Id = zeros(n, n) + I
5 # I is a special operator that when added to a square zero matrix
6 # produces an identity matrix of the appropriate size
```

Output

```
4×4 Matrix{Float64}:
 1.0  0.0  0.0  0.0
 0.0  1.0  0.0  0.0
 0.0  0.0  1.0  0.0
 0.0  0.0  0.0  1.0
```


The operator `I` is part of the `LinearAlgebra` package. If you have not already called `using LinearAlgebra`, you will not be able to use the identity operator `I`.

Fortunately, you cannot overwrite the identity operator `I` with a fixed identity matrix. That is why we used `Id = zeros(n, n) + I` in the above cell.

```
1 I = zeros(n, n) + I
```

Output

cannot assign a value to variable `LinearAlgebra.I` from module `Main`

Stacktrace:

```
[1] top-level scope
    @ In[15]:1
[2] eval
    @ ./boot.jl:360 [inlined]
[3] include_string(mapexpr::typeof(REPL.softscope), mod::Module,
code::String,
filename::String)
    @ Base ./loading.jl:1094
```

2.7 Length of a Vector and Size of a Matrix

If you want to know the **total number of elements** in any array, then use the `length` command as illustrated in the following examples.

```
1 using Random # Using an external package called Random
2 Random.seed!(1234) # Set the seed
3 V=randn(6) # generate a vector
4 display(V)
5 @show b=length(V);
```

Output

```
6-element Vector{Float64}:
 0.8673472019512456
-0.9017438158568171
-0.4944787535042339
-0.9029142938652416
 0.8644013132535154
 2.2118774995743475
```

```
b = length(V) = 6
```

```
1 A=rand(3, 4)
2 display(A)
3 @show length(A);
```

Output

```
3×4 Matrix{Float64}:
 0.200586  0.579672  0.066423  0.112486
 0.298614  0.648882  0.956753  0.276021
 0.246837  0.0109059  0.646691  0.651664
```

```
length(A) = 12
```

```
1 B=rand(2,2,2)
2 display(B)
3 @show length(B);
```

Output

```
2×2×2 Array{Float64, 3}:
[:, :, 1] =
 0.0566425  0.950498
 0.842714   0.96467

[:, :, 2] =
 0.945775  0.82116
 0.789904  0.0341601
length(B) = 8
```

Important: The `length` command is providing the total number of elements in any array! If you want the number of rows or columns, you must use the `size` command.

```
1 A=rand(3,4)
2 display(A)
3 @show nRows, nCols = size(A);
```

Output

```
3×4 Matrix{Float64}:
 0.449182  0.698356  0.372575  0.283401
 0.875096  0.365109  0.150508  0.404953
 0.0462887 0.302478  0.147329  0.499531
```

```
(nRows, nCols) = size(A) = (3, 4)
```

You can also obtain only the number of rows or just the number of columns.

```
1 A=rand(3,4)
2 display(A)
3 @show nRows = size(A,1);
```

Output

```
3×4 Matrix{Float64}:
 0.658815  0.59552  0.61816  0.0368842
 0.515627  0.292462 0.66426  0.643704
 0.260715  0.28858  0.753508 0.
```

```
nRows = size(A, 1) = 3 # note the 1
```

```
1 A=rand(3,4)
2 display(A)
3 @show nCols = size(A,2); # note the 2
```

Output

```
3×4 Matrix{Float64}:
 0.525057  0.082207  0.218177  0.932984
 0.61201  0.199058  0.362036  0.827263
 0.432577  0.576082  0.204728  0.0992992
```

```
nCols = size(A, 2) = 4
```

Length vs Size

- `length(X)` is the total number of elements in the array X . If X is a vector, then it is the usual length. If X is a matrix, then `length` returns the product of the number of rows and the number of columns.
- `nRows, nCols = size(A)` is best applied to matrices. It provides the number of rows and the number of columns. You can also use it as follows to return just the number of rows or just the number of columns,
- `nRows = size(A, 1)`
- `nCols = size(A, 2)`
- **Applying the `size` command to vectors is just asking for trouble. Please avoid doing this.**
- **Applying the `length` command to matrices is normally a mistake.** Be very careful when doing this.

2.8 Debugging

What is Debugging All About?

- Taken From the Pro Tip page at Georgia Tech for a graduate-level course: https://cse6040.gatech.edu/fa22/pro_tips.html
- Debugging is an iterative process where you must trace the undesired behavior back to the root cause. Sometimes it's simple; other times it can be frustratingly complex.
- Identify what is wrong. This needs to be as precise as possible. If you don't understand the error in the traceback - Google it!. Check the syntax, parameters, and inputs of any function calls.
- Identify where the wrong thing got set. This will usually be an assignment or a function call.
- Rinse and repeat until you have found and corrected the root cause.

At this point, in the course, we'll focus on showing some common errors so you get used to them and how Julia responds. Later, we can be more strategic in our thinking.

2.8.1 Errors with the `size` command

We use the `size` command on a 5×1 vector. The `size` command only works on matrices. You should use the `length` command here. The error message does tell you the error is on line #2 of your code.

```
1 v=[1; 2; 3; 4; 5]
2 nRows, nCols = size(v)
```

Output

```
BoundsError: attempt to access Tuple{Int64} at index [2]
```

Stacktrace:

```
[1] indexed_iterate(t::Tuple{Int64}, i::Int64, state::Int64)
  @ Base ./tuple.jl:86
[2] top-level scope
  @ In[112]:2
[3] eval
  @ ./boot.jl:360 [inlined]
```

```
[4] include_string(mapexpr::typeof(REPL.softscope), mod::Module, code::String, filename::String)
    @ Base ./loading.jl:1094
```

The `size` command does its best to do what you ask, so it returns something for the number of rows and an empty space where the number of columns is supposed to go. It does not throw an error if you do not attempt to assign a name to something that does not exist (here, the number of columns does not exist).

```
1 v=[1; 2; 3; 4; 5]
2 size(v)
```

Output

```
(5,)
```

We use the `size` command on a 1×5 “vector”. The `size` command works this time because Julia treats row vectors as matrices. You could also use the `length` command here.

```
1 v=[1 2 3 4 5]
2 @show nRows, nCols = size(v)
3 @show length(v);
```

Output

```
(nRows, nCols) = size(v) = (1, 5)
length(v) = 5
```

2.8.2 Bounds errors due to improper indexing

We next try to index into a non-existent entry of a matrix.

```
1 A = [1 2 3 4; 5 6 7 8.0]
2 k=5;
```

Output Nothing because we used a semicolon to suppress the output!

```
1 a = A[2, k]
```

Output

```
BoundsError: attempt to access 2x4 Matrix{Float64} at index [2, 5]
```

Stacktrace:

```
[1] getindex(::Matrix{Float64}, ::Int64, ::Int64)
    @ Base ./array.jl:802
[2] top-level scope
    @ In[119]:2
[3] eval
    @ ./boot.jl:360 [inlined]
[4] include_string(mapexpr::typeof(REPL.softscope), mod::Module,
code::String, filename::String)
    @ Base ./loading.jl:1094
```

The best thing to do when you see this is to check the size of the matrix and the value of the index. Yes, here we see easily that A is 2×4 and $k = 5$, but in a more complicated situation, you will make mistakes like this that are much less obvious.

By using the `show` command, we obtain enough information to understand our error.

```
1 @show size(A)
2 @show k
3 a = A[2, k]
```

Output

```
size(A) = (2, 4)
```

```
k = 5
```

```
BoundsError: attempt to access 2×4 Matrix{Float64} at index [2, 5]
```

```
Stacktrace:
```

```
[1] getindex(::Matrix{Float64}, ::Int64, ::Int64)
```

```
@ Base ./array.jl:802
```

```
[2] top-level scope
```

```
@ In[124]:3
```

```
[3] eval
```

```
@ ./boot.jl:360 [inlined]
```

```
[4] include_string(mapexpr::typeof(REPL.softscope), mod::Module,  
code::String, filename::String)
```

```
@ Base ./loading.jl:1094
```


Chapter 3

Julia Lab 3: For Loops and 1-Element Vectors

Learning Objectives

- What is and how to use a `for` loop
- Managing the fact that in Julia, 1×1 matrices and 1-element vectors are not scalar variables
- Debugging, or finding and fixing errors in your code

Outcomes

- How to write basic for loops
- How to control the counter
- Big sums and for loops go hand in hand
- Extracting real values from 1-element vectors
- Back substitution at scale, with a `for` loop
- (Optional Read) While loops and multiplicative persistence, an open-problem in math
- (Optional Read) Nesting multiple `for` loops

Either download Lab3 from our Canvas site or open up a Jupyter notebook so that you can enter code as we go. It is suggested that you have line numbering toggled on.

A `for` loop specifies a block of code that is to be repeatedly executed a fixed (finite) number of times. Various keywords are used to specify this statement: descendants of ALGOL use **for**, while descendants of Fortran use **do**; see https://en.wikipedia.org/wiki/For_loop#Traditional_for-loops. Once you get the knack of `for` loops, you will begin to feel like you are really doing programming. Hence, it's important to get this right.

Remark 3.1 *The syntax for writing a `for` loop in Julia is quite different from that in C++ and nearly identical to MATLAB. Make a note in your Google Doc of Programming hints and you'll be fine. If you try to memorize everything without good notes, you'll be frustrated. **Wotcha gonna do? Yes. Make that Google Doc;; see Chapter 0.4.***

3.1 Basic For Loops

A basic `for` loop in Julia looks like this

```
for k = k_start : k_end

    line 1 of code
    line 2 of code
    .
    .
    .
    line n of code

end
```

The key words `for` and `end` indicate the beginning and end, respectively of the `for` loop. `k` is called the **counter**. It starts at `k_start` and ends at `k_end`. The code between the `for` and `end`, indicated here by `line1` through `linen`, constitute the “loop” and, if `k_start ≤ k_end`, the loop is executed for all $k \in \{k_{start}, k_{start} + 1, \dots, k_{end} - 1, k_{end}\}$. By default, the counter `k` is **incremented** by `+1` each time the loop is completed. We'll show later how to increment or decrement the counter by values other than `+1` (e.g., `-1` or `+5`).

Remark 3.2 *By its very nature, a `for` loop will execute a finite number of times*

$$k_{\max} = 1 + |k_{\text{end}} - k_{\text{start}}|.$$

It cannot get “stuck” and loop forever. Later, we'll introduce a `while` loop, which will loop until its exit condition is met. A `while` loop can loop forever.

In the following, we present a series of examples that show the ease with which repetitive operations can be automated with a `for` loop

```
1 # Compute the sum from 6 to 15
2 x = 6 + 7 + 8 + 9 + 10 + 11 + 12 + 13 + 14 + 15
```

Output

105

```
1 # Here is how to do the same with a for loop
2 # We printout the intermediate calculations
3 # to highlight what is happening at each step
4 #
5 x=0 # initialize x to zero
6 for k = 6 : 15 # note the colon separating k_start from k_end
7     x = x + k # add k to x
8     @show k, x
9 end
```

Output


```
(k, x) = (6, 6)
(k, x) = (7, 13)
(k, x) = (8, 21)
(k, x) = (9, 30)
(k, x) = (10, 40)
(k, x) = (11, 51)
(k, x) = (12, 63)
(k, x) = (13, 76)
(k, x) = (14, 90)
(k, x) = (15, 105)
```

A common error is to fail to initialize the variable that is holding the sum.

```
1 # The goal is to add up the integers from 3 to 8 and store the answer in z
2 #
3 # A common error is to forget to initialize the variable
4 # that is holding the result of doing the for loop
5 # In our case, that variable is z
6 for k = 3 : 8
7     z = z + k # add k to z
8     @show z
9 end
10 # The for loop says to take the current value of z and add k
11 # to it everytime you go through the loop
12 # However, the first time we execute the code between the **for** and
13 # the **end**, the variable **z**
14 # is undefined and hence we cannot perform the addition **z + k**
```

Output

UndefVarError: z not defined

Stacktrace:

```
[1] top-level scope
  @ ./In[5]:8
[2] eval
  @ ./boot.jl:360 [inlined]
[3] include_string(mapexpr::typeof(REPL.softscope), mod::Module, code::String,
filename::String)
  @ Base ./loading.jl:1094
```

Adding up the numbers from 6 to 15 is not so bad to do by hand. How about the numbers from 1 to 100? You can read this article to see how the mathematician Gauss did it as a schoolboy <http://mathcentral.uregina.ca/qq/database/qq.02.06/jo1.html>. Below, we do it in Julia.

```
1 mySum=0
2 for i = 1 : 100
3     mySum = mySum + i # add i to mySum
4 end
5 mySum
```

Output

5050

3.2 Incrementing the Counter by Something other than 1

What if we wanted to add up every third number between 1 and 100? We do this by properly setting how the counter increments through each loop.

```

1 mySum=0
2 # note the pattern counter_start:counter_increment:counter_end
3 # each separated by a colon :
4 for i = 1:3:100
5     mySum = mySum + i # add i to mySum
6 end
7 mySum

```

Output

1717

The pattern is **counter_start:counter_increment:counter_end**, each separated by a colon. We use this to add up every 23rd number between 1 and 100.

```

1 mySum=0
2 for k = 1:23:100 # note the pattern k_start:k_increment:k_end
3     mySum = mySum + k # add k to mySum
4     @show k, mySum
5 end
6 mySum

```

Output

```

(k, mySum) = (1, 1)
(k, mySum) = (24, 25)
(k, mySum) = (47, 72)
(k, mySum) = (70, 142)
(k, mySum) = (93, 235)

```

235

Note that the next value of k would have been $93 + 23 > 100$ and hence the loop stopped at $k = 93$.

We can also decrement the counter. Now, decrementing the counter is the same as incrementing by a negative number, which is what we do. Let's now add up the even numbers between 1 and 16, going backwards!

```

1 # Example: compute the sum mySum = 16 + 14 + ... + 2
2 # We wrote it backwards so you can see the order of the computations
3 #
4 mySum = 0
5 for i = 16:(-2):1 # the parentheses are optional, but make the code more clear
6     mySum = mySum + i
7     @show i, mySum
8 end

```

Output

```

(i, mySum) = (16, 16)
(i, mySum) = (14, 30)
(i, mySum) = (12, 42)
(i, mySum) = (10, 52)
(i, mySum) = (8, 60)
(i, mySum) = (6, 66)
(i, mySum) = (4, 70)
(i, mySum) = (2, 72)

```

3.3 Summation Symbol in Math Equals a For Loop in Programming

The summation symbol is covered in Chapter 4.1 of the ROB 101 textbook. You will notice straightaway that the summation symbol in math is a `for loop` in programming! The only difference is we have to remember to initialize the sum at zero when we do a `for loop`.

The summation symbol is a for loop and vice versa

- $1 + 2 + 3 = \sum_{k=1}^3 k$. Here, k is called an index and \sum is the symbol for **sum or summation**. $\sum_{k=1}$ gives the initial value of the index in the sum, which in this case is 1, and \sum^3 gives the final value of the index in the sum, which in this case, is 3. Unless indicated otherwise, the index always increments by one in a sum, just like a counter defaults to incrementing by one in a `for loop`.
- The “index” in a summation symbol is equivalent to a “counter” in a `for loop`. After a while, you begin to forget which is which!
- $1 + 2 + \dots + n = \sum_{k=1}^n k$. We note that $\sum_{k=1}$ defines the initial value of the index in the sum to be 1, and \sum^n defines the final value of the index in the sum to be n .

```
1 n=20
2 mySum = 0 # Initialize the sum
3 for k = 1:n
4     mySum = mySum + k
5 end
6 mySum
```

Output

210

- Changing the name of the index from k to i , for example, does not change anything

$$\sum_{i=1}^n i = 1 + 2 + \dots + n = \sum_{k=1}^n k$$

- $\sum_{k=1}^n k^2 := (1)^2 + (2)^2 + (3)^2 + \dots + (n)^2$.
- $a_1 + a_2 + a_3 = \sum_{i=1}^3 a_i$
- $a_7 + a_8 + a_9 = \sum_{i=7}^9 a_i$. We note that the index is i , the initial value of the index is 7, and final value of the index is 9.
- $a_1 + a_1 + \dots + a_n = \sum_{i=1}^n a_i$

```
1 n=20
2 mySum = 0 # Initialize the sum
3 for i = 0:n # define the counter and its initial and final values
4     ai = 1/factorial(i) # Define ai
5     mySum = mySum + ai # add stuff up
6 end
7 @show exp(1) - mySum
8 mySum
```

Output

exp(1) - mySum = -4.440892098500626e-16

2.7182818284590455

Remark 3.3 (Optional Read) There is also a product symbol in math. It looks like this \prod (an upper case π). $\prod_{i=1}^n a_i = a_1 \cdot a_2 \cdot a_3 \cdots a_n$, the product of all the terms. In code, it works like this.

```

1 # product
2 n=10
3 prod = 1.0 # Initialize the product at 1.0 and not zero
4 for i = 1:n
5     ai = i^2      # you can put anything here
6     prod = prod * ai
7 end
8 prod

```

Output

```
1.316818944e13
```

3.4 A Brief Intro to Matrix-Vector Multiplication and the Unnerving Fact that 1-Element Vectors in Julia are not Scalars

Let $a^{\text{row}} = [a_1 \ a_2 \ \cdots \ a_k]$ be a row vector with k elements and let $b^{\text{col}} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_k \end{bmatrix}$ be a column vector **with the same number of**

elements as a^{row} . The product of a^{row} and b^{col} is defined as

$$a^{\text{row}} \cdot b^{\text{col}} := \sum_{i=1}^k a_i b_i := a_1 b_1 + a_2 b_2 + \cdots + a_k b_k. \quad (3.1)$$

```

1 aRow = [1.0 pi 5 sqrt(2)]
2 bCol = [21; 0; 11; sqrt(2)]
3 MVProd1 = 0.0 # Matrix-Vector Product
4 for i = 1:length(aRow)
5     MVProd1 = MVProd1 + aRow[i]*bCol[i]
6 end
7 @show typeof(MVProd1)
8 MVProd1

```

Output

```
typeof(MVProd1) = Float64
```

```
78.0
```

Because the variable `MVProd1` is of Type `Float64`, we can use it to assign a component of a vector, like this,

```

1 x=zeros(3,1)
2 x[2]=MVProd1 # Assign second component of x
3 x

```

Output

```
3×1 Matrix{Float64}:
 0.0
 78.0
 0.0
```

Julia has a built in multiplication command that makes short work of multiplying a $1 \times n$ matrix times an n -element vector. Let's use it and try to assign the result to a vector as we did above.

```
1 # Much easier and shorter to type
2 aRow=[1.0 pi 5 sqrt(2)]
3 bCol=[21; 0; 11; sqrt(2)]
4 MVProd2 = aRow*bCol # Multiplication command in Julia
5 @show MVProd2
6 @show typeof(MVProd2)
7 x=zeros(3,1)
8 x[2]=MVProd2 # Attempt to assign the second entry of x
9 x
```

Output

```
MVProd2 = [78.0]
typeof(MVProd2) = Vector{Float64}
```

```
MethodError: Cannot `convert` an object of type Vector{Float64} to an object of type Float64
```

Closest candidates are:

```
convert(::Type{T}, ::T) where T<:Number at number.jl:6
convert(::Type{T}, ::Number) where T<:Number at number.jl:7
convert(::Type{T}, ::Base.TwicePrecision) where T<:Number at twiceprecision.jl:250
...
```

Stacktrace:

```
[1] setindex!(A::Matrix{Float64}, x::Vector{Float64}, i1::Int64)
  @ Base ./array.jl:839
[2] top-level scope
  @ In[57]:8
[3] eval
  @ ./boot.jl:360 [inlined]
[4] include_string(mapexpr::typeof(REPL.softscope), mod::Module, code::String,
filename::String)
  @ Base ./loading.jl:1094
```

What went wrong?

- Here is what Julia told you: `MethodError: Cannot `convert` an object of type Vector{Float64} to an object of type Float64`
- The above is Julia-speak for a TYPE mismatch. In Julia, the product of a $1 \times n$ matrix with an n -element vector is a 1-element `Vector{Float64}`.
- You can see this when Julia's output, `MVProd2 = [78.0]`, where you notice the square brackets around the number 78.0. This signals the variable is either a matrix or a vector. In any case, it is not a regular number. Julia objects vociferously when you attempt the assignment `x[2]=MVProd2`.
- The solution is quite easy. You need to extract the number from the vector like this, `x[2]=MVProd2[1]`
- We illustrate below

```
1 # Much easier and shorter to type
2 aRow=[1.0 pi 5 sqrt(2)]
3 bCol=[21; 0; 11; sqrt(2)]
4 MVProd2 = aRow*bCol # Multiplication command in Julia
5 @show MVProd2
```

```

6 @show typeof(MVProd2)
7 @show typeof(MVProd2[1])
8 x=zeros(3,1)
9 x[2]=MVProd2[1] # extract the value as a number
10                # when assigning as an entry of a vector
11 x

```

Output

```

MVProd2 = [78.0]
typeof(MVProd2) = Vector{Float64}

typeof(MVProd2[1]) = Float64

3×1 Matrix{Float64}:
 0.0
 78.0
 0.0

```

There is a useful shortcut in Julia that is based on some graduate level mathematics as explained below in Chapter 3.10. We illustrate it now and then use it in the next section. It's Ok to use the shortcut without understanding the underlying mathematics (adjoint vectors in dual spaces). It saddens me that we have to even deal with this because we do not even teach adjoint vectors in ROB 501 Mathematics for Robotics! But hey, with a bit of practice, we can learn anything.

```

1 # Shortcut and probably hard to understand
2 aCol=[1.0; pi; 5; sqrt(2)] # I'm a column vector this time
3 bCol=[21; 0; 11; sqrt(2)] # I always was a column vector
4 MVProd3 = (aCol')*bCol # parentheses are not necessary,
5                    # but note the apostrophe
6 @show typeof(MVProd3)
7 MVProd3

```

Output

```

typeof(MVProd3) = Float64

78.0

```

The key point is that with the shortcut, no extraction of a value is required. When you “transpose” a column vector to create an “adjoint vector”, which acts like a special kind of row vector, the multiplication operator directly produces a number—Float64—without enclosing it in a vector. Is your head spinning? Then you can use `aRow*bCol` and extract values.

3.5 Back Substitution as a Backward For Loop

Admittedly, doing backward summations does not make a whole lot of sense. In Chapter 3 of our textbook, and in `juliahw2`, we'll encounter upper triangular systems of equations, which are equations that look like the one below, except in HW, you'll have 100 unknowns!

$$\begin{aligned}
 x_1 + 3x_2 + 2x_3 &= 6 \\
 2x_2 + x_3 &= -2 \\
 3x_3 &= 4,
 \end{aligned}
 \iff
 \underbrace{\begin{bmatrix} 1 & 3 & 2 \\ 0 & 2 & 1 \\ 0 & 0 & 3 \end{bmatrix}}_A
 \underbrace{\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}}_x
 =
 \underbrace{\begin{bmatrix} 6 \\ -2 \\ 4 \end{bmatrix}}_b.
 \tag{3.2}$$

```

1 @show b = [6;-2;4] # create a column vector
2 # @show b = [6 -2 4] # Would make a row vector
3 @show typeof(b)
4 A = [1 3 2; 0 2 1; 0 0 3]

```

Output

```
b = [6; -2; 4] = [6, -2, 4]
typeof(b) = Vector{Int64}
```

```
3×3 Matrix{Int64}:
 1  3  2
 0  2  1
 0  0  3
```

Note that we added the command `@show` because Julia only prints out the result of the last calculation in the cell. We included the command `typeof` to make it clear that `b` is a column vector. If we had created a row vector, Julia would have said the Type was `typeof(b) = Matrix{Int64}`. You can use the above code and uncomment the second definition of `b` to verify this statement. When you are just learning a programming language, it never hurts to include extra “print” or “show” statements.

Here is a line by line solution of (3.2).

```
1 @show x=zeros(3,1)
2 @show x[3] = b[3]/A[3,3]
3 @show x[2] = (b[2]-A[2,3]*x[3])
4 @show x[1] = (b[1]-A[1,2:3]'*x[2:3])/A[1,1]
5 x
```

Output

```
x = zeros(3, 1) = [0.0; 0.0; 0.0]
x[3] = b[3] / A[3, 3] = 1.3333333333333333
x[2] = b[2] - A[2, 3] * x[3] = -3.3333333333333333
x[1] = (b[1] - (A[1, 2:3])' * x[2:3]) / A[1, 1] = 13.333333333333334
```

```
3×1 Matrix{Float64}:
 13.333333333333334
 -3.333333333333333
  1.3333333333333333
```

Here is an alternative line-by-line solution

```
1 @show x=zeros(3,1)
2 @show x[3] = b[3]/A[3,3]
3 @show x[2] = (b[2]-A[2,2+1:end]'*x[2+1:end])/A[2,2]
4 @show x[1] = (b[1]-A[1,1+1:end]'*x[1+1:end])/A[1,1]
5 x
```

Output

```
x = zeros(3, 1) = [0.0; 0.0; 0.0]
x[3] = b[3] / A[3, 3] = 1.3333333333333333
x[2] = (b[2] - (A[2, 2 + 1:end])' * x[2 + 1:end]) / A[2, 2] = -1.6666666666666665
x[1] = (b[1] - (A[1, 1 + 1:end])' * x[1 + 1:end]) / A[1, 1] = 8.333333333333334
```

```
3×1 Matrix{Float64}:
 8.333333333333334
 -1.6666666666666665
  1.3333333333333333
```

In case that was too fast, here is a blow by blow analysis of what is happening.

```
1 println("Initialize x to zero")
2 @show x=zeros(3,1)
```

```

3 println (" ")
4 println ("Show elements contributing to x[3] ")
5 @show x[3] = b[3]/A[3,3]
6 println (" ")
7 println ("Show elements contributing to x[2] ")
8 @show A[2,2+1:end]'
9 @show x[2+1:end]
10 @show temp = A[2,2+1:end]'*x[2+1:end]
11 @show x[2] = (b[2]-temp)/A[2,2]
12 println (" ")
13 println ("Show elements contributing to x[1] ")
14 @show A[1,1+1:end]'
15 @show x[1+1:end]
16 @show temp = A[1,1+1:end]'*x[1+1:end]
17 @show x[1] = (b[1]-temp)/A[1,1]
18 x

```

Output

Initialize x to zero

```
x = zeros(3, 1) = [0.0; 0.0; 0.0]
```

Show elements contributing to x[3]

```
x[3] = b[3] / A[3, 3] = 1.3333333333333333
```

Show elements contributing to x[2]

```
(A[2, 2 + 1:end])' = [1]
```

```
x[2 + 1:end] = [1.3333333333333333]
```

```
temp = (A[2, 2 + 1:end])' * x[2 + 1:end] = 1.3333333333333333
```

```
x[2] = (b[2] - temp) / A[2, 2] = -1.6666666666666665
```

Show elements contributing to x[1]

```
(A[1, 1 + 1:end])' = [3 2]
```

```
x[1 + 1:end] = [-1.6666666666666665, 1.3333333333333333]
```

```
temp = (A[1, 1 + 1:end])' * x[1 + 1:end] = -2.3333333333333335
```

```
x[1] = (b[1] - temp) / A[1, 1] = 8.333333333333334
```

We now show how to implement the above cells with a for loop.

```

1 x=zeros(3,1)
2 x[3] = b[3]/A[3,3]
3 for i = 2:-1:1
4     x[i] = (b[i]-A[i,i+1:end]'*x[i+1:end])/A[i,i]
5 end
6 x

```

Output

3×1 Matrix{Float64}:

```
8.333333333333334
```

```
-1.6666666666666665
```

```
1.3333333333333333
```

You can totally avoid using the adjoint vector if you wish. You then need to extract the element from the resulting 1-element vector.

```

1 # Same code without using the adjoint operator
2 # A few comments added
3 x=zeros(3,1)

```



```

4 x[3] = b[3]/A[3,3]
5 for i = 2:(-1):1
6     temp = A[i:i,i+1:end]*x[i+1:end] # row times column
7                                         # note the i:i
8     @show temp # 1-element vector
9     @show temp[1] # Float64
10    #
11    x[i] = (b[i]-temp[1])/A[i,i] # We use temp[1] because
12                                         # temp only has one element
13 end
14 x

```

Output

```

temp = [1.3333333333333333]
temp[1] = 1.3333333333333333
temp = [-2.3333333333333335]
temp[1] = -2.3333333333333335

```

```

3×1 Matrix{Float64}:
 8.333333333333334
-1.6666666666666665
 1.3333333333333333

```

x is a solution of $Ax = b$ if, and only if, $Ax - b = 0_{3 \times 1}$.

```

1 # Let's see if we really do have a solution
2 A*x-b

```

Output

```

3×1 Matrix{Float64}:
 0.0
 2.220446049250313e-16
 0.0

```

The above looks pretty close to zero. We'll take it! We agree that solving an equation with three variables does not look so impressive. Let's create one that has six variables.

```

1 A=[
2 -0.991273  -0.850184  -1.06297   0.609128  0.498376  0.411742
3  -0.0       0.592798  -0.509626  -0.547334  -0.670348  -0.826093
4   0.0       -0.0       0.673963   0.24726   -0.52135   0.680463
5  -0.0       -0.0       0.0        -0.860824  -0.71363   -0.322208
6  -0.0       0.0       -0.0       0.0        0.249046   0.373825
7   0.0       -0.0       -0.0       0.0        0.0        -0.044526
8 ]
9 #
10 b=[0.6849753810315695
11 0.7387064229251636
12 0.8252920694637993
13 0.2707345660171885
14 0.254653180382145
15 0.1555903546558295]

```

Output

```

6-element Vector{Float64}:
 0.6849753810315695

```

```
0.7387064229251636
0.8252920694637993
0.2707345660171885
0.254653180382145
0.1555903546558295
```

We make a few modest improvements to the code so that it works for “any” upper triangular square system of linear equations.

```
1 N=length(b)
2 x=zeros(N,1) # Makes x an N x 1 matrix of zeros
3 x[N] = b[N]/A[N,N]
4 for i = (N-1):-1:1
5     x[i] = (b[i]-A[i,i+1:end]'*x[i+1:end])/A[i,i]
6 end
7 x
```

Output

```
6×1 Matrix{Float64}:
-21.381592956092216
 9.163388707678338
11.142804824616672
-4.202499623521532
 6.267662793927989
-3.494370809321059
```

```
1 A*x-b
```

Output

```
6×1 Matrix{Float64}:
 4.9960036108132044e-15
-1.1102230246251565e-16
-4.440892098500626e-16
 1.1102230246251565e-16
-1.1102230246251565e-16
 0.0
```

And once again, that is a very good approximation to a vector of zeros.

In case you are wondering why x is a `6×1 Matrix{Float64}` instead of a 6-element `Vector{Float64}`, it’s because we used the `zeros` command to create the vector x . Below is a way to force x to be a Julia column vector, though we emphasize that in most cases, like this one, we just do not care!

```
1 N=length(b)
2 x=0.0*b # x has the same TYPE and SIZE as b
3 x[N] = b[N]/A[N,N]
4 for i = (N-1):-1:1
5     x[i] = (b[i]-A[i,i+1:end]'*x[i+1:end])/A[i,i]
6 end
7 x
```

Output

```
6-element Vector{Float64}:
-21.381592956092216
 9.163388707678338
11.142804824616672
-4.202499623521532
```

```
6.267662793927989
-3.494370809321059
```

Remark 3.4 (Optional Read) Can you figure out why this works?

```
1 N=length(b)
2 x=0.0*b # x has the same TYPE and SIZE as b
3 for i = N:-1:1
4     x[i] = (b[i]-A[i,i+1:end]'*x[i+1:end])/A[i,i]
5 end
6 x
```

Output

```
6-element Vector{Float64}:
-21.381592956092216
 9.163388707678338
11.142804824616672
-4.202499623521532
 6.267662793927989
-3.494370809321059
```

We're exploiting the fact that (i) $A[N, N+1:end]'$ is an EMPTY row (adjoint) vector, (ii) $x[N+1:end]$ is an EMPTY column vector, and (iii) in Julia, their product is zero!

3.6 Hidden Variables in For Loops (aka Scope of a Variable)

In Julia, variables created within a for loop are “hidden from you”. Here are illustrations of what we mean.

```
1 # Hidden variables in for loops
2 x=0.0
3 for k = 1:10
4     y=k^2
5     x = x+y
6 end
7
8 @show y
```

Output

```
UndefVarError: y not defined
```

Stacktrace:

```
[1] top-level scope
@ show.jl:955
[2] eval
@ ./boot.jl:360 [inlined]
[3] include_string(mapexpr::typeof(REPL.softscope), mod::Module, code::String,
filename::String)
@ Base ./loading.jl:1094
```

Even the counter used in the for loop is hidden from you.

```
1 x=0.0
2 for k = 1:10
3     y=k^2
4     x = x+y
5 end
6
```

```
7 @show k
```

Output

```
UndefVarError: k not defined
```

```
Stacktrace:
```

```
[1] top-level scope
    @ show.jl:955
[2] eval
    @ ./boot.jl:360 [inlined]
[3] include_string(mapexpr::typeof(REPL.softscope), mod::Module, code::String,
filename::String)
    @ Base ./loading.jl:1094
```

However, variables defined outside of the `for` loop are available to you, except for the counter used in the loop.

The variable x is defined outside of the loop, hence we can see its final value when the loop is finished.

```
1 x=0.0
2 for k = 1:10
3     y=k^2
4     x = x+y
5 end
6
7 x
```

Output

```
385.0
```

Here, we initialize the counter k to zero and the intermediate variable y to not a number, NaN (we could have used any value available in Julia).

```
1 x=0.0; k=0; y=NaN
2 for k = 1:10
3     y=k^2
4     x = x+y
5 end
6
7 # Show the final values of these quantities
8 [x y k]
```

Output

```
1×3 Matrix{Float64}:
 385.0  100.0  0.0
```

What we see are the final values for x and y and our defined initial value for k . However, if use the `@show` command within the loop, then we can see how k is evolving.

```
1 x=0.0; k=0; y=NaN
2 for k = 1:5
3     y=k^2
4     x = x+y
5     @show k
6 end
```

Output

```
k = 1
k = 2
k = 3
k = 4
k = 5
```

Scope of a Variable

The technical term for what we are illustrating above is the “Scope of a Variable”. You can learn more in the Julia documentation <https://docs.julialang.org/en/v1/manual/variables-and-scoping>. For ROB 101, the little we have presented should cover your needs.

3.7 Debugging

The main strategy we emphasize here is to **locate the line on which the error occurs and then use a bunch of @show commands to see what is wrong**. This simple strategy can take you a long ways!

3.7.1 Dimension Mismatch

To add or multiply matrices, they must have the correct sizes.

```
1 A=[1 2 3; 4 5 6]
2 B=[7 8; 9 10]
3 #
4 A+B
```

Output

```
DimensionMismatch("dimensions must match: a has dims (Base.OneTo(2), Base.OneTo(3)),
b has dims (Base.OneTo(2), Base.OneTo(2)), mismatch at 2")
```

Stacktrace:

```
[1] promote_shape
  @ ./indices.jl:178 [inlined]
[2] promote_shape(a::Matrix{Int64}, b::Matrix{Int64})
  @ Base ./indices.jl:169
[3] +(A::Matrix{Int64}, Bs::Matrix{Int64})
  @ Base ./arraymath.jl:45
[4] top-level scope
  @ In[1]:4
[5] eval
  @ ./boot.jl:360 [inlined]
[6] include_string(mapexpr::typeof(REPL.softscope), mod::Module, code::String,
filename::String)
  @ Base ./loading.jl:1094
```

We add some appropriate show commands.

```
1 A=[1 2 3; 4 5 6]
2 B=[7 8; 9 10]
3 @show size(A)
4 @show size(B)
5 #
6 A+B
```

Output

```

size(A) = (2, 3)
size(B) = (2, 2)
DimensionMismatch("dimensions must match: a has dims (Base.OneTo(2), Base.OneTo(3)),
b has dims (Base.OneTo(2), Base.OneTo(2)), mismatch at 2")

```

OK, we now see the problem: A and B have different sizes.

Below, please note that Julia’s error message assumes that the generic matrix product is $A * B$, even when we have formed $B * A$. When you read the error message below, “DimensionMismatch(“matrix A has dimensions (2,3), matrix B has dimensions (2,2)”), **you must realize that A refers to the first matrix in the matrix product and B to the second matrix.** We see clearly that B is 2×3 (2,3) while A is 2×2 (2,2).

```

1 B=[1 2 3; 4 5 6]
2 A=[7 8; 9 10]
3 #
4 B*A

```

Output

```
DimensionMismatch("matrix A has dimensions (2,3), matrix B has dimensions (2,2)")
```

Stacktrace:

```

 [1] _generic_matmatmul!(C::Matrix{Int64}, tA::Char, tB::Char, A::Matrix{Int64},
      B::Matrix{Int64}, _add::LinearAlgebra.MulAddMul{true, true, Bool, Bool})
      @ LinearAlgebra
 /buildworker/worker/package_linux64/build/usr/share/julia/stdlib/v1.6/
LinearAlgebra/src/matmul.jl:814
 [2] generic_matmatmul!(C::Matrix{Int64}, tA::Char, tB::Char, A::Matrix{Int64},
      B::Matrix{Int64}, _add::LinearAlgebra.MulAddMul{true, true, Bool, Bool})
      @ LinearAlgebra /buildworker/worker/package_linux64/build/usr/share/julia/stdlib/
v1.6/LinearAlgebra/src/matmul.jl:802
 [3] mul!
      @ /buildworker/worker/package_linux64/build/usr/share/julia/stdlib/v1.6/LinearAlgebra
 /src/matmul.jl:302 [inlined]
 [4] mul!
      @ /buildworker/worker/package_linux64/build/usr/share/julia/stdlib/v1.6/
LinearAlgebra/src/matmul.jl:275 [inlined]
 [5] *(A::Matrix{Int64}, B::Matrix{Int64})
      @ LinearAlgebra /buildworker/worker/package_linux64/build/usr/share/julia/stdlib/v1.6/
LinearAlgebra/src/matmul.jl:153
 [6] top-level scope
      @ In[2]:4
 [7] eval
      @ ./boot.jl:360 [inlined]
 [8] include_string(mapexpr::typeof(REPL.softscope), mod::Module,
      code::String, filename::String)
      @ Base ./loading.jl:1094

```

```

1 B=[1 2 3; 4 5 6]
2 A=[7 8; 9 10]
3 @show size(A)
4 @show size(B)
5 #
6 B*A

```

Output

```
size(A) = (2, 2)
```

```
size(B) = (2, 3)
DimensionMismatch("matrix A has dimensions (2,3), matrix B has dimensions (2,2)")
```

3.7.2 More on 1×1 Matrices are not Numbers

After providing some data, we create what looks like a perfectly fine `for` loop to implement back substitution. Yet, it fails. What's up?

```
1 A=[
2 -0.991273  -0.850184  -1.06297    0.609128  0.498376  0.411742
3 -0.0       0.592798  -0.509626  -0.547334  -0.670348  -0.826093
4 0.0       -0.0       0.673963   0.24726   -0.52135   0.680463
5 -0.0      -0.0       0.0       -0.860824  -0.71363   -0.322208
6 -0.0      0.0       -0.0      0.0       0.249046   0.373825
7 0.0       -0.0      -0.0      0.0       0.0        -0.044526
8 ]
9 #
10 b=[0.6849753810315695
11 0.7387064229251636
12 0.8252920694637993
13 0.2707345660171885
14 0.254653180382145
15 0.1555903546558295];
```

Output Nothing due to the semicolon.

```
1 N=length(b)
2 x=0.0*b # x has the same TYPE and SIZE as b
3 x[N] = b[N]/A[N,N]
4 for i = (N-1):-1:1
5     x[i] = (b[i]-A[i:i,i+1:end]*x[i+1:end])/A[i,i]
6 end
7 x
```

Output

```
MethodError: no method matching -(::Float64, ::Vector{Float64})
For element-wise subtraction, use broadcasting with dot syntax: scalar .- array
Closest candidates are:
 - (::SparseArrays.AbstractSparseMatrixCSC, ::Array) at /buildworker/worker/package_linux64/build/usr/share/julia/stdlib/v1.6/SparseArrays/src/sparsematrix.jl:1746
 - (::AbstractArray, ::AbstractArray) at arraymath.jl:37
 - (::Float64) at float.jl:320
 ...
```

Stacktrace:

```
[1] top-level scope
@ ./In[9]:5
[2] eval
@ ./boot.jl:360 [inlined]
[3] include_string(mapexpr::typeof(REPL.softscope), mod::Module,
code::String, filename::String)
@ Base ./loading.jl:1094
```

We see that the error is on line 5. Hence, we place a bunch of `show` commands to see if we can figure this out. We note that `A[i:i, i+1:end]` extracts out the i -th row of A and keeps it as a “row vector” in the form of a $1 \times 6 - i$ matrix. When it multiplies the column vector `x[i+1:end]`, we get a 1×1 matrix in Julia.

```

1 N=length(b)
2 x=0.0*b # x has the same TYPE and SIZE as b
3 x[N] = b[N]/A[N,N]
4 for i = (N-1):-1:1
5     @show i
6     @show b[i]
7     @show A[i:i,i+1:end]*x[i+1:end]
8     @show A[i,i]
9     x[i] = (b[i]-A[i:i,i+1:end]*x[i+1:end])/A[i,i]
10 end
11 x

```

Output

```

i = 5
b[i] = 0.254653180382145
A[i:i, i + 1:end] * x[i + 1:end] = [-1.3062831677944449]
A[i, i] = 0.249046
MethodError: no method matching - (::Float64, ::Vector{Float64})
For element-wise subtraction, use broadcasting with dot syntax: scalar .- array
Closest candidates are:
 - (::SparseArrays.AbstractSparseMatrixCSC, ::Array) at
   /buildworker/worker/package_linux64/build/usr/share/julia/stdlib/v1.6/
   SparseArrays/src/sparsematrix.jl:1746
 - (::AbstractArray, ::AbstractArray) at arraymath.jl:37
 - (::Float64) at float.jl:320
 ...

```

Stacktrace:

```

[1] top-level scope
   @ ./In[10]:9
[2] eval
   @ ./boot.jl:360 [inlined]
[3] include_string(mapexpr::typeof(REPL.softscope), mod::Module,
code::String, filename::String)
   @ Base ./loading.jl:1094

```

Key things to notice

- $b[i] = 0.254653180382145$ is a number
- $A[i:i, i + 1:end] * x[i + 1:end] = [-1.3062831677944449]$ results in a 1×1 matrix. **We can tell by the square brackets around the number!**
- We cannot subtract a matrix from a number.
- The fix is to extract the number from the matrix via
- $(A[i:i, i + 1:end] * x[i + 1:end])[1]$

```

1 N=length(b)
2 x=0.0*b # x has the same TYPE and SIZE as b
3 x[N] = b[N]/A[N,N]
4 for i = (N-1):-1:1
5     @show A[i:i,i+1:end]*x[i+1:end]
6     @show (A[i:i,i+1:end]*x[i+1:end])[1]
7     x[i] = ( b[i] - (A[i:i,i+1:end]*x[i+1:end])[1] ) / A[i,i]
8 end
9 x

```


Output

```
A[i:i, i + 1:end] * x[i + 1:end] = [-1.3062831677944449]
(A[i:i, i + 1:end] * x[i + 1:end])[1] = -1.3062831677944449
A[i:i, i + 1:end] * x[i + 1:end] = [-3.3468779699011106]
(A[i:i, i + 1:end] * x[i + 1:end])[1] = -3.3468779699011106
A[i:i, i + 1:end] * x[i + 1:end] = [-6.684546098549327]
(A[i:i, i + 1:end] * x[i + 1:end])[1] = -6.684546098549327
A[i:i, i + 1:end] * x[i + 1:end] = [-4.69333207620914]
(A[i:i, i + 1:end] * x[i + 1:end])[1] = -4.69333207620914
A[i:i, i + 1:end] * x[i + 1:end] = [-20.51002041333283]
(A[i:i, i + 1:end] * x[i + 1:end])[1] = -20.51002041333283
6-element Vector{Float64}:
 -21.381592956092216
   9.163388707678338
  11.142804824616672
 -4.202499623521532
   6.267662793927989
 -3.494370809321059
```

OK, we did not need to print out the intermediate results all six times, but hopefully, this drives home the point: use the `@show` command to help you figure out what is going wrong. **Computers do what you tell them to do, not what you want them to do.** What is perfectly fine code in one language (here, the command `"A[i:i,i+1:end]*x[i+1:end]"` would produce a number in MATLAB), may fail to do what you expect in another language. It's the same with *false friends* in human languages.

3.7.3 Hidden Variables

```
1 Avg=0.0
2 x = randn(1,4)
3 for k = 1:length(x)
4     Avg = Avg + x[k]
5 end
6 Avg/k
```

Output

```
UndefVarError: k not defined
```

```
Stacktrace:
```

```
[1] top-level scope
   @ In[4]:6
[2] eval
   @ ./boot.jl:360 [inlined]
[3] include_string(mapexpr::typeof(REPL.softscope), mod::Module,
code::String, filename::String)
   @ Base ./loading.jl:1094
```

The counter remains hidden even if you initialize it before the `for` loop. Here, Julia uses $k = 0$ and not the final value of the counter, which would be `length(x)`.

```
1 Avg=0.0
2 k=0
3 x = randn(1,4)
4 for k = 1:length(x)
5     Avg = Avg + x[k]
6 end
7 Avg/k
```

Output

-Inf

3.8 (Optional Read) While Loops and an Open Problem in Mathematics

With a `for` loop, the maximum number of iterations is upper bounded by $1 + |k_{end} - k_{start}|$. A `while` loop, however, will iterate (loop) until a Boolean (means **T/F**) condition is satisfied. Unlike a `for` loop, a `while` loop can loop forever, so be careful when you use them. We'll show first a really easy example, which you could also solve with a `for` loop and then some less trivial examples.

```
1 # How to compute the factorial of a non-negative integer
2 counter = 5
3 myFactorial = 1
4
5 while (counter > 0)
6     myFactorial = myFactorial * counter;      #Multiply
7     counter = counter - 1;                  #Decrement
8 end
9
10 myFactorial
```

Output

120

Because we have not used Booleans before, let's see what the expression `(counter > 0)` is doing for us.

```
1 # How to compute the factorial of a non-negative integer
2 counter = 5
3 myFactorial = 1
4
5 while (counter > 0)
6     myFactorial = myFactorial * counter;      #Multiply
7     counter = counter - 1;                  #Decrement
8     @show (counter > 0)
9     @show typeof((counter > 0))
10 end
11
12 myFactorial
```

Output

```
counter > 0 = true
typeof(counter > 0) = Bool
counter > 0 = true
typeof(counter > 0) = Bool
counter > 0 = true
typeof(counter > 0) = Bool
counter > 0 = true
typeof(counter > 0) = Bool
counter > 0 = false
typeof(counter > 0) = Bool
```

120

The `while` loop stopped when it encountered a **F** condition: `counter > 0 = false`.

Question: What is the smallest value of n so that the sum $1 + 2 + \dots + n$ is greater than 500?

```

1 sum=0
2 n=0
3 while (sum < 501)
4     n = n + 1
5     sum = sum + n
6 end
7 n

```

Output

32

Here is another way to express the logic, where this time, we want to be greater than 50.

```

1 sum=0
2 n=0
3 #
4 # the exclamation point is logical not
5 # !true = false
6 # !false = true
7 #
8 while !(sum > 50)
9     n = n + 1
10    sum = sum + n
11    @show((sum > 50))
12    @show(!(sum > 50))
13 end
14 println("The while loop terminated when !(sum > 50) = false")
15 n

```

Output

```

sum > 50 = false
!(sum > 50) = true
sum > 50 = false
!(sum > 50) = true
sum > 50 = false
!(sum > 50) = true
sum > 50 = false
!(sum > 50) = true
sum > 50 = false
!(sum > 50) = true
sum > 50 = false
!(sum > 50) = true
sum > 50 = false
!(sum > 50) = true
sum > 50 = false
!(sum > 50) = true
sum > 50 = true
!(sum > 50) = false
The while loop terminated when !(sum > 50) = false

```

10

An Open Math Problem: Multiplicative Persistence

Consider the number 327. It has digits 3, 2, and 7. If we multiply the digits, we obtain 42. The number 42 has two digits, 4 and 2. If we multiply them, we obtain 8, which now has a single digit. The **multiplicative persistence** of a positive integer is the number of times you can go through the cycle of multiplying its digits together, then multiplying the digits of that number, etc., until you reach a single digit number.

It is a **Conjecture in Mathematics** that for any integer $n > 0$, its multiplicative persistence is less than or equal to eleven (yes, the number 11). Can you believe that? A number with a billion trillion digits will always have a multiplicative persistence less than or equal to 11? Say what? No one has found a number with multiplicative persistence greater than 11. And yes, of course, thousands of people, amateurs and professionals alike, have written search routines for a number with multiplicative persistence equal to 12 or higher. A few staunch mathematicians have tried to prove that 11 is in fact the largest it can ever be, and others have tried to show that larger is possible. So far, all attempts have failed! **The genius Srinivasa Ramanujan has worked on this problem** https://it.wikipedia.org/wiki/Srinivasa_Ramanujan. It's fun to play with the code.

The command `digits` takes a positive integer and decomposes it into the digits multiplying the various powers of 10.

```
1 N=277777788888899
2 i=0; Ndigits=digits(N); K=length(Ndigits)
3 while (K>1)
4     i=i+1
5     prodN=Ndigits[1]
6     for k = 2:K
7         prodN=prodN*Ndigits[k]
8     end
9     @show prodN
10    Ndigits=digits(prodN)
11    K=length(Ndigits)
12 end
13 println("The multiplicative persistence of $N is $i")
```

Output

```
prodN = 4996238671872
prodN = 438939648
prodN = 4478976
prodN = 338688
prodN = 27648
prodN = 2688
prodN = 768
prodN = 336
prodN = 54
prodN = 20
prodN = 0
The multiplicative persistence of 277777788888899 is 11
```

3.9 (Optional Read) Double For Loops

You can nest one `for` loop within another. Here is an example that takes a square matrix and zeros all of its entries below the diagonal. In terms of indices, we are going to zero $A[i, j]$ for all $j < i$, where j is the column number and i is the row number. Hence, in row one, there is nothing to zero. In row two, we need to zero $A[2, 1]$. In row three, we need to zero $A[3, 1]$ and $A[3, 2]$, etc.

```
1 using Random
2 Random.seed!(4321)
```

```

3 A=randn(6,6)
4 nRows, nCols =size(A)
5 for i = 1:nRows
6     for j=1:i-1
7         A[i,j] = 0.0
8     end
9 end
10 A

```

Output

```

6×6 Matrix{Float64}:
-0.991273 -0.850184 -1.06297  0.609128  0.498376  0.411742
 0.0      0.592798 -0.509626 -0.547334 -0.670348 -0.826093
 0.0      0.0      0.673963  0.24726  -0.52135  0.680463
 0.0      0.0      0.0      -0.860824 -0.71363  -0.322208
 0.0      0.0      0.0      0.0      0.249046  0.373825
 0.0      0.0      0.0      0.0      0.0      -0.044526

```

With a bit of clever indexing, you can do the same thing with a single for loop.

```

1 using Random
2 Random.seed!(4321)
3 A=randn(6,6)
4 nRows, nCols = size(A)
5 for i = 1:nRows
6     A[i, 1:i-1]=0.0*A[i, 1:i-1]
7 end
8 A

```

Output

```

6×6 Matrix{Float64}:
-0.991273 -0.850184 -1.06297  0.609128  0.498376  0.411742
-0.0      0.592798 -0.509626 -0.547334 -0.670348 -0.826093
 0.0      -0.0      0.673963  0.24726  -0.52135  0.680463
-0.0      -0.0      0.0      -0.860824 -0.71363  -0.322208
-0.0      0.0      -0.0      0.0      0.249046  0.373825
 0.0      -0.0      -0.0      0.0      0.0      -0.044526

```

3.10 (Optional Read) A Less Brief Intro to Matrix-Vector Multiplication and the Unnerving Fact that 1 x 1 Matrices and 1-Element Vectors in Julia are not Scalars

Let $a^{\text{row}} = [a_1 \ a_2 \ \dots \ a_k]$ be a row vector with k elements and let $b^{\text{col}} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_k \end{bmatrix}$ be a column vector with the same number of

elements as a^{row} . The product of a^{row} and b^{col} is defined as

$$a^{\text{row}} \cdot b^{\text{col}} := \sum_{i=1}^k a_i b_i := a_1 b_1 + a_2 b_2 + \dots + a_k b_k. \quad (3.3)$$

```

1 aRow = [1.0 pi 5 sqrt(2)]
2 bCol = [21; 0; 11; sqrt(2)]
3 MVProd1 = 0.0
4 for i = 1:length(aRow)

```

```

5  MVProd1 = MVProd1 + aRow[i]*bCol[i]
6  end
7  @show typeof(MVProd1)
8  MVProd1

```

Output

```

typeof(MVProd1) = Float64
78.0

```

```

1  x=zeros(3,1)
2  x[2]=MVProd1 # Assign second component of x
3  x

```

Output

```

3×1 Matrix{Float64}:
 0.0
 78.0
 0.0

```

Because all of the a_i and b_i are real numbers, the sum of their products is a real number. **When you use the built-in multiplication command in Julia, however, you obtain a 1-element vector.** We illustrate this.

```

1  # Much easier and shorter to type
2  aRow=[1.0 pi 5 sqrt(2)]
3  @show typeof(aRow)
4  bCol=[21; 0; 11; sqrt(2)]
5  @show typeof(bCol)
6  MVProd2 = aRow*bCol

```

Output

```

typeof(aRow) = Matrix{Float64}
typeof(bCol) = Vector{Float64}

1-element Vector{Float64}:
 78.0

```

Does this make a difference? It sure does!

```

1  x=zeros(3,1)
2  x[2]=MVProd2
3  x

```

Output

```

MethodError: Cannot `convert` an object of type Vector{Float64} to an object of
type Float64

```

Closest candidates are:

```

convert(::Type{T}, ::T) where T<:Number at number.jl:6
convert(::Type{T}, ::Number) where T<:Number at number.jl:7
convert(::Type{T}, ::Base.TwicePrecision) where T<:Number at twiceprecision.jl:250
...

```

Stacktrace:

```

[1] setindex!(A::Matrix{Float64}, x::Vector{Float64}, i1::Int64)
@ Base ./array.jl:839

```

```

[2] top-level scope
  @ In[40]:2
[3] eval
  @ ./boot.jl:360 [inlined]
[4] include_string(mapexpr::typeof(REPL.softscope), mod::Module, code::String,
filename::String)
  @ Base ./loading.jl:1094

```

The source of the error is that we need to assign numbers to the entries of vectors and matrices and not other vectors or matrices. When we coded `x[2] = MVProd2` we attempted to insert the 1-element vector into the second entry of the vector x . Note the error message **MethodError: Cannot 'convert' an object of type Vector{Float64} to an object of type {Float64}**. Because x already contains numbers, for a given value of the index k , we need to place a number into $x[k]$; in particular, **we cannot insert a 1-element vector into $x[2]$** . What do we do? We extract the value of the 1-element `Vector{Float64}` and then assign it to $x[k]$.

```

1 x=zeros(3,1)
2 @show typeof(MVProd2)
3 @show typeof(MVProd2[1])
4 x[2]=MVProd2[1] # When we extract the value from a
5                 # Vector{Float64} it becomes a Float64
6                 # and we can assign it as a component of a vector
7 x

```

Output

```

typeof(MVProd2) = Vector{Float64}
typeof(MVProd2[1]) = Float64
3×1 Matrix{Float64}:
 0.0
 78.0
 0.0

```

Here is second way to solve the “problem”.

```

1 # Second Solution
2 aCol=[1.0; pi; 5; sqrt(2)]
3 @show typeof(aCol)
4 @show typeof(aCol') # Note the apostrophe
5 bCol=[21; 0; 11; sqrt(2)]
6 @show typeof(bCol)
7 MVProd3 = (aCol')*bCol # parentheses are not necessary,
8                 # but note the apostrophe

```

Output

```

typeof(aCol) = Vector{Float64}
typeof(aCol') = LinearAlgebra.Adjoint{Float64, Vector{Float64}}
typeof(bCol) = Vector{Float64}

78.0

```

```

1 x=zeros(3,1)
2 @show typeof(MVProd3)
3 x[2]=MVProd3
4 x

```

Output

```

typeof(MVProd3) = Float64

```

```
3×1 Matrix{Float64}:  
 0.0  
 78.0  
 0.0
```

`LinearAlgebra.AdjointFloat64`, `VectorFloat64` is another TYPE in Julia. An adjoint is a kind of vector that you will only see in **advanced linear algebra courses**. An adjoint can be thought of as a special kind of row vector. In advanced math, an adjoint vector times an ordinary (column) vector is a real number. The fact that Julia uses this TYPE is maddening for me, because it makes the language confusing for beginning college students. In fact, most engineering students will never see “adjoints” even when working on a PhD. Now, your author took most of the PhD qualifying courses in math, so for me, it’s OK. But for you, it’s just an unneeded distraction. Enough said!

Chapter 4

Julia Lab 4: If Statements, Function Creation, and “Peeling the Onion”

Learning Objectives

- What is an `if` statement and how to use it
- A better way to build your own functions
- A key step in Lower-Upper (LU) Matrix Factorization
- Debugging, or finding and fixing errors in your code

Outcomes

- Understand how to use Booleans
- Ability to control program flow via conditional statements
- Keywords `if`, `else`, `elseif`, and `end`
- Keywords `function`, `return`, and `end`
- How to peel off the top row and left-most column of a matrix.
- (Optional Read) How to compute more easily the area of a general triangle

Either download Lab4 from our Canvas site or open up a Jupyter notebook so that you can enter code as we go. It is suggested that you have line numbering toggled on.

4.1 If Statements aka Conditional Statements

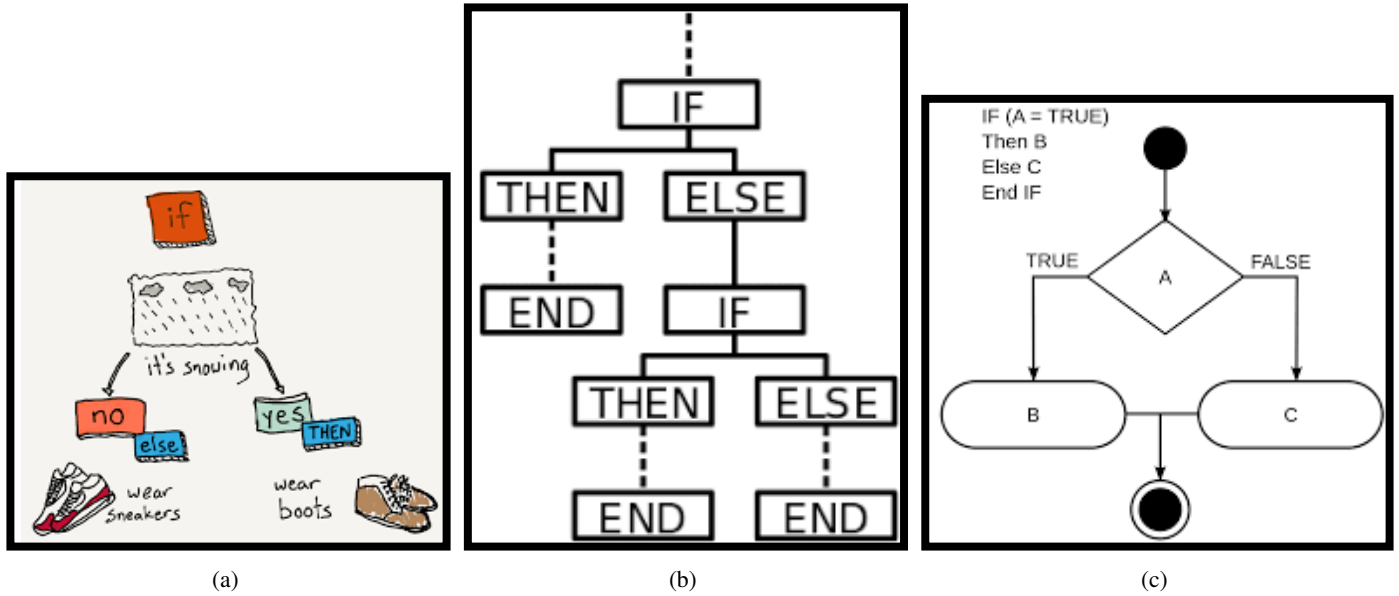


Figure 4.1: If statements are also called conditional statements. They allow you to test for a condition before executing a section of code, such as making a recommendation of apparel as a function of the weather. The conditions are always **T/F** tests. (a) is from <https://makecode.microbit.org/courses/csintro/conditionals/overview>, while (b) and (c) are from [https://en.wikipedia.org/wiki/Conditional_\(computer_programming\)](https://en.wikipedia.org/wiki/Conditional_(computer_programming))

“In computer science, conditionals (that is, conditional statements, conditional expressions and conditional constructs,) are programming language commands for handling decisions;” see [https://en.wikipedia.org/wiki/Conditional_\(computer_programming\)](https://en.wikipedia.org/wiki/Conditional_(computer_programming)). A simple example would be to check for a zero in a denominator before doing a division operation.

Julia provides a variety of control flow constructs. Control flow regulates the order of statements to be executed. ‘if-else’ is a common control flow in Julia and other programming languages. The syntax is as follows:

```

if Boolean is T
    **Execute statement block1**
    **Execute statement block1**
else
    **Execute statement block2**
    **Execute statement block2**
    **Execute statement block2**
end

```

Remark 4.1 Note that in Julia, as in MATLAB, **then** never appears as a key word! It is implied. The first true “if” or “elseif” condition is executed, and then the entire block of if statements is exited. Nothing further is evaluated. If you do not include an else before the end statement, then, if all of the if conditions fail, nothing will be done. This is often very useful, for example, when checking for potential errors in the incoming data to a function.

```

1 if (Boolean 1)
2     **statement block 1**
3 elseif (Boolean 2)
4     **statement block 2**
5 elseif (Boolean 3)
6     **statement block 3**
7 ...

```

```

8 elseif (Boolean N-1)
9     **statement block N-1**
10 else # nothing goes here
11     **statement block N**
12 end

```

```

1 # Our first if then else statement
2 #
3 # If the condition is evaluated to be true, execute statement1,
4 # otherwise, execute statement2
5 x = sqrt(2)
6 y = pi/2
7 if x < y # testing the Boolean (x < y)
8     println("x is less than y")
9 else
10    println("x is not less than y")
11 end

```

Output

```
x is less than y
```

You can string together multiple Boolean conditions, like so.

```

1 # More conditions can be evaluated using elseif
2 x = 1
3 y = 1
4 if x < y
5     println("x is less than y")
6 elseif x > y
7     println("x is greater than y")
8 else
9     println("x is equal to y")
10 end

```

Output

```
x is equal to y
```

And it is OK to skip the else statement.

```

1 # It is optional to include the else
2 #
3 x = 1.4142-sqrt(2)
4 y = NaN
5 if x < 0
6     y = -x
7 end
8 [x y]

```

Output

```

1×2 Matrix{Float64}:
-1.35624e-5  1.35624e-5

```

Example 4.2 Write an “if statement” that tests if a variable x is a 1-element vector or not, and when it is a 1-element vector, redefines it as a scalar.

Solution We first test the `length` of `x`. When the length is greater than one, we do nothing. If the length equals one, we then test if `x` has `Type Vector`. If it does, we extract its first element to create a scalar.

```
1 x=[pi]
2 # x = pi # uncomment and run, comparing to x = [pi]
3 @show string(typeof(x))
4 @show string(typeof(x))[1:6]
5 if length(x) == 1
6     if string(typeof(x))[1:6] == "Vector" # Double equals means EQUIVALENT TO
7         @show x=x[1]
8         @show typeof(x)
9     else
10        @show x
11    end
12 end
13 X
```

Output

```
string(typeof(x)) = "Vector\{Irrational\{:pi\}\}"
(string(typeof(x))[1:6] = "Vector"
x = x[1] = pi
typeof(x) = Irrational\{:pi\}

pi = 3.1415926535897...
```

4.2 Writing Better Functions

“A function is a block of organized code that is used to perform a single task. [Functions] provide better modularity for your application and reuse-ability. Depending on the programming language, a function may be called a subroutine, a procedure, a routine, a method, or a subprogram. The generic term, callable unit, is sometimes used. Using functions can allow you to be able to keep your code clean and organized, making it easy to read, and allows the debugging process to be easier;” from https://en.wikiversity.org/wiki/Programming_Fundamentals/Functions.

In Julia, the keywords are `function`, `return`, and `end`. Here we show a very simple function that takes in a variable x and returns $y = x + \pi x^2$.

```
1 # A simple function using the keywords FUNCTION, END, and RETURN
2 function f(x)
3     y = x+pi*x^2
4     return y
5 end
6 # Call the function
7 f(2)
```

Output

```
14.566370614359172
```

You can also return more than one value, as in the following example, which also illustrates that any commands that follow the first `return` statement are ignored.

```
1 #
2 function f(x)
3     y = x+pi*x^2
4     z = sin(x) + 27*x^3
```

```

5     return y, z
6     w=x+2 # a superfluous line that will be completely ignored
7 end
8 # All operations after the return keyword will be ignored
9 #
10 # Call the function
11 f(3)

```

Output

```
(31.274333882308138, 729.1411200080598)
```

Here is a function that determines the absolute value of a scalar.

```

1 # Build your own absolute value function
2 function myAbs(a)
3     if a >= 0
4         return a
5     else
6         return -a
7     end
8 end
9 # Call the function
10 @show myAbs(-2)
11 @show myAbs(-2.0)

```

Output

```

myAbs(-2) = 2
myAbs(-2.0) = 2.0

2.0

```

Julia has its own built-in `maximum` function. We'll build our own function just to show how it can be done.

```

1 # Define a more useful function with multiple returns
2 function myMax(input_vector)
3     # initialize the max to the first element
4     # then update when we find a bigger element
5     maximum_value = input_vector[1]
6     maximum_index = 1
7     for i = 1:length(input_vector)
8         if input_vector[i] > maximum_value
9             maximum_value = input_vector[i]
10            maximum_index = i
11        end
12    end
13    return maximum_value, maximum_index
14 end

```

Output

```
myMax (generic function with 1 method)
```

When we test it, we obtain the maximum value as well as the index of the vector where the maximum value is stored.

```

1 # Test the function
2 a = [0, 1, 5, 3.0]
3 maximum_value, maximum_index = myMax(a)

```

Output

```
(5.0, 3)
```

What if we use a single variable name for the output of the function?

```
1 # Test the function
2 a = [0, 1, 5, 3.0]
3 Answer = myMax(a)
```

Output

```
(5.0, 3)
```

Then `Answer` is a `Tuple`. Note that `Answer[1]` holds the maximum value and `Answer[2]` holds the index.

```
1 @show Answer[1]
2 Answer[2]
```

Output

```
Answer[1] = 5.0
```

```
3
```

4.3 A Function for Multiplying Matrices

Julia has a built-in matrix multiplication function, $A * B$. Just for practice, we'll build a function that uses the method of Chapter 4 in our textbook, where we sum over the product of the columns of A times the rows of B . We will then verify that our function gives the same answer as the usual method of doing matrix multiplication.

```
1 function myMatrixMultiply(A, B)
2     nRowsA, nColsA = size(A)
3     nRowsB, nColsB = size(B)
4     if nColsA != nRowsB
5         println("Error: Matrix sizes are not compatible for multiplication")
6         C=NaN
7         return C
8     end
9     C=zeros(nRowsA, nColsB)
10    for i = 1:nColsA
11        C = C + A[:,i]*B[i:i,:]
12    end
13    return C
14 end
```

Output

```
myMatrixMultiply (generic function with 1 method)
```

```
1 # test of myMatrixMultiply(A,B)
2 using Random
3 Random.seed!(4321)
4 A=randn(3,4)
5 B=randn(4,5)
6
7 C=myMatrixMultiply(A,B)
8 # Matrix of zeros means we got it right!
9 C-A*B
```

Output

```
3×5 Matrix{Float64}:
 0.0          2.22045e-16  0.0  0.0          0.0
-1.11022e-16  0.0          0.0  0.0          1.11022e-16
 0.0          0.0          0.0 -2.22045e-16 -5.55112e-17
```

The next function implements matrix multiplication the way it is done in Julia.

```
1 function myMatrixMultiplyOrdinary (A, B)
2     nRowsA, nColsA = size(A)
3     nRowsB, nColsB = size(B)
4     if nColsA != nRowsB
5         println("Error: Matrix sizes are not compatible for multiplication")
6         C=NaN
7         return C
8     end
9     C=zeros(nRowsA, nColsB)
10    for i = 1:nRowsA
11        for j = 1:nColsB
12            C[i,j] = (A[i:i, :] * B[:, j]) [1]
13        end
14    end
15    return C
16 end
```

Output

```
myMatrixMultiplyOrdinary (generic function with 1 method)
```

```
1 # test of myMatrixMultiplyOrdinary(A, B)
2 using Random
3 Random.seed!(4321)
4 A=randn(3, 4)
5 B=randn(4, 5)
6
7 C=myMatrixMultiplyOrdinary(A, B)
8 # Matrix of zeros means we got it right!
9 C-A*B
```

Output

```
3×5 Matrix{Float64}:
 0.0          0.0  0.0  0.0  0.0
-1.11022e-16  0.0  0.0  0.0  0.0
 0.0          0.0  0.0  0.0  0.0
```

4.4 Forward Substitution

This is how to solve $Ax = b$ when A is a square lower triangular matrix, with no zeros on the diagonal. See Chapter 3 of the ROB 101 Textbook.

```
1 using LinearAlgebra
2 function forwardsub(L, b)
3     # Assert no entries on the diagonal of U
4     # are 0 (or very close to 0)
5     @assert minimum(abs.(diag(L))) > 1e-4
```

```

6 # START of our computations
7 n = length(b)
8 x = Vector{Float64}(undef, n); #initialize vector x to the correct size
9 x[1] = b[1]/L[1,1] #find the first entry of x
10 for i = 2:n #find every entry from the 2nd to the end
11     x[i]=( b[i] - (L[i,1:i-1])' *x[1:i-1] )/L[i,i]
12     #notice that we used a transpose operator to get the row of L
13 end
14 # END of our computations.
15 return x
16 end

```

Output

forwardsub (generic function with 1 method)

Here we work an example with 300 variables.

```

1 using Random
2 Random.seed!(4321)
3 N=300
4 L=rand(N,N)
5 for i = 1:N
6     L[i, (i+1):N] = 0.0 * L[i, (i+1):N]
7     L[i,i]=1.0
8 end
9 b=randn(N,1)
10 x=forwardsub(L, b)
11 #x = inv(L)*b
12 norm(L*x-b)

```

Output

1.303218919812078e-9

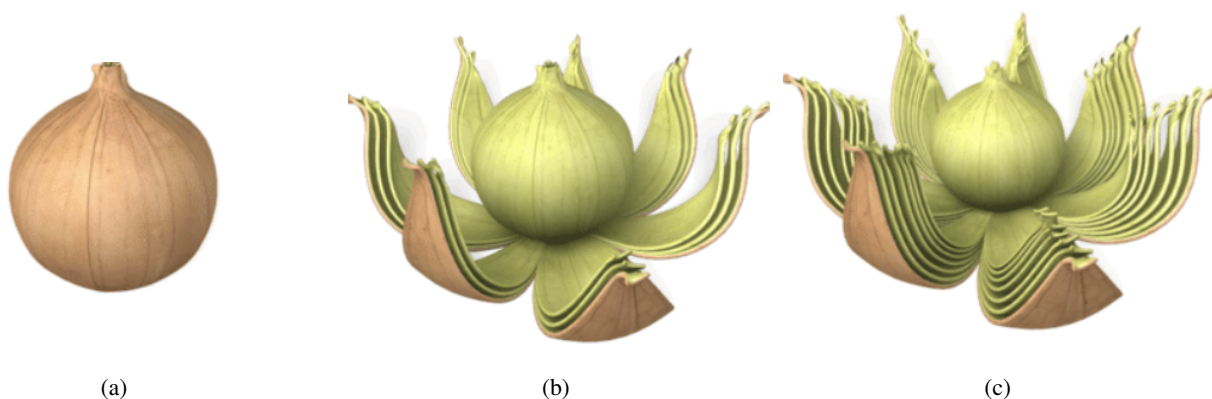


Figure 4.2: We will take a matrix apart, row by row, column by column, until there is nothing left but zeros. Images are snapshots from https://content.presentermedia.com/content/animsp/00021000/21530/onion_layers_300_wht.gif.

4.5 Peeling the Onion: the Base Step for LU Factorization

This material is tied to Chapter 5 of the ROB 101 Textbook. “Peeling the Onion” is an elementary step in factoring a matrix into the product of a lower triangular matrix L and an upper triangular matrix U . We will build a function that peels one layer of a matrix at

a time. It will work for the special case that the “pivot” element is non-zero. Chapter 5 explains how to modify the process when the “pivot” is zero, in case you are curious!

Consider the square matrix

$$M_0 = \begin{bmatrix} 1 & 4 & 5 \\ 2 & 9 & 17 \\ 3 & 18 & 58 \end{bmatrix}.$$

Our goal is to find a column vector C_1 and a row vector R_1 such that

$$M - C_1 \cdot R_1 = \begin{bmatrix} 0 & 0 & 0 \\ 0 & * & * \\ 0 & * & * \end{bmatrix},$$

where * denotes “don’t care” in the sense that we do not care about their particular values. Another way to say this is, we want to zero out the first column and the first row of M . That means, C_1 and R_1 are chosen so that the first column and first row of their matrix product $C_1 \cdot R_1$ match the first column and first row of M .

To see that this is possible, observe that

$$\begin{aligned} M_0 &= \begin{bmatrix} 1 & 4 & 5 \\ 2 & 9 & 17 \\ 3 & 18 & 58 \end{bmatrix} = \underbrace{\begin{bmatrix} 0.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 7.0 \\ 0.0 & 6.0 & 43.0 \end{bmatrix}}_{M_1} + \underbrace{\begin{bmatrix} 1.0 \\ 2.0 \\ 3.0 \end{bmatrix}}_{C_1} \cdot \underbrace{\begin{bmatrix} 1.0 & 4.0 & 5.0 \end{bmatrix}}_{R_1} \\ &= \underbrace{\begin{bmatrix} 0.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 7.0 \\ 0.0 & 6.0 & 43.0 \end{bmatrix}}_{M_1} + \underbrace{\begin{bmatrix} 1.0 & 4.0 & 5.0 \\ 2.0 & 8.0 & 10.0 \\ 3.0 & 12.0 & 15.0 \end{bmatrix}}_{C_1 \cdot R_1} \end{aligned}$$

where we arrive at M_1 after **“Peeling off the First Row and First Column”** of M_0 .

Furthermore, we can write

$$\begin{aligned} M_1 &= \begin{bmatrix} 0.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 7.0 \\ 0.0 & 6.0 & 43.0 \end{bmatrix} = \underbrace{\begin{bmatrix} 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 1.0 \end{bmatrix}}_{M_2} + \underbrace{\begin{bmatrix} 0.0 \\ 1.0 \\ 6.0 \end{bmatrix}}_{C_2} \cdot \underbrace{\begin{bmatrix} 0.0 & 1.0 & 7.0 \end{bmatrix}}_{R_2} \\ &= \underbrace{\begin{bmatrix} 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 1.0 \end{bmatrix}}_{M_2} + \underbrace{\begin{bmatrix} 0.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 7.0 \\ 0.0 & 6.0 & 42.0 \end{bmatrix}}_{C_2 \cdot R_2} \end{aligned}$$

where we arrive at M_2 after **“Peeling off the Second Row and Second Column”** of M_1 .

Finally, we can write

$$\begin{aligned} M_2 &= \begin{bmatrix} 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 1.0 \end{bmatrix} = \underbrace{\begin{bmatrix} 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \end{bmatrix}}_{M_3} + \underbrace{\begin{bmatrix} 0.0 \\ 0.0 \\ 1.0 \end{bmatrix}}_{C_3} \cdot \underbrace{\begin{bmatrix} 0.0 & 0.0 & 1.0 \end{bmatrix}}_{R_3} \\ &= \underbrace{\begin{bmatrix} 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \end{bmatrix}}_{M_3} + \underbrace{\begin{bmatrix} 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 1.0 \end{bmatrix}}_{C_3 \cdot R_3} \end{aligned}$$

where we arrive at M_3 after **“Peeling off the Third Row and Third Column”** of M_2 . Because M_3 is the zero matrix, the process stops.

We have arrived at

$$\begin{aligned}
 M_0 = \begin{bmatrix} 1 & 4 & 5 \\ 2 & 9 & 17 \\ 3 & 18 & 58 \end{bmatrix} &= \underbrace{\begin{bmatrix} 1.0 \\ 2.0 \\ 3.0 \end{bmatrix}}_{C_1} \cdot \underbrace{\begin{bmatrix} 1.0 & 4.0 & 5.0 \end{bmatrix}}_{R_1} + \underbrace{\begin{bmatrix} 0.0 \\ 1.0 \\ 6.0 \end{bmatrix}}_{C_2} \cdot \underbrace{\begin{bmatrix} 0.0 & 1.0 & 7.0 \end{bmatrix}}_{R_2} + \underbrace{\begin{bmatrix} 0.0 \\ 0.0 \\ 1.0 \end{bmatrix}}_{C_3} \cdot \underbrace{\begin{bmatrix} 0.0 & 0.0 & 1.0 \end{bmatrix}}_{R_3} \\
 &= \underbrace{\begin{bmatrix} 1.0 & 0.0 & 0.0 \\ 2.0 & 1.0 & 0.0 \\ 3.0 & 6.0 & 1.0 \end{bmatrix}}_{\substack{C_1 & C_2 & C_3}} \cdot \underbrace{\begin{bmatrix} 1.0 & 4.0 & 5.0 \\ 0.0 & 1.0 & 7.0 \\ 0.0 & 0.0 & 1.0 \end{bmatrix}}_{\substack{R_1 \\ R_2 \\ R_3}} \\
 &= L \cdot U
 \end{aligned}$$

How do we find the column vectors C_i and the row vectors R_i , for $1 \leq i \leq 3$? We have an algorithm for that.

```

1 function peel_one_layer (Temp, k)
2     # k is the layer we are peeling (row and column)
3     # Temp is the matrix from which we are
4     # peeling the k-th row and k-th column
5     # We assume layers 1:k-1 are already zeroed
6     pivot = Temp[k, k] # We assume pivot != 0
7     C = Temp[:, k] / pivot # Normalized k-th column
8     R = Temp[k:k, :] # k-th row, no normalization
9     Temp = Temp - C*R
10    return C, R, Temp
11 end

```

Output

peel_one_layer (generic function with 1 method)

Now, let's apply our function.

```

1 M0=[1.0 4 5; 2 9 17; 3 18 58]
2 C1, R1, M1=peel_one_layer (M0, 1) # layer 1
3 @show C1
4 @show R1
5 M1

```

Output

```

C1 = [1.0, 2.0, 3.0]
R1 = [1.0 4.0 5.0]

3×3 Matrix{Float64}:
 0.0  0.0  0.0
 0.0  1.0  7.0
 0.0  6.0  43.0

```

Note that we “peeled off” the first column and row.

```

1 C2, R2, M2=peel_one_layer (M1, 2) # layer 2
2 @show C2
3 @show R2
4 M2

```

Output

```
C2 = [0.0, 1.0, 6.0]
R2 = [0.0 1.0 7.0]
```

```
3x3 Matrix{Float64}:
 0.0  0.0  0.0
 0.0  0.0  0.0
 0.0  0.0  1.0
```

Note that we “peeled off” the second column and row.

```
1 C3, R3, M3=peel_one_layer(M2, 3) # layer 3
2 @show C3
3 @show R3
4 M3
```

Output

```
C3 = [0.0, 0.0, 1.0]
R3 = [0.0 0.0 1.0]
```

```
3x3 Matrix{Float64}:
 0.0  0.0  0.0
 0.0  0.0  0.0
 0.0  0.0  0.0
```

We arrived at a matrix of zeros. Next, we form a lower triangular matrix, L

```
1 L=[C1 C2 C3]
```

Output

```
3x3 Matrix{Float64}:
 1.0  0.0  0.0
 2.0  1.0  0.0
 3.0  6.0  1.0
```

and an upper triangular matrix, U

```
1 U = [R1; R2; R3]
```

Output

```
3x3 Matrix{Float64}:
 1.0  4.0  5.0
 0.0  1.0  7.0
 0.0  0.0  1.0
```

We now show that $M_0 = L \cdot U$,

```
1 M0=L*U
```

Output

```
3x3 Matrix{Float64}:
 0.0  0.0  0.0
 0.0  0.0  0.0
 0.0  0.0  0.0
```

We can place our function `peel_one_layer (Temp, k)` in a for loop to compute an LU Factorization when row permutation is not required.

```
1 function myLU(A)
2     Temp = copy(A) # Initialize Temp matrix by
3                 # copying the original matrix A
4     nRows, nCols = size(Temp) # Get the size of the input matrix
5     K = minimum([nRows, nCols])
6     # Initialize the lower and upper triangular matrix
7     L = zeros(Float64, (nRows, K)) # Using zeros function by
8                                     # specifying both type and size
9
10    U = zeros(Float64, (K, nCols))
11    # Here we do the actual factorization
12    for k = 1:K
13        C, R, Temp = peel_one_layer(Temp, k)
14        L[:, k] = C
15        U[k:k, :] = R
16    end
17    return L, U
end
```

Output

myLU (generic function with 1 method)

```
1 using Random
2 using LinearAlgebra
3 Random.seed!(09182022)
4 A = randn(5, 7)
5 #
6 L, U = myLU(A)
```

Output

```
([1.0 -0.0 ... 0.0 -0.0; 1.3841751352784288 1.0 ... 0.0 -0.0; ... ;
1.8240441684383097 1.7669720431792901 ... 1.0 -0.0;
1.1660675070497575 0.4629237529415664 ... -0.8475551711339656 1.0],
[-0.3236905701556438 0.9372755784538437 ... -0.009357226530146447 -0.0076384514441543774;
0.0 -2.1005108102249452 ... 1.4460922800584077 0.41893233388727713;
... ; 0.0 0.0 ... 0.19792290864355966 -0.6285612979102336;
-5.551115123125783e-17 0.0 ... -1.3304631421106152 1.0052145594104471])
```

The above output is hard to read. It is better if you look at L and U in separate cells.

```
1 L
```

Output

```
5×5 Matrix{Float64}:
 1.0      -0.0      0.0      0.0      -0.0
 1.38418  1.0      0.0      0.0      -0.0
 8.38948  4.18061  1.0      0.0      -0.0
 1.82404  1.76697  0.271808 1.0      -0.0
 1.16607  0.462924 -0.135905 -0.847555 1.0
```

```
1 U
```

Output

```
5×7 Matrix{Float64}:
-0.323691    0.937276    0.578516    ... 0.487937   -0.00935723  -0.00763845
 0.0         -2.10051   -1.18148    -0.516585    1.44609      0.418932
 0.0         0.0        2.42504     -1.45327     -6.78688     -1.58847
 0.0         0.0        0.0         0.717684     0.197923    -0.628561
-5.55112e-17 0.0        0.0         -2.41577     -1.33046     1.00521
```

```
1 A-L*U
```

Output

```
5×7 Matrix{Float64}:
 0.0  0.0    0.0    0.0    ... 0.0    0.0
 0.0  0.0    0.0    6.93889e-17    0.0    0.0
 0.0 -4.44089e-16 -4.44089e-16 6.66134e-16    -4.44089e-16 -5.55112e-17
 0.0 -4.44089e-16 -1.11022e-16 5.55112e-17    3.33067e-16 0.0
 0.0 1.38778e-17 8.32667e-17 0.0    1.38778e-16 0.0
```

4.6 Debugging

We'll introduce errors into a few of the functions and if-then-else statements we have used in the Chapter and see how the “bugs” manifest themselves.

4.6.1 Misusing operations designed for scalar variables

Here is a function designed for x a real number.

```
1 function f(x)
2     y = x+pi*x^2
3     return y
4 end
```

Output

```
f (generic function with 1 method)
```

We forget that we designed it for a real number and try to apply it to a vector.

```
1 f([1;2])
```

Output

```
MethodError: no method matching ^(::Vector{Int64}, ::Int64)
Closest candidates are:
 ^(::Union{AbstractChar, AbstractString}, ::Integer) at strings/basic.jl:718
 ^(::Complex{var"#s79"} where var"#s79"<:AbstractFloat, ::Integer) at complex.jl:818
 ^(::Complex{var"#s79"} where var"#s79"<:Integer, ::Integer) at complex.jl:820
 ...
```

Stacktrace:

```
[1] macro expansion
  @ ./none:0 [inlined]
[2] literal_pow
  @ ./none:0 [inlined]
[3] f(x::Vector{Int64})
  @ Main ./In[1]:2
[4] top-level scope
```

```

@ In[2]:1
[5] eval
@ ./boot.jl:360 [inlined]
[6] include_string(mapexpr::typeof(REPL.softscope), mod::Module,
code::String, filename::String)
@ Base ./loading.jl:1094

```

In the above error message, top-level scope, @ In[1]:2 is referring to the Cell where we defined the function. The message “**MethodError: no method matching** $\wedge(::\text{VectorInt64}, ::\text{Int64})$ ” is telling us that Julia does not know how to compute the square of a vector. There are two solutions to this.

```

1 f.([1;2]) # Solution 1 is to apply broadcasting

```

Output

```

2-element Vector{Float64}:
 4.141592653589793
14.566370614359172

```

```

1 function f(x)
2     y = x .+ pi * x.^2 # Solution 2 is make the function work for vectors or scalars
3     return y
4 end
5
6 f([1;2])

```

Output

```

2-element Vector{Float64}:
 4.141592653589793
14.566370614359172

```

4.6.2 Bounds Error

We recall our function for multiplying matrices.

```

1 function myMatrixMultiply(A, B)
2     nRowsA, nColsA = size(A)
3     nRowsB, nColsB = size(B)
4     if nColsA != nRowsB
5         println("Error: Matrix sizes are not compatible for multiplication")
6         C=NaN
7         return C
8     end
9     C=zeros(nRowsA, nColsB)
10    for i = 1:nColsA
11        C = C + A[:,i]*B[i:i, :]
12    end
13    return C
14 end

```

Output

```

myMatrixMultiply (generic function with 1 method)

```

We give the function what looks like perfectly good data.

```

1 A=[1 2 3 4]
2 B=[2;3;4;5.0]
3 myMatrixMultiply(A,B)

```

Output

BoundsError: attempt to access Tuple{Int64} at index [2]

Stacktrace:

```

[1] indexed_iterate
   @ ./tuple.jl:86 [inlined]
[2] myMatrixMultiply(A::Matrix{Int64}, B::Vector{Float64})
   @ Main ./In[6]:3
[3] top-level scope
   @ In[7]:3
[4] eval
   @ ./boot.jl:360 [inlined]
[5] include_string(mapexpr::typeof(REPL.softscope), mod::Module,
code::String, filename::String)
   @ Base ./loading.jl:1094

```

Cell [6] is where we defined the function `myMatrixMultiply` and line 3 is the command `nRowsB, nColsB = size(B)`. The problem is that `B` is a vector and the `size` command does not return a value for `nColsB`, even though we “know” it should be one.

In the following, we show two ways to fix this issue. In the first `if-then-else` statement, we make use of Types in Julia. In the second `if-then-else` statement, we exploit the fact that if `B` is a matrix, then `size(B)` has two components, and hence its length is two, and when `B` is a vector or scalar, `size(B)` will have length one. This is certainly the simpler solution of the two. **An even simpler solution is only to call the function with matrix arguments!**

```

1 function myMatrixMultiply(A,B)
2     if (typeof(A) == Matrix{Int64}) || (typeof(A) == Matrix{Float64})
3         nRowsA, nColsA = size(A)
4     else
5         nRowsA = length(A)
6         nColsA = 1
7     end
8
9     if length(size(B)) == 2
10        nRowsB, nColsB = size(B)
11    else
12        nRowsB = length(B)
13        nColsB = 1
14    end
15    if nColsA != nRowsB
16        @show nColsA
17        @show nRowsB
18        println("Error: Matrix sizes are not compatible for multiplication")
19        C=NaN
20        return C
21    end
22    C=zeros(nRowsA, nColsB)
23    for i = 1:nColsA
24        C = C + A[:,i]*B[i:i,:]
25    end
26    return C
27 end

```

Output

myMatrixMultiply (generic function with 1 method)

```
1 myMatrixMultiply(A, B)
```

Output

1×1 Matrix{Float64}:

40.0

4.6.3 Using the display() command to find an error buried in a function

We insert an error in a function that worked just fine for us in Chapter 4.5. Can you spot the error before we run the function?

```
1 function peel_one_layer(Temp, k)
2 # k is the layer we are peeling (row and column)
3 # Temp is the matrix from which we are
4 # peeling the k-th row and k-th column
5     pivot = Temp[k, k]
6     C = Temp[:, k] # kth column
7     # next we normalize
8     C = C / pivot # We assume pivot != 0
9     R = Temp[k, :] # kth row
10    Temp = Temp - C*R
11    return C, R, Temp
12 end
```

Output

peel_one_layer (generic function with 1 method)

```
1 Temp=[1.0 2; 3 4]
2 display(Temp)
3 C, R, Temp = peel_one_layer(Temp, 1)
```

Output

2×2 Matrix{Float64}:

1.0 2.0

3.0 4.0

MethodError: no method matching * (::Vector{Float64}, ::Vector{Float64})

Closest candidates are:

* (::Any, ::Any, ::Any, ::Any...) at operators.jl:560

* (::StridedMatrix{T}, ::StridedVector{S}) where

{T<:Union{Float32, Float64, ComplexF32, ComplexF64}, S<:Real} at

/buildworker/worker/package_linux64/build/usr/share/julia/stdlib/v1.6/

LinearAlgebra/src/matmul.jl:44

* (::StridedVecOrMat{T} where T, ::LinearAlgebra.Adjoint{var"#s832", var"#s831"})

where {var"#s832", var"#s831"<:LinearAlgebra.LQPackedQ}) at

/buildworker/worker/package_linux64/build/usr/share/julia/stdlib

/v1.6/LinearAlgebra/src/lq.jl:254

...

Stacktrace:

[1] peel_one_layer(Temp::Matrix{Float64}, k::Int64)

@ Main ./In[49]:10


```

[2] top-level scope
    @ In[52]:3
[3] eval
    @ ./boot.jl:360 [inlined]
[4] include_string(mapexpr::typeof(REPL.softscope), mod::Module, code::String,
filename::String)
    @ Base ./loading.jl:1094

```

The message no method matching `*` (`::VectorFloat64`, `::VectorFloat64`) is telling us that Julia does not know how to multiply two vectors. We are surprised by this message because we think, in line 10 of our function (from the message `In[49]:10`) that we are multiplying a column vector C times a row vector R , which is a legitimate operation. **To find the error, we use the `display` command** (you could also have used the `@show` command).

```

1 function peel_one_layer(Temp, k)
2 # k is the layer we are peeling (row and column)
3 # Temp is the matrix from which we are
4 # peeling the k-th row and k-th column
5     pivot = Temp[k, k]
6     C = Temp[:, k] # kth column
7     # next we normalize
8     C = C / pivot # We assume pivot != 0
9     R = Temp[k, :] # kth row
10    display(C)
11    display(R)
12    Temp = Temp - C*R
13    return C, R, Temp
14 end
15
16 C, R, Temp = peel_one_layer(Temp, 1)

```

Output

```

2-element Vector{Float64}:
 1.0
 3.0
2-element Vector{Float64}:
 1.0
 2.0
MethodError: no method matching * (::Vector{Float64}, ::Vector{Float64})

```

Sure enough, R is a column vector. And we note that we should have used `R = Temp[k:k, :]` to extract a row and keep it a row.

If we use the `@show` command, we get the same information but it's less direct in that we have to recognize that `R = [1.0, 2.0]` is a column vector due to the comma separating the two entries.

```

1 function peel_one_layer(Temp, k)
2 # k is the layer we are peeling (row and column)
3 # Temp is the matrix from which we are
4 # peeling the k-th row and k-th column
5     pivot = Temp[k, k]
6     C = Temp[:, k] # kth column
7     # next we normalize
8     C = C / pivot # We assume pivot != 0
9     R = Temp[k, :] # kth row
10    @show C
11    @show R
12    Temp = Temp - C*R
13    return C, R, Temp
14 end

```

```
15
16 C, R, Temp = peel_one_layer (Temp, 1)
```

Output

```
C = [1.0, 3.0]
R = [1.0, 2.0]
```

```
MethodError: no method matching * (::Vector{Float64}, ::Vector{Float64})
```

4.7 (Optional Read) “Hero 60 AD” or Heron’s Formula for the Area of a Triangle

While everyone knows the formula $\text{Area} = \frac{1}{2} \text{base} \times \text{height}$ for the area of a triangle, very few of us were taught Heron’s formula, which avoids the use of trigonometry or geometry to compute the height of the triangle (of course, when it is not a right triangle). We create a function using this formula and then illustrate the programming adage, “garbage in, garbage out”.

A derivation of Heron’s formula can be found here on YouTube <https://www.youtube.com/watch?v=9qbpmYqr4so> or in Wikipedia https://en.wikipedia.org/wiki/Heron%27s_formula, where it is pointed out that “The formula is credited to Heron (or Hero) of Alexandria, and a proof can be found in his book *Metrica*, written around AD 60. It has been suggested that Archimedes knew the formula over two centuries earlier, and since *Metrica* is a collection of the mathematical knowledge available in the ancient world, it is possible that the formula predates the reference given in that work.” Moreover, the Chinese had discovered a similar formula, documented in 1247 https://en.wikipedia.org/wiki/Mathematical_Treatise_in_Nine_Sections, though it may have been known much earlier.

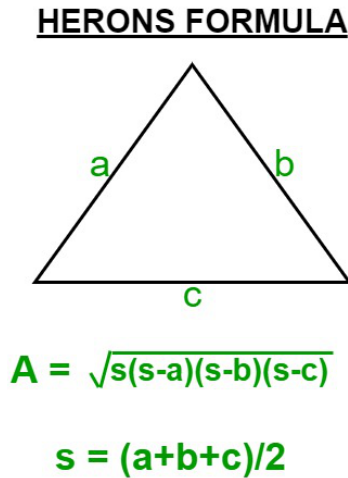


Figure 4.3: Heron’s formula for the area of a triangle. Source <https://www.geeksforgeeks.org/herons-formula/>.

```
1 function heron (a, b, c)
2     s = (a + b + c) / 2
3     Area = sqrt ( s * (s-a) * (s-b) * (s-c) )
4     return Area
5 end
```

Output

```
heron (generic function with 1 method)
```

We now insert the canonical right triangle, (3, 4, 5), where we can take the base as 3 and the height as 2 or the base as 4 and the height as 1.5. In either case, the area is 6.0, and voilà, Heron agrees with us!

```
1 area = heron (3, 4, 5)
```

Output

6.0

What about a triangle with side lengths (3, 4, 4.5)?

```
1 area = heron (3, 4, 4.5)
```

Output

5.881313097429858

Can you correctly compute the area using trigonometry? Go for it!

Garbage in, garbage out. What if we insert data into our function where the lengths of the sides are not compatible with forming a triangle, such as (3, 4, 8)?

```
1 area = heron (3, 4, 8)
```

Output

```
DomainError with -59.0625:  
sqrt will only return a complex result if called with a complex argument.  
Try sqrt(Complex(x)).
```

Stacktrace:

```
[1] throw_complex_domainerror(f::Symbol, x::Float64)  
  @ Base.Math ./math.jl:33  
[2] sqrt  
  @ ./math.jl:582 [inlined]  
[3] heron(a::Int64, b::Int64, c::Int64)  
  @ Main ./In[50]:3  
[4] top-level scope  
  @ In[51]:1  
[5] eval  
  @ ./boot.jl:360 [inlined]  
[6] include_string(mapexpr::typeof(REPL.softscope), mod::Module, code::String,  
filename::String)  
  @ Base ./loading.jl:1094
```

The error is on line 3; we tried to take the square root of a negative number. We know if the sum of the two shortest sides exceeds the largest side, then we cannot form a triangle. Let's add this as a check on our function, and while we are at it, also check for negative lengths!

```
1 function heron(a, b, c)  
2     # Let's include tests on the inputs to make  
3     # sure they correspond to a feasible triangle  
4     #  
5     data=sort([a; b; c])  
6     a=data[1]; b=data[2]; c=data[3]  
7     if a <= 0  
8         println("Cannot have sides with non-positive length")  
9         return im # return an imaginary number, just for fun  
10    elseif c > a + b  
11        println("Provided data does not form a triangle")  
12        return im # return an imaginary number, just for fun  
13    end
```

```

14 s = (a + b + c)/2 # semi-perimeter
15 Area = sqrt( s*(s-a) * (s-b) * (s-c) )
16 return Area
17 end

```

Output

heron (generic function with 1 method)

The above is a more professional version of our function.

```

1 area = heron(3,4,8)

```

Output

Provided data does not form a triangle

im

The sky is the limit! Here is a version of the function where we can specify the input data in two forms:

- as three points in \mathbb{R}^n and then compute the area of the triangle formed by the points, or
- as the lengths of the sides, as we did before.

```

1 using LinearAlgebra
2 function heron(A, B, C)
3     dim = maximum([length(A), length(B), length(C)])
4     if dim > 1
5         # Input consists of points in Rn for n >= 2
6         if (length(A) == length(B)) & (length(B) == length(C))
7             a=norm(A-B); b = norm(B-C); c = norm(C-A)
8             # Lengths will automatically form a triangle
9             s = (a + b + c)/2 # semi-perimeter
10            Area = sqrt( s*(s-a) * (s-b) * (s-c) )
11            return Area
12        else
13            println("Data is not consistent as points in Rn")
14            return im # return an imaginary number, just for fun
15        end
16    else
17        # Input consists of lengths of the sides
18        # Check the lengths to make sure
19        # they correspond to a feasible triangle
20        #
21        data=sort([A; B; C])
22        a=data[1]; b=data[2]; c=data[3]
23        if a <= 0
24            println("Cannot have sides with non-positive length")
25            return im # return an imaginary number, just for fun
26        elseif c > a + b
27            println("Provided data does not form a triangle")
28            return im # return an imaginary number, just for fun
29        end
30        s = (a + b + c)/2 # semi-perimeter
31        Area = sqrt( s*(s-a) * (s-b) * (s-c) )
32        return Area
33    end
34 end

```

Output

heron (generic function with 1 method)

```
1 heron([1 2 0], [2 3 5], [4 0 6])
```

Output

9.513148795220223

4.8 (Optional Read) More Examples Combining Functions and Flow Control

```
1 # Here is a function that takes in a square matrix
2 # and returns an upper triangular matrix, meaning one that
3 # has all zeros BELOW the diagonal
4 function makeTriangularMat(A)
5     B=copy(A)
6     nRows, nCols = size(B)
7     for k = 2: nRows
8         B[k,1:k-1] = 0.0 * B[k,1:k-1]
9     end
10    return B
11 end
```

Output

makeTriangularMat (generic function with 1 method)

```
1 using Random
2 Random.seed!(4321)
3 A=randn(5,5)
4 Atri=makeTriangularMat(A)
```

Output

```
5 x 5 Matrix{Float64}:
-0.2229071 -1.95979 -1.00753 -0.0688075 0.673963
 0.0 -0.991273 0.825056 1.51188 0.317936
 0.0 -0.0 -0.850184 -0.718068 -1.13022
 0.0 0.0 0.0 -1.06297 -0.0391889
-0.0 -0.0 -0.0 -0.0 0.609128
```

```
1 # Here is a function that takes in a rectangular matrix
2 # and returns an upper triangular matrix, meaning one that
3 # has all zeros BELOW the diagonal
4 function makeTriangularMat(A)
5     B=copy(A)
6     nRows, nCols = size(B)
7     for k = 2: nRows
8         K=minimum([k-1, nCols]) # min of the two numbers
9         B[k,1:K] = 0.0 * B[k,1:K]
10    end
11    return B
12 end
```

Output

makeTrinagularMat (generic function with 1 method)

Because of the additional line of code `K=minimum([k-1, nCols])`, our function `makeTriangularMat(A)` will now do the correct thing for rectangular matrices. The additional line of code prevents us from indexing into non-existent columns of the matrix because `K` can never be greater than `nCols`, the number of columns of our matrix.

```
1 # Our code works on non-square matrices
2 using Random
3 A=randn(5,4)
4 Atri=makeTrinagularMat(A)
```

Output

```
5x4 Matrix{Float64}:
-0.535644 -0.11      0.258952 -1.30922
 0.0      0.423592 -1.48126 -1.52763
 0.0      -0.0     1.25757 -0.215213
 0.0      0.0     -0.0     1.99146
-0.0      0.0     -0.0     0.0
```

```
1 # Our code works on non-square matrices
2 using Random
3 Random.seed!(4321)
4 A=randn(4,5)
5 Atri=makeTriangularMat(A)
```

Output

```
4 x 5 Matrix{Float64}:
-0.229071 -0.540755  1.02869 -0.850184  1.51188
 0.0      -1.95979 -1.94117  0.592798 -0.718068
 0.0      -0.0    -1.00753 -0.358643 -1.06297
 0.0      -0.0     0.0     -0.0688075 -0.509626
```

```
1 # Our code now works on non-square matrices
2 using Random
3 Random.seed!(4321)
4 A=randn(5,4)
5 Atri=makeTriangularMat(A)
```

Output

```
5x4 Matrix{Float64}:
-0.229071 -1.95979 -1.00753 -0.0688075
 0.0      -0.991273  0.825056  1.51188
 0.0      -0.0    -0.850184 -0.718068
 0.0      0.0     0.0     -1.06297
-0.0      -0.0     -0.0     -0.0
```

Our function `myAbs` only works for scalars. When we try it on a vector, we upset Julia.

```
1 myAbs([-pi 2])
```

Output

```
MethodError: no method matching isless(::Int64, ::Matrix{Float64})
Closest candidates are:
```

```

isless(::Any, ::Missing) at missing.jl:88
isless(::Missing, ::Any) at missing.jl:87
isless(::Real, ::AbstractFloat) at operators.jl:168
...

```

Stacktrace:

```

[1] <(x::Int64, y::Matrix{Float64})
  @ Base ./operators.jl:279
[2] <=(x::Int64, y::Matrix{Float64})
  @ Base ./operators.jl:328
[3] >=(x::Matrix{Float64}, y::Int64)
  @ Base ./operators.jl:352
[4] myAbs(a::Matrix{Float64})
  @ Main ./In[3]:3
[5] top-level scope
  @ In[4]:1
[6] eval
  @ ./boot.jl:360 [inlined]
[7] include_string(mapexpr::typeof(REPL.softscope), mod::Module, code::String,
filename::String)
  @ Base ./loading.jl:1094

```

We will now add in conditional checks to make it work for row vectors and column vectors. The first thing to ask ourselves, is how to check for a scalar? In the code below, we are checking that our input has length one and is not a `Vector` or a `Matrix`!

```

1 # Extend the your absolute value function to work on vectors
2 function myAbs(a)
3     B = (string(typeof(a))[1] !== 'V' ) & (string(typeof(a))[1] !== 'M' )
4     if (length(a) == 1) & B
5         if a >= 0
6             return a
7         else
8             return -a
9         end
10    else
11        y = copy(a) # Preserves TYPE meaning Vector or row Matrix
12        for i=1:length(a)
13            if a[i] < 0
14                y[i] = -a[i]
15            end
16        end
17        return y
18    end
19 end

```

Output

myAbs (generic function with 1 method)

The expression `(length(a) == 1)` is **T** when `a` is a scalar or a 1-element vector or a 1×1 matrix. In Line 3 of the code below, `B = T` when the Type of the input `a` does not equal `Vector` or `Matrix`. To break this down

- `string(typeof(a))` checks the Type of the variable `a` and then turns that into a string of characters. When `a = [-2 3.14159]`, the result is `"Matrix{Float64}"`. When `a = [-2; 3.14159]`, the result is `"Vector{Float64}"`. When `a = π`, the result is `"Irrational{π}"`.
- `string(typeof(a))[1]` picks off the first element of the string.
- The symbol “`!==`” means “NOT Equivalent to”

- $(\text{string}(\text{typeof}(a))[1] \neq 'V')$ is **T** when the first entry of the string is not a V, and thus we know that a is not a Vector. $(\text{string}(\text{typeof}(a))[1] \neq 'M')$ is **T** when the first entry of the string is not an M, and thus we know that a is not a Matrix.
- The symbol & is logical and.
- Hence, B is **T** when both $(\text{string}(\text{typeof}(a))[1] \neq 'V')$ and $(\text{string}(\text{typeof}(a))[1] \neq 'M')$ are **T**.
- When a has length one and is neither a vector nor a matrix, we assume it is already a scalar, and hence we can take its absolute value.
- Otherwise, a is a row vector ($1 \times n$ Matrix) or a column vector and we take the absolute value of each entry.

```

1 # Call the function
2 @show x=-2
3 @show string(typeof(x))[1]
4 @show myAbs(x)
5 @show x=[-2]
6 @show string(typeof(x))[1]
7 @show myAbs(x)
8 @show x=[-2; pi]
9 @show string(typeof(x))[1]
10 @show myAbs(x)
11 @show x=[-2 pi]
12 @show string(typeof(x))[1]
13 myAbs(x)

```

Output

```

x = -2 = -2
(string(typeof(x)))[1] = 'I'
myAbs(x) = 2
x = [-2] = [-2]
(string(typeof(x)))[1] = 'V'
myAbs(x) = [2]
x = [-2; pi] = [-2.0, 3.141592653589793]
(string(typeof(x)))[1] = 'V'
myAbs(x) = [2.0, 3.141592653589793]
x = [-2 pi] = [-2.0 3.141592653589793]
(string(typeof(x)))[1] = 'M'

1×2 Matrix{Float64}:
 2.0  3.14159

```


Chapter 5

Julia Lab 5: Linear Independence and LDLT Factorization, a Souped-up Version of LU

Learning Objectives

- Checking linear independence vs determining the number of linearly independent vectors
- What is LDLT?
- How can it be used to count the number of linearly independent vectors?
- Debugging, or finding and fixing errors in your code.

Outcomes

- Applying LU for checking linear Independence
- Matrices of the form $A^T A$ can always be factored as $L \cdot D \cdot L^T$, where L is uni-lower triangular and D is diagonal with non-negative entries
- Counting the number of linearly independent vectors in a set via LDLT
- Finding a maximal subset of linearly independent vectors

Either download Lab5 from our Canvas site or open up a Jupyter notebook so that you can enter code as we go. It is suggested that you have line numbering toggled on.

We've covered most of the Julia coding that we need for ROB 101. This lab will focus on (a) using LU Factorization for checking if a set of vectors is linearly independent or not, and (b) presenting an enhanced version of LU Factorization for separating a set of vectors into a first group of linearly independent vectors and a second group of vectors that are dependent on the first group. The enhanced form of LU factorization goes by the name of LDLT Factorization or the Cholesky Factorization.

5.1 Checking Linear Independence with LU

From Chapter 7 of the ROB 101 Textbook,

Pro-tip! Linear Independence in a Nutshell

Consider the vectors in \mathbb{R}^n ,

$$\left\{ v_1 = \begin{bmatrix} a_{11} \\ a_{21} \\ \vdots \\ a_{n1} \end{bmatrix}, v_2 = \begin{bmatrix} a_{12} \\ a_{22} \\ \vdots \\ a_{n2} \end{bmatrix}, \dots, v_m = \begin{bmatrix} a_{1m} \\ a_{2m} \\ \vdots \\ a_{nm} \end{bmatrix} \right\},$$

and use them as the columns of a matrix that we call A ,

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{21} & a_{22} & \cdots & a_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nm} \end{bmatrix}. \quad (5.1)$$

The following statements are equivalent:

- The set of vectors $\{v_1, v_2, \dots, v_m\}$ is linearly independent.
- The $m \times m$ matrix $A^\top \cdot A$ is invertible.
- $\det(A^\top \cdot A) \neq 0$.
- For any LU Factorization $P \cdot (A^\top \cdot A) = L \cdot U$ of $A^\top A$, the $m \times m$ upper triangular matrix U has no zeros on its diagonal.

Here is an implementation of the LU Factorization Algorithm.

```

1 using LinearAlgebra
2 function myLU(M::Array{<:Number, 2})
3     a, b = size(M)
4     n=min(a, b)
5     Temp = deepcopy(M)
6     L = Matrix{Float64}(undef, a, n)
7     U = Matrix{Float64}(undef, n, b)
8     epsilon=1e-12
9     P=zeros(a, a) + I
10    for k = 1:n
11        C = Temp[:, k] # k-th column
12        R = Temp[k:k, :] # k-th row
13        if maximum(abs.(C)) <= epsilon #column of zeros
14            C=0.0*C
15            C[k]=1.0
16            Temp=Temp-C*R
17            L[:, k]=C
18            U[k:k, :]=R

```

```

19     else # put the biggest entry to the top
20         ii=argmax( abs.(C) )
21         nrow=ii[1]
22         # Do row permutations
23         P[[k, nrow], :]=P[[nrow, k], :]
24         Temp[[k, nrow], :]=Temp[[nrow, k], :]
25         if k>1
26             L[[k, nrow], :]=L[[nrow, k], :]
27         end
28         C = Temp[:, k] # k-th column
29         pivot = C[k]
30         C=C/pivot #normalize all entieres by C[i]
31         R = Temp[k:k, :] # k-th row
32         Temp=Temp-C*R
33         L[:, k]=C
34         U[k:k, :]=R
35     end
36 end
37 return L, U, P
38 end

```

Output

myLU (generic function with 1 method)

We generate two matrices. One that we know will be linearly dependent because we have four vectors in \mathbb{R}^3 and another we expect to be linearly independent because we have four random vectors in \mathbb{R}^5 .

```

1 using Random
2 Random.seed!(3141596)
3 A=randn(3,4)
4 B=randn(5,4)

```

Output

```

5×4 Matrix{Float64}:
-1.45267  0.0432641  0.562999  0.0499658
-0.271694 -0.426124 -1.58141 -0.457054
 0.371157  1.99933 -0.0944284 -2.27898
 0.950984  1.19788 -0.588755  1.61326
 0.422272 -0.440388  1.13876 -0.70359

```

Once again, we know that the columns of A must be linearly dependent because we have four vectors in \mathbb{R}^3 . If the four columns of A were linearly independent, then \mathbb{R}^3 would have at least dimension four. But we know that \mathbb{R}^3 has dimension three, and hence the set of four vectors must be dependent. From the **Pro-tip!**, to evaluate (check) the linear independence of the columns of A , we can perform the LU factorization of $A^T \cdot A$ and inspect the diagonal of U for zeros.

```

1 L, U, P = myLU(A'*A)
2 U

```

Output

```

4×4 Matrix{Float64}:
 7.00075  6.54966 -2.93873  1.27537
 0.0     -0.446701 -0.0255008  0.928349
 0.0     0.0      0.386878  0.56443
 0.0     0.0      0.0      1.05471e-15

```

$U[4, 4] = 1.05471e-15$ is numerically zero and thus we see that the columns of A are **linearly dependent**. Just for the fun of it, we'll give a function that "cleans up" the almost zero entries of a matrix or vector and makes them equal to 0.0. We'll set the tolerance as $1e-10$

```

1 function cleanUp(A, tol=1e-10)
2     # Zero out small entries of a matrix or vector
3     B=copy(A)
4     indicesSmall=findall(x->x<tol, abs.(B))
5     B[indicesSmall]=0.0*B[indicesSmall]
6 return B
7 end
8 #
9 cleanUp(U)

```

Output

```

4×4 Matrix{Float64}:
 7.00075  6.54966 -2.93873  1.27537
 0.0      -0.446701 -0.0255008  0.928349
 0.0      0.0       0.386878   0.56443
 0.0      0.0       0.0        0.0

```

Now, there is clearly a zero on the diagonal of U and hence the columns of A are not linearly independent.

The function `cleanUP` uses the Julia function `findall` that allows you to find the indices of all values of an array that satisfy a criterion that you set.

```

1 ? findall

```

Output

It's super long. You can check it out yourself if interested.

Now we check the matrix B ,

```

1 L, U, P = myLU(B'*B)
2 cleanUp(U)

```

Output

```

4×4 Matrix{Float64}:
 3.40451  1.74819 -0.50227  0.442818
 0.0      4.91194 -0.439397 -2.34454
 0.0      0.0     4.35675  -0.929314
 0.0      0.0     0.0      7.12791

```

For the matrix B , there are no zeros on the diagonal of U and hence the columns of B are linearly independent.

5.2 LU is a Suboptimal Tool for Determining the Number of Linearly Independent Vectors in a Set

A more advanced question than linear independence is **how many linearly independent vectors** are there in a given set of vectors? An even more advanced question is **which ones are they?** In this section, we'll provide a first treatment of these questions.

We consider the 5×5 matrix

$$A := \begin{bmatrix} 0.8814 & 1.1974 & 2.5033 & -1.8788 & -0.9375 \\ -0.4241 & 1.1761 & 2.1315 & -3.5430 & 0.8276 \\ 1.0306 & -0.3120 & -0.3325 & 2.1482 & -1.4951 \\ -0.1551 & 2.9496 & 5.5761 & -7.6896 & -0.1905 \\ -1.2632 & -0.1046 & -0.5197 & -1.3970 & 0.7202 \end{bmatrix} \quad (5.2)$$

We check `myLU(AT · A)`.

```
1 L, U, P = myLU(A' * A)
2 cleanUp(diag(U))
```

Output

```
5-element Vector{Float64}:
 5.018367341067237
26.544273902523273
-0.0
-0.0
-0.0
```

We conclude that the columns of A are not linearly independent. Now, we ask, how many of the columns of A can we choose and still have a linearly independent set? Looking at the diagonal of U , you may be tempted to say two! **But you would be wrong!**

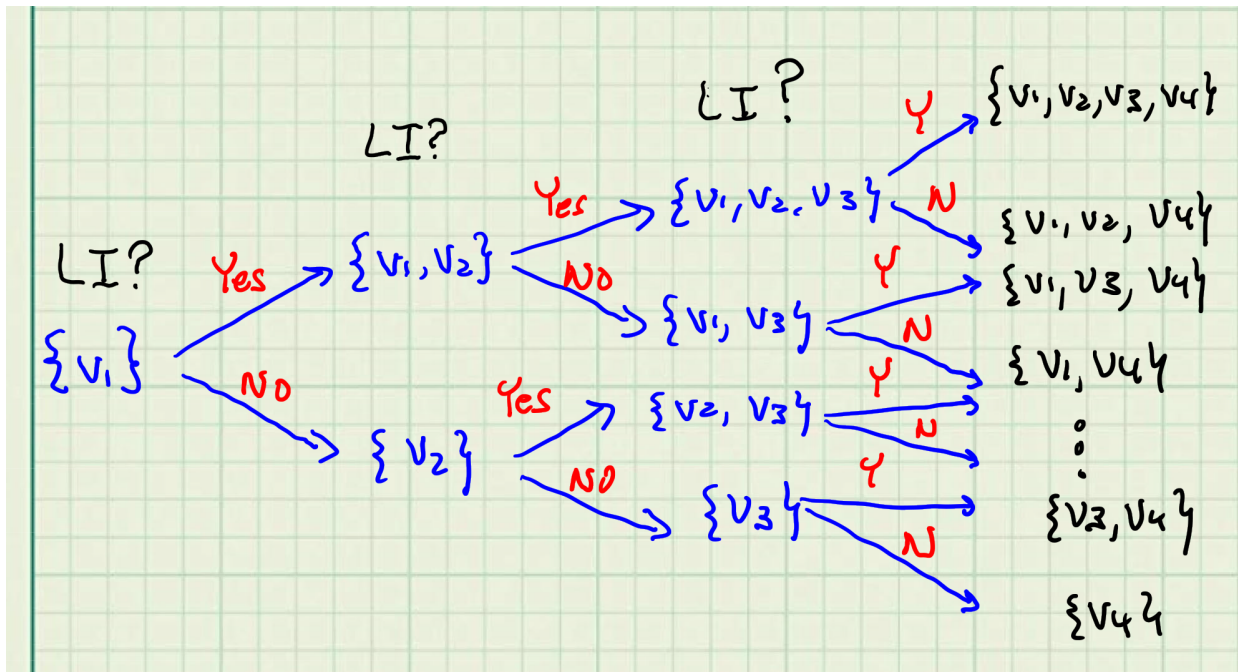


Figure 5.1: Checking linear independence from left to right. You could also start from the right and go to the left, or you could start in the middle and proceed to the two ends. You just need to do an organized search of the vectors!

In the following, we will implement the process indicated in Fig. 5.1, which shows one way to systematically check linear independence of a set of vectors to arrive at a maximal subset of linearly independent vectors. Along the way, we'll also keep track of the indices of the vectors that are linearly independent.

```
1 function mySearchRoutine4NumIndepVectors(A, atol=1e-10)
2     nRowsA, nColsA = size(A)
3     V=Matrix{Float64}(undef, nRowsA, 0) # create an empty matrix
```

```

4                                     # for storing independent vectors
5 numIndVectors= 0
6 indices = Vector{Int64}(undef, 0) # create an empty vector for storing
7                                     # the indices of independent vectors
8 for i = 1 : nColsA
9     Temp=[V A[:,i]]
10    L, U, P = myLU(Temp'*Temp)
11    minDiagU = minimum(abs.(diag(U)))
12    # If minDiagU > atol, then U is invertible and hence the columns
13    # of Temp are linearly independent
14    if minDiagU > atol
15        numIndVectors = numIndVectors + 1
16        V = Temp
17        indices = [indices; i] # keep track of indep vectors
18    end
19 end
20 return numIndVectors, V, indices
21 end

```

Output

mySearchRoutine4NumIndepVectors (generic function with 2 methods)

We next show that A has three linearly independent columns and we can choose them as the first, second, and fifth columns.

```
1 numIndVectors, V, indices = mySearchRoutine4NumIndepVectors(A)
```

Output

```
3, [0.8814395099307569 1.1974345124883454 -0.9375085811788099; -0.4241418472936163
1.1761409436174899 0.8276128787991263; ... ; -0.15514669800915096 2.949627319800578
-0.1905215623370098; -1.2632013933716764 -0.10460087472162416 0.7201716618604121],
[1, 2, 5])
```

```
1 numIndVectors
```

Output

```
3
```

```
1 V
```

Output

```
5×3 Matrix{Float64}:
 0.88144  1.19743  -0.937509
-0.424142  1.17614  0.827613
 1.03061  -0.312031  -1.49514
-0.155147  2.94963  -0.190522
-1.2632   -0.104601  0.720172
```

```
1 indices
```

Output

```
3-element Vector{Int64}:
 1
 2
 5
```

Could we have made other choices for three linearly independent vectors? Absolutely. The following shows that we could have taken the last three vectors.

```
1 numIndVectors, V, indices = mySearchRoutine4NumIndepVectors (A[:, 3:5])
```

Output

```
(3, [2.503340396444761 -1.8787774833565118 -0.9375085811788099; 2.1314687886946517
-3.5429649155470724 0.8276128787991263; ... ; 5.576055157377987 -7.68964057029396
-0.1905215623370098; -0.5197147429767035 -1.3969985094856063 0.7201716618604121],
[1, 2, 3])
```

Could we have taken any three vectors? Let's check! We'll do a brute force search to find all combinations of three vectors that are linearly **dependent**, if any. Note that we can assume without loss of generality that $i < j < k$ when we consider three columns $\{A_i, A_j, A_k\}$ of A . Why? Because order does not change linear independence.

This code block outputs all combinations of three vectors that are linearly dependent.

```
1 for i = 1:3
2     for j = i+1:5
3         for k = j+1:5
4             numIndVectors, V, indices = mySearchRoutine4NumIndepVectors (A[:, [i;j;k]])
5             if numIndVectors < 3
6                 @show [i;j;k] # dependent columns
7             end
8         end
9     end
10 end
```

Output

```
[i; j; k] = [1, 2, 3]
[i; j; k] = [1, 2, 4]
[i; j; k] = [1, 3, 4]
[i; j; k] = [2, 3, 4]
```

Once we know all combinations of columns of A that are linearly dependent, we can deduce that the following sets of vectors are linearly independent.

- $\{A_1, A_2, A_5\}$
- $\{A_1, A_3, A_5\}$
- $\{A_2, A_3, A_5\}$
- $\{A_2, A_4, A_5\}$
- $\{A_3, A_4, A_5\}$

5.3 LDLT Factorization: A One-shot Means to Find Linearly Independent Vectors in a Set

Once again, consider the vectors in \mathbb{R}^n ,

$$\left\{ v_1 = \begin{bmatrix} a_{11} \\ a_{21} \\ \vdots \\ a_{n1} \end{bmatrix}, v_2 = \begin{bmatrix} a_{12} \\ a_{22} \\ \vdots \\ a_{n2} \end{bmatrix}, \dots, v_m = \begin{bmatrix} a_{1m} \\ a_{2m} \\ \vdots \\ a_{nm} \end{bmatrix} \right\},$$

and use them as the columns of a matrix that we call A ,

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{21} & a_{22} & \cdots & a_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nm} \end{bmatrix}. \quad (5.3)$$

Uber Pro-Tip: Number of Linearly Independent Vectors via an Enhanced LU Factorization

Assume that a set of vectors has been stacked to form the columns of an $n \times m$ matrix A as in (5.3). **Fact:** The matrix $A^\top \cdot A$ always has an **LDLT Factorization**

$$P \cdot A^\top \cdot A \cdot P^\top = L \cdot D \cdot L^\top, \quad (5.4)$$

where

- P is a (row) permutation matrix;
- P^\top , the transpose of P , permutes the columns of $A^\top A$;
- L is uni-lower triangular and L^\top , the transpose of L , is therefore uni-upper triangular; and
- D is diagonal and has non-negative entries.

Moreover,

- **the number of linearly independent columns of A is equal to the number of non-zero entries on the diagonal of D ;** and, if we denote this number by k ,
- **then for the version of the LDLT given below, the first k -columns of $A \cdot P^\top$ are linearly independent, and the remaining $(m - k)$ -columns (if any) are linearly dependent on the first k columns.**
- Because the columns of $A \cdot P^\top$ are simply the columns of A permuted by P^\top (that is, re-ordered by the permutation matrix), **the first k -columns of $A \cdot P^\top$ provide a selection of linearly independent columns of A .**

While the derivation of this algorithm is not so different than what we did for the LU Factorization, we are still going to skip it. The main point is that matrices of the form $A^\top \cdot A$ always have an LDLT Factorization and, moreover, the diagonal of D and the permutation matrix P tell you everything you need to know about the linearly independent vectors that make up the columns of A .

```

1 function ldlTROB101 (A)
2     epsilon = 1e-12
3     M = A' * A
4     n, m = size (A)
5     A_reduced = M
6     L = Array{Float64, 2} (undef, m, 0)
7     Id = zeros (m, m) + I
8     P = Id
9     D=zeros (m, m)
10    for i=1:m
11        ii=argmax (diag (A_reduced [i:m, i:m]))
12        mrow=ii [1] + (i-1)
13        if ! (i==mrow)
14            P [ [i, mrow], : ] = P [ [mrow, i], : ]
15            A_reduced [ [i, mrow], : ] = A_reduced [ [mrow, i], : ]
16            A_reduced [ :, [i, mrow] ] = A_reduced [ :, [mrow, i] ]
17        end
18        if (i>1)
19            L [ [i, mrow], : ] = L [ [mrow, i], : ]
20        end
21        pivot=A_reduced [i, i]
22        if ! isapprox (pivot, 0, atol=epsilon)

```



```

23     D[i,i]=pivot
24     C=Areduced[:,i]/pivot
25     L=[L C]
26     Areduced=Areduced-(C*pivot*C')
27     else
28         L=[L Id[:,i:m]]
29         break
30     end
31 end
32 diagD=diag(D)
33 return L,P,D,diagD
34 end

```

Output

ldltROB101 (generic function with 1 method)

We apply LDLT to the same 5×5 matrix we used before, namely

$$A := \begin{bmatrix} 0.8814 & 1.1974 & 2.5033 & -1.8788 & -0.9375 \\ -0.4241 & 1.1761 & 2.1315 & -3.5430 & 0.8276 \\ 1.0306 & -0.3120 & -0.3325 & 2.1482 & -1.4951 \\ -0.1551 & 2.9496 & 5.5761 & -7.6896 & -0.1905 \\ -1.2632 & -0.1046 & -0.5197 & -1.3970 & 0.7202 \end{bmatrix} \quad (5.5)$$

```

1 L,P,D,diagD = ldltROB101(A)
2 @show diagD
3 D

```

Output

diagD = [81.779373907533, 5.130204240478989, 0.7812212873817588, 0.0, 0.0]

```

5x5 Matrix{Float64}:
 81.7794  0.0    0.0    0.0  0.0
  0.0    5.1302  0.0    0.0  0.0
  0.0    0.0    0.781221  0.0  0.0
  0.0    0.0    0.0    0.0  0.0
  0.0    0.0    0.0    0.0  0.0

```

We see immediately that we have three linearly independent vectors. Can we find easily a set of three columns of A that are linearly independent? Yes. If we form $A \cdot P^T$, the **Uber Pro-tip** tells us that then the first three columns will be linearly independent.

```

1 Abar=A*P'

```

Output

```

5x5 Matrix{Float64}:
-1.87878  2.50334  -0.937509  1.19743  0.88144
-3.54296  2.13147  0.827613  1.17614  -0.424142
 2.14821  -0.332486  -1.49514  -0.312031  1.03061
-7.68964  5.57606  -0.190522  2.94963  -0.155147
-1.397    -0.519715  0.720172  -0.104601  -1.2632

```

$$\bar{A} := A \cdot P = \begin{bmatrix} -1.8788 & 2.5033 & -0.9375 & 1.1974 & 0.8814 \\ -3.5430 & 2.1315 & 0.8276 & 1.1761 & -0.4241 \\ 2.1482 & -0.3325 & -1.4951 & -0.3120 & 1.0306 \\ -7.6896 & 5.5761 & -0.1905 & 2.9496 & -0.1551 \\ -1.3970 & -0.5197 & 0.7202 & -0.1046 & -1.2632 \end{bmatrix} \quad (5.6)$$

```

1 numIndVectors, V, indices = myNumIndepVectors (Abar[:, 1:3])
2 numIndVectors

```

Output

3

Which columns are they? From the permutation matrix, we can see that LDLT has selected the fourth, third, and fifth columns of A , $\{A_4, A_3, A_5\}$. The order is not very important; it's a consequence of my implementation using the largest available pivot element at each pass of "peeling the onion".

```

1 P'

```

Output

```

5×5 adjoint(::Matrix{Float64}) with eltype Float64:
 0.0  0.0  0.0  0.0  1.0
 0.0  0.0  0.0  1.0  0.0
 0.0  1.0  0.0  0.0  0.0
 1.0  0.0  0.0  0.0  0.0
 0.0  0.0  1.0  0.0  0.0

```

Why this order? Once again, the way I wrote the LDLT algorithm, it permutes large elements on the diagonal of $A^T \cdot A$ to the front.

Below is another version of a function to find the number of linear independent columns in a matrix, along with a specification of which columns to use.

```

1 function myNumIndepVectors (A, myTol=1e-10)
2     nRowsA, nColsA = size (A)
3     V=Matrix{Float64} (undef, nRowsA, 0)
4     L, P, D, diagD = ldlTROB101 (A)
5     indicesLDLT=findall (x->x>myTol, diagD)
6     numIndVectors = length (indicesLDLT)
7     Ptrans = P'
8     indices = Vector{Int64} (undef, numIndVectors)
9     for i = 1:numIndVectors
10         ind=argmax (Ptrans[:, i])
11         indices[i]=ind[1]
12     end
13     V=A[:, indices]
14
15     return numIndVectors, V, indices
16 end

```

Output

```
myNumIndepVectors (generic function with 2 methods)
```

```

1 numIndVectors, V, indices = myNumIndepVectors (A)
2 @show numIndVectors
3 @show indices
4 V

```

Output

```

numIndVectors = 3
indices = [4, 3, 5]

```

```
5×3 Matrix{Float64}:
-1.87878  2.50334  -0.937509
-3.54296  2.13147   0.827613
 2.14821 -0.332486  -1.49514
-7.68964  5.57606  -0.190522
-1.397    -0.519715  0.720172
```

5.4 Debugging

Here, we'll focus on the LU Function in Julia. Let's build a solver for $Ax = b$, when A is square and invertible, and A and b have compatible sizes. We assume that the functions `forwardsub(L, b)` and `backwardsub(U, b)` have already been defined and error checked!

Let's include some packages and create some data.

```
1 using LinearAlgebra
2 using Random
3 Random.seed!(2001)
4
5 A = randn(4, 4)
6 b=randn(4, 1);
```

Output Nothing due to the semicolon. We next write our first cut at a function to solve $Ax = b$

```
1 function mySolver(A, b)
2     L, U = lu(A)
3     y = forwardsub(L, b)
4     x = backwardsub(U, y)
5     return x
6     return x
7 end
```

Output

```
mySolver (generic function with 1 method)
```

We run our function and it seems to work because it does produce output!

```
1 x = mySolver(A, b)
```

Output

```
4-element Vector{Float64}:
 0.2793805449608082
 2.6408473207373446
-2.3464083290282343
-0.7884784181920488
```

About this time, you run one of our Friendly Checks and fail it. And you go, no way, my function works; it produces no errors. Look, I have actual output! So you post to Piazza, saying hey, I failed the Friendly Check! What's wrong with my code or with the Friendly Check itself?

About this time of the semester, we start asking you to do some serious debugging yourself. So, let's do it.

We first check if the solution is correct.

```
1 norm(A*x-b)
```

Output

1.5606151971175222

Oops! That is not close to zero. What's wrong? Well, on line 2 of your function, you called Julia's LU function with permutations! OH my gosh, you're right. Let's include the permutation matrix. So we do it.

```
1 function mySolver(A, b)
2     L, U, P = lu(A)
3     y = forwardsub(L, P*b)
4     x = backwardsub(U, y)
5     return x
6     return x
7 end
```

Output

mySolver (generic function with 1 method)

Everything looks good. We give it a whirl....and Julia is super unhappy.

```
1 x = mySolver(A, b)
```

Output

DimensionMismatch("matrix A has dimensions (4,1), matrix B has dimensions (4,1)")

Stacktrace:

```
[1] _generic_matmatmul!(C::Matrix{Float64}, tA::Char, tB::Char, A::Matrix{Int64},
B::Matrix{Float64}, _add::LinearAlgebra.MulAddMul{true, true, Bool, Bool})
 @ LinearAlgebra /buildworker/worker/package_linux64/build/usr/share/julia/stdlib/v1.6/
LinearAlgebra/src/matmul.jl:814
[2] generic_matmatmul!(C::Matrix{Float64}, tA::Char, tB::Char, A::Matrix{Int64},
B::Matrix{Float64}, _add::LinearAlgebra.MulAddMul{true, true, Bool, Bool})
 @ LinearAlgebra /buildworker/worker/package_linux64/build/usr/share/julia/stdlib/
v1.6/LinearAlgebra/src/matmul.jl:802
[3] mul!
 @ /buildworker/worker/package_linux64/build/usr/share/julia/stdlib/
v1.6/LinearAlgebra/src/matmul.jl:302 [inlined]
[4] mul!
 @ /buildworker/worker/package_linux64/build/usr/share/julia/stdlib/
v1.6/LinearAlgebra/src/matmul.jl:275 [inlined]
[5] *
 @ /buildworker/worker/package_linux64/build/usr/share/julia/stdlib/
v1.6/LinearAlgebra/src/matmul.jl:153 [inlined]
[6] *
 @ /buildworker/worker/package_linux64/build/usr/share/julia/stdlib/
v1.6/LinearAlgebra/src/matmul.jl:63 [inlined]
[7] mySolver(A::Matrix{Float64}, b::Matrix{Float64})
 @ Main ./In[24]:3
[8] top-level scope
 @ In[25]:1
[9] eval
 @ ./boot.jl:360 [inlined]
[10] include_string(mapexpr::typeof(REPL.softscope), mod::Module,
code::String, filename::String)
 @ Base ./loading.jl:1094
```

We have a **DimensionMismatch("matrix A has dimensions (4,1), matrix B has dimensions (4,1)")** and it seems to be on line 3 of our function; we get that from `@ Main ./In[14]:3`. So, we immediately suspect the function `backwardsub` has an error. However, we've used it many times before, and the line

Stacktrace:

```
[1] _generic_matmatmul!(C::Matrix{Float64}, tA::Char, tB::Char, A::Matrix{Int64},  
B::Matrix{Float64}, _add::LinearAlgebra.MulAddMul{true, true, Bool, Bool})
```

could tell us that we have a matrix multiplication problem.

Hence, let's use the `display` function to see what's up with that line of code. To save space here, we also include the function call on line 13. The output looks great, until it doesn't!

```
1 function mySolver(A, b)  
2     L, U, P = lu(A)  
3     display(L)  
4     display(U)  
5     display(P)  
6     display(P*b)  
7     y = forwardsub(L, P*b)  
8     x = backwardsub(U, y)  
9     return x  
10    return x  
11 end  
12  
13 x = mySolver(A, b)
```

Output

```
4×4 Matrix{Float64}:  
 1.0      0.0      0.0      0.0  
-0.374657 1.0      0.0      0.0  
-0.841164 -0.808034 1.0      0.0  
-0.207949 -0.591682 0.91447  1.0  
4×4 Matrix{Float64}:  
-1.61455  0.810841  0.867077 -0.318762  
 0.0      -1.66425  -0.958626 -1.56293  
 0.0      0.0      0.811276  -1.17566  
 0.0      0.0      0.0      -0.920034  
4-element Vector{Int64}:  
 1  
 3  
 4  
 2
```

```
DimensionMismatch("matrix A has dimensions (4,1), matrix B has dimensions (4,1)")
```

Stacktrace:

```
[1] _generic_matmatmul!(C::Matrix{Float64}, tA::Char, tB::Char, A::Matrix{Int64},  
B::Matrix{Float64}, _add::LinearAlgebra.MulAddMul{true, true, Bool, Bool})  
  @ LinearAlgebra /buildworker/worker/package_linux64/build/usr/share/  
  julia/stdlib/v1.6/LinearAlgebra/src/matmul.jl:814  
[2] generic_matmatmul!(C::Matrix{Float64}, tA::Char, tB::Char, A::Matrix{Int64},  
B::Matrix{Float64}, _add::LinearAlgebra.MulAddMul{true, true, Bool, Bool})  
  @ LinearAlgebra /buildworker/worker/package_linux64/build/usr/share/julia/stdlib/  
  v1.6/LinearAlgebra/src/matmul.jl:802  
[3] mul!  
  @ /buildworker/worker/package_linux64/build/usr/share/julia/stdlib/  
  v1.6/LinearAlgebra/src/matmul.jl:302 [inlined]  
[4] mul!  
  @ /buildworker/worker/package_linux64/build/usr/share/julia/stdlib/  
  v1.6/LinearAlgebra/src/matmul.jl:275 [inlined]
```

```

[5] *
   @ /buildworker/worker/package_linux64/build/usr/share/julia/stdlib/v1.6/
   LinearAlgebra/src/matmul.jl:153 [inlined]
[6] *
   @ /buildworker/worker/package_linux64/build/usr/share/julia/stdlib/
   v1.6/LinearAlgebra/src/matmul.jl:63 [inlined]
[7] mySolver(A::Matrix{Float64}, b::Matrix{Float64})
   @ Main ./In[27]:6
[8] top-level scope
   @ In[27]:13
[9] eval
   @ ./boot.jl:360 [inlined]
[10] include_string(mapexpr::typeof(REPL.softscope), mod::Module,
code::String, filename::String)
   @ Base ./loading.jl:1094

```

From `Main ./In[27]:6`, we see that we have an error on line 6. Also, looking at the displayed results, we note that the permutation matrix P is a vector. Yikes! What is that all about? **$L, U, P = \text{lu}(A)$ returns, L, U , and the list of permutation indices, but not the permutation matrix. To obtain the matrix, you need to use**

```

1 function mySolver(A, b)
2     F = lu(A)
3     U=F.U
4     L=F.L
5     P=F.P
6     y = forwardsub(L, P*b)
7     x = backwardsub(U, y)
8     return x
9 end
10
11 x = mySolver(A, b)
12 display(x)
13 norm(A*x- b)

```

Output

```

4-element Vector{Float64}:
-0.22542848860215622
-2.222759959807066
 2.0136373165391372
 1.2566870405250985

7.2815418263566e-16

```

Bottom line: We learned the following:

- Just because a function runs with no errors does not mean it is correct. You, the creator of the function, need to run some simple tests to check the correctness of your function. Here, we simply applied the definition of x being a solution to $Ax = b$, namely, $\|Ax - b\| = 0$.
- There is useful information in the error messages. It takes practice to interpret them. At the very least, you can find the **line number** where things failed.
- Once you know approximately what is failing, insert a copious number of `display` or `@show` commands and compare what you see to what you expect.
- Your checks can be much more focused than the generic Friendly Checks that we insert because you get to see where the error is occurring.

5.5 (Optional Read) Finding Counterexamples via Search

How did we find a matrix where LU would mess up? We did a brute force search!

```
1 using LinearAlgebra
2 using Random
3 function CounterExample (atol=1e-10)
4     # Set the seed because we want each student to obtain the same results
5     Random.seed! (12121212)
6     flag = 1
7     N=5
8     n=floor (Int, N/2)
9     k=0
10    while flag > 0
11        # count how many times through the loop
12        k=k+1
13        Random.seed! (k)
14        # Build a matrix A that has dependent column vectors
15        B=randn (N, N-1)
16        C=randn (n, n)
17        A=[B[:, 1:n] B[:, 1:n]*C B[:, n+n:end]]
18        # Apply LU to A'A
19        F=lu (A'*A, check=false)
20        diagU=diag (F.U)
21        # Find all indices where the magnitude of the diagonal element is not too small
22        indicesLU=findall (x->x>atol, abs. (diagU))
23        # Now count them. This is LU's prediction of the number of linearly independent
24        vectors
25        NumLinIndepLU=length (indicesLU)
26        # Apply LDLT to A'A; recall, our function forms A'*A
27        L, P, D, diagD = ldltROB101 (A)
28        # Find all indices where the magnitude of the diagonal element is not too small
29        indicesLDLT=findall (x->x>atol, diagD)
30        # Now count them. This is LDLT's prediction of the number of linearly
31        independent vectors
32        NumLinIndepLDLT=length (indicesLDLT)
33        # Check for a discrepancy in their reported number of linearly indep columns
34        # and also, do not run forever. 1e5 seems like enough times to loop through
35        random matrices
36        if (NumLinIndepLDLT > NumLinIndepLU) || (k>1e5)
37            return k, A, F, L, P, D, diagD
38            flag=0
39        end
40    end
41end
42
43function cleanUp (A, tol=1e-10)
44    # Zero out small entries of a matrix or vector
45    B=copy (A)
46    indicesSmall=findall (x->x<tol, abs. (B))
47    B [indicesSmall]=0.0*B [indicesSmall]
48end
49return B
50end
```

Output

CounterExample (generic function with 2 methods)

```
1 k, A, L, P, D, diagD = CounterExample()
2 println("It took us $k random matrices to generate an example where LU fails.")
3 println("Let's explore the example a bit.")
4 A
```

Output

It took us 3 random matrices to generate an example where LU fails.

```
7×7 Matrix{Float64}:
 1.19156  -0.00197414  -0.51281  ...  -1.06384   1.63691   0.633455
-2.51973   1.00879   -1.58024   2.81498  -1.4239   0.68743
 2.07481   0.844223   0.511659  -3.14415  2.06393  -0.135094
-0.97325   1.15807   0.733867  -0.118041 -1.31149  0.492822
-0.101607 -0.475159  -1.40705   1.13615   0.756238  0.15671
-1.54251  -0.244612  -0.707273  ...   2.21941  -1.23019  1.23958
 0.100793  0.0718727  -1.1661    0.40164   0.923609 -1.21044
```


Chapter 6

Julia Lab 6: Matrix Null Space and Linear Regression

Learning Objectives

- Null space of a matrix and relation to linear independence
- Regression
- Debugging, or finding and fixing errors in your code

Outcomes

- A new way to check for linear independence
- A brief introduction to basis vectors
- Practice with a linear algebra Super Power, Linear Regression
- Practice with “broadcasting”
- Practice with plotting
- Knowledge that you can use linear regression to fit nonlinear functions to “curved data” as long as the unknown coefficients enter in a linear manner

Either download Lab6 from our Canvas site or open up a Jupyter notebook so that you can enter code as we go. It is suggested that you have line numbering toggled on.

We've covered most of the Julia coding that we need for ROB 101. This lab will focus on the null space of a matrix and linear regression.

6.1 Null Space of a Matrix

The null space of a matrix can be used to find linear combinations of vectors that add up to zero. Specifically, consider a set of vectors $\{v_1, v_2, \dots, v_m\} \subset \mathbb{R}^n$ and form the $n \times m$ matrix

$$A = [v_1 \ v_2 \ \dots \ v_m].$$

From lecture, the **null space of A** is

$$\text{null}(A) := \{x \in \mathbb{R}^m \mid Ax = 0_{n \times 1}\}.$$

If $x \in \text{null}(A)$ and $x \neq 0_{m \times 1}$, then we have a set of non-zero coefficients such

$$x_1 v_1 + x_2 v_2 + \dots + x_m v_m = 0_{n \times 1}$$

which means the set of vectors is **linearly dependent**. On the other hand, if the null space of A consists only of the zero vector, then the set of vectors is **linearly independent**.

Remark 6.1 *If you do not like denoting the coefficients in the linear combination by x_i , then we can use α_i instead and write*

$$A \cdot \alpha = 0_{n \times 1} \iff \alpha_1 v_1 + \alpha_2 v_2 + \dots + \alpha_m v_m = 0_{n \times 1}.$$

```

1 # The nullspace command in Julia LinearAlgebra package
2 using LinearAlgebra
3
4 A = [1 2 3 4; 5 6 7 8; 9 10 11 12]
5 V = nullspace(A)

```

Output

```

4×2 Matrix{Float64}:
 0.261535  -0.481248
-0.707414   0.446727
 0.630224   0.550289
-0.184345  -0.51576

```

```

1 v1 = V[:, 1]
2 v2 = V[:, 2]
3 @show A*v1 # needs to be zero for v1 to be in the null space
4 @show A*v2 # similar.

```

Output

```

A * v1 = [-1.1102230246251565e-16, 2.220446049250313e-16, 2.220446049250313e-15]
A * v2 = [0.0, -8.881784197001252e-16, -8.881784197001252e-16]

3-element Vector{Float64}:
 0.0
-8.881784197001252e-16
-8.881784197001252e-16

```

We check that vectors in the null space tell us how to form linear combinations of the columns of A that add up to zero.

```

1 # Linear combination of the columns of A that adds up to zero
2 a1=V[1,1]
3 a2=V[2,1]
4 a3=V[3,1]
5 a4=V[4,1]
6 #
7 a1*A[:,1] + a2*A[:,2] + a3*A[:,3] + a4*A[:,4]

```

Output

```

3-element Vector{Float64}:
 -1.1102230246251565e-16
  2.220446049250313e-16
  2.220446049250313e-15

```

```

1 # Linear combination of the columns of A that adds up to zero
2 a1=V[1,2]
3 a2=V[2,2]
4 a3=V[3,2]
5 a4=V[4,2]
6 #
7 a1*A[:,1] + a2*A[:,2] + a3*A[:,3] + a4*A[:,4]

```

Output

```

3-element Vector{Float64}:
  0.0
 -8.881784197001252e-16
 -8.881784197001252e-16

```

We also recall from lecture that the null space of a matrix is a subspace. Indeed, $v_i \in \text{null}(A) \iff Av_i = 0_{n \times 1}$, and thus, if we consider a linear combination of vectors in the null space of A , and apply A to it, we get the zero vector. Indeed,

$$A \cdot (\alpha_1 v_1 + \alpha_2 v_2) = \alpha_1 A \cdot v_1 + \alpha_2 A \cdot v_2 = \alpha_1 0_{n \times 1} + \alpha_2 0_{n \times 1} = 0_{n \times 1}.$$

So, if a null space is a subspace, why is the command `nullspace` only returning two vectors?

```

1 # The nullspace command in Julia LinearAlgebra package
2 using LinearAlgebra
3
4 A = [1 2 3 4; 5 6 7 8; 9 10 11 12]
5 println("If a null space is a subspace, why is the command only returning two vectors?")
6 V = nullspace(A)

```

Output

If a null space is a subspace, why is the command only returning two vectors?

```

4×2 Matrix{Float64}:
 0.261535 -0.481248
 -0.707414  0.446727
 0.630224  0.550289
 -0.184345 -0.51576

```

The `nullspace` command in Julia is returning a “basis” for the subspace. When you are first reading this, we may not have covered the concept of basis vectors in lecture. Hence, we give a preview of the definition here.

Preview of Basis Vectors

We are delaying the formal study of basis vectors until Chapter 10 in our textbook. In the meantime, it helps to have an intuitive understanding:

- A set of vectors $\{v_1, v_2, \dots, v_m\} \subset V$ is a **basis for V** if
 - (a) $\{v_1, v_2, \dots, v_m\}$ is linearly independent
 - (b) $x \in V \iff$ there exist coefficients $\alpha_1, \dots, \alpha_m$ such that $x = \alpha_1 v_1 + \dots + \alpha_m v_m$.
- Personally, your instructor likes to think of a basis $\{v_1, v_2, \dots, v_m\} \subset V$ in the following way:
 - (a) The set $\{v_1, v_2, \dots, v_m\}$ is **big enough to generate V by forming linear combinations**, and
 - (b) **small enough to be linearly independent**.

A Goldilocks interpretation of a basis for a subspace V : a basis is a set of vectors that is not too big and not too small.

- If every $x \in V$ can be expressed as a linear combination of elements of $\{v_1, v_2, \dots, v_m\}$, then $\{v_1, v_2, \dots, v_m\}$ is **big enough** to “generate” all of V .
- If the set $\{v_1, v_2, \dots, v_m\}$ is linearly independent, then it is **small enough** that you cannot discard elements without changing the number of linearly independent vectors.
- Hence, a basis is “**just right**” in terms of the quantity of vectors you need to build every vector in the subspace through the operation of forming linear combinations.

Another Test for Linear Independence

Consider a set of vectors in \mathbb{R}^n ,

$$\left\{ v_1 = \begin{bmatrix} a_{11} \\ a_{21} \\ \vdots \\ a_{n1} \end{bmatrix}, v_2 = \begin{bmatrix} a_{12} \\ a_{22} \\ \vdots \\ a_{n2} \end{bmatrix}, \dots, v_m = \begin{bmatrix} a_{1m} \\ a_{2m} \\ \vdots \\ a_{nm} \end{bmatrix} \right\},$$

and use them as the columns of a matrix that we call A ,

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{21} & a_{22} & \cdots & a_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nm} \end{bmatrix}. \quad (6.1)$$

The following statements are equivalent:

- The set of vectors $\{v_1, v_2, \dots, v_m\}$ is linearly independent.
- The null space of A is the zero vector, that is, the only vector $x \in \mathbb{R}^m$ satisfying $Ax = 0_{n \times 1}$ is the vector $x = 0_{m \times 1}$

Determine if the set of vectors $\{v_1, v_2, v_3, v_4\}$ is linearly independent or not.

```
1 using Random
2 Random.seed! (2021)
3 v1=randn(5, 1)
4 v2=randn(5, 1)
5 v3=randn(5, 1)
6 v4=randn(5, 1)
7
8 # Place your code below
```

```

9 A = [v1 v2 v3 v4]
10 println ("Note how Julia reports that there are no non-zero vectors satisfying Ax = 0")
11 V = nullspace(A)

```

Output

Note how Julia reports that there are no non-zero vectors satisfying $Ax = 0$

```
4×0 Matrix{Float64}
```

Julia reported an “empty basis”... there are no non-zero vectors required to generate the null space of A .

Determine if the set of vectors $\{v_1, v_2, \dots, v_6\}$ is linearly independent or not. If they are linearly dependent, find a set of not all identically zero coefficients such that $\alpha_1 v_1 + \alpha_2 v_2 + \dots + \alpha_6 v_6 = 0$. In other words, find a non-trivial linear combination of the set of vectors that adds up to the zero vector.

```

1 using Random
2 Random.seed!(2021)
3 v1=randn(4, 1)
4 v2=randn(4, 1)
5 v3=randn(4, 1)
6 v4=randn(4, 1)
7 v5=randn(4, 1)
8 v6=randn(4, 1)
9
10 # Place your code below
11 # Compute alpha = [alpha1; alpha2; ...; alpha6]
12 # such that alpha1*v1 + alpha2*v2 + ... + alpha6*v6 = 0
13 # your code here
14 A = [v1 v2 v3 v4 v5 v6]
15 V = nullspace(A)

```

Output

```
6×2 Matrix{Float64}:
-0.884195  0.175427
-0.0530824 0.686466
-0.254579  0.093228
-0.274888 -0.461607
 0.158941 -0.150481
 0.223037  0.50356
```

```

1 # solution continued
2 alpha = sqrt(3)*V[:, 1] + pi*V[:, 2]

```

Output

```
6-element Vector{Float64}:
-0.9803501037910766
 2.064655155074321
-0.14805907841326366
-1.926298965466871
-0.19745609676639247
 1.9682934386167301
```

```
1 A*alpha
```

Output

```
4-element Vector{Float64}:
 4.996003610813204e-16
 0.0
-8.881784197001252e-16
 0.0
```

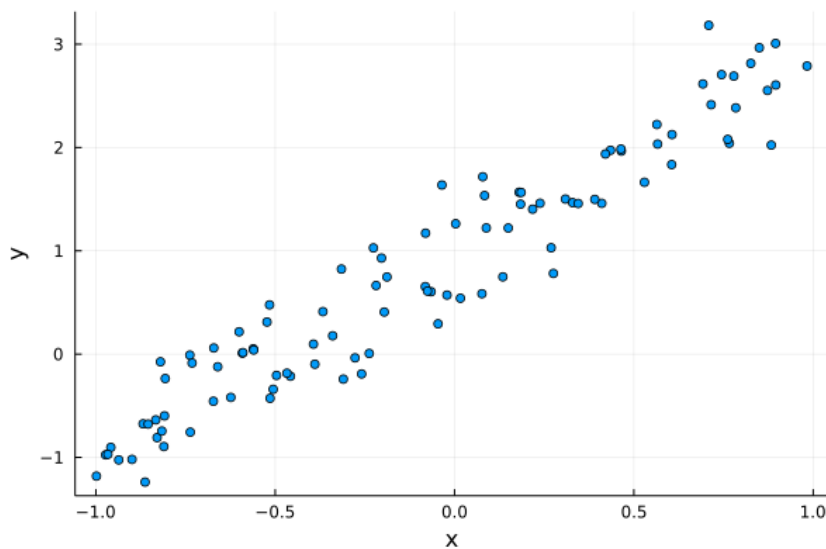
6.2 Linear Regression or Least Squares Fit to Data

We seek to fit a function to data. For example, we have a scatter plot of data points and wish to approximate the points with a straight line or perhaps a polynomial. It is assumed that you have scanned Chapter 8 in our textbook (not the Lab Manual).

The first step is to plot our data and decide if we are fitting a line or something more complicated. As we will see, both can be accommodated via linear regression.

```
1 # Generate some points for line fitting
2 # Run me, don't change me.
3 using Random
4 Random.seed!(2021)
5 NumPts = 100
6 m = 2
7 b = 1
8 x = -1 .+ 2*rand(NumPts, 1)
9 y = m*x .+ b + 0.30*randn(NumPts, 1)
10
11 using Plots
12 scatter(x, y, legend=false)
```

Output



This looks appropriate for fitting a line. Hence, we assume a linear model $y = mx + b$. We set up the linear equations

$$y_i = b + mx_i = \begin{bmatrix} 1 & x_i \end{bmatrix} \begin{bmatrix} b \\ m \end{bmatrix}, \quad 1 \leq i \leq N,$$

where N is the number of data points (100 in our case), and write it out in matrix form

$$\underbrace{\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{bmatrix}}_Y = \underbrace{\begin{bmatrix} 1 & x_1 \\ 1 & x_2 \\ \vdots & \vdots \\ 1 & x_N \end{bmatrix}}_\Phi \cdot \underbrace{\begin{bmatrix} b \\ m \end{bmatrix}}_\alpha, \quad (6.2)$$

where Y is the vector of y -data, Φ is called the **regressor matrix** and α is the vector of **unknown coefficients** that parameterize the model.

```
1 # Create the Y and the regressor matrix Phi
2 # Run me, don't change me.
3 Y = y
4 Phi = [ones(NumPts, 1) x]
```

Abridged Output

```
100x2 Matrix{Float64}:
 1.0 -0.188407
 1.0 -0.868452
 1.0 -0.203676
 1.0 -0.672369
 1.0  0.566187
 1.0 -0.731769
 .
 .
 .
 1.0  0.39111
 1.0 -0.0753372
 1.0  0.149722
 1.0 -0.816126
 1.0  0.420627
```

Least Squares Fit to Data also called Linear Regression

From our textbook, if the columns of Φ are linearly independent, or equivalently, $\Phi^T \Phi$ is invertible, then the following are equivalent

$$\alpha^* = (\Phi^T \Phi)^{-1} \Phi^T Y \iff \alpha^* = \arg \min_{\alpha} \|Y - \Phi \alpha\|^2 \iff (\Phi^T \Phi) \alpha^* = \Phi^T Y. \quad (6.3)$$

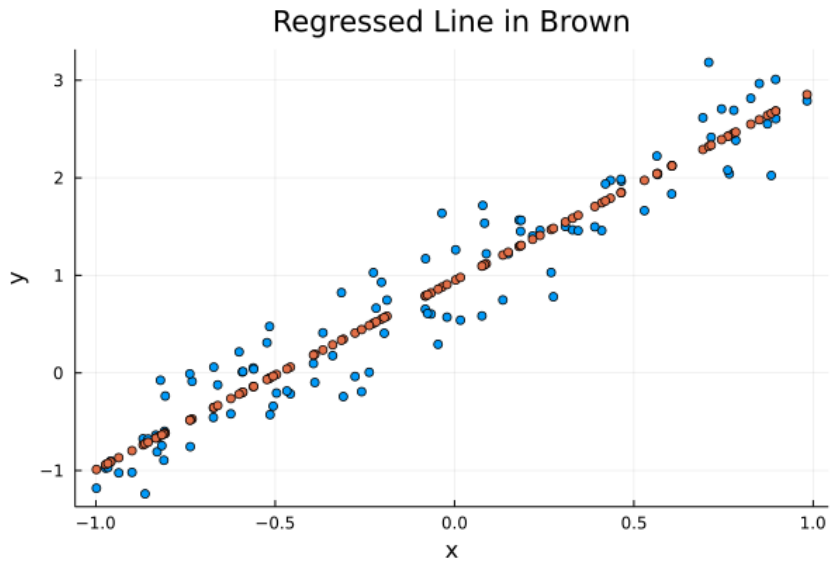
```
1 # OK for small problems, such as here, with two unknowns
2 A=Phi' *Phi
3 b=Phi' *Y
4 alpha = inv(A) *b
```

Output

```
2x1 Matrix{Float64}:
 0.9472834926740609
 1.9393293464390031
```

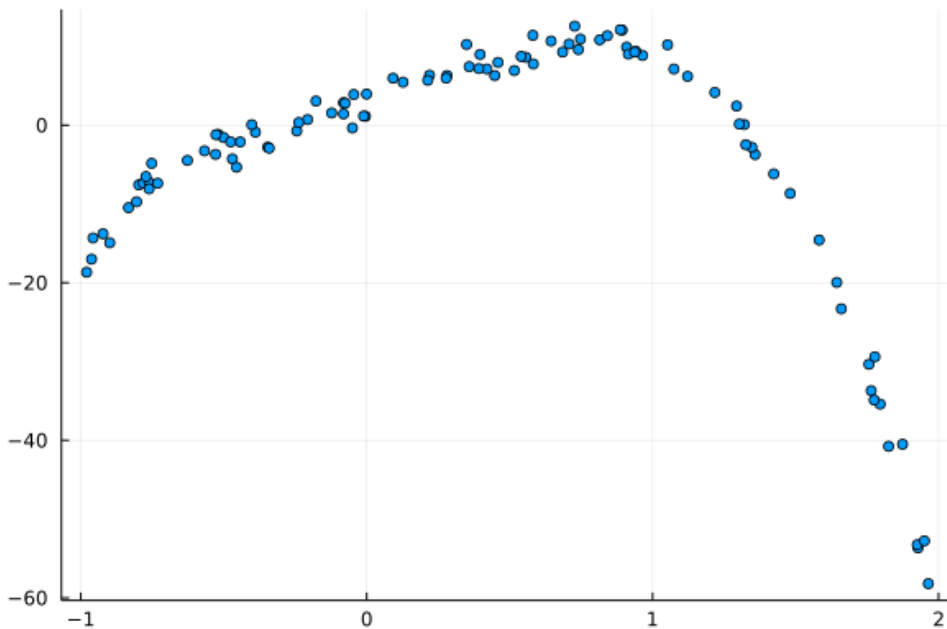
```
1 scatter(x, y, legend=false)
2 y_hat = Phi * alpha
3 scatter!(x, y_hat)
4 titre="Regressed Line in Brown"
5 plot!(xlabel="x", ylabel="y", title=titre)
```

Output



```
1 # Generate some data for polynomial curve fitting
2 # Run me, don't change me.
3 #
4 using Random
5 Random.seed!(123456789)
6 NumPts = 100
7 x = -1 .+ 3*rand(NumPts, 1)
8 a0 = 3
9 a1 = 6
10 a2 = 4
11 a3 = 6
12 a4 = -11
13 a5 = 2
14 a6 = -.2
15 # note the sin(2x) term
16 y = a6*x.^6 + a5*x.^5 + a4*x.^4 + a3*x.^3 +
17     a2*x.^2 .+ a1*sin.(2*x) .+ a0 + 1.50*randn(NumPts, 1)
18
19 using Plots
20 scatter(x, y, legend=false)
21 #png("PolyScatterPlot")
```

Output



Large Scale Least Squares via the LU Factorization

From our textbook, if the columns of Φ are linearly independent, or equivalently, $\Phi^T \Phi$ is invertible, we know the following are equivalent

$$\alpha^* = (\Phi^T \Phi)^{-1} \Phi^T Y \iff \alpha^* = \arg \min_{\alpha} \|Y - \Phi \alpha\|^2 \iff (\Phi^T \Phi) \alpha^* = \Phi^T Y. \quad (6.4)$$

The suggested “pipeline” for computing a least squared error solution to $(\Phi^T \Phi) \alpha^* = \Phi^T Y$ is

- factor $P \cdot (\Phi^T \Phi) =: L \cdot U$, that is, do the LU Factorization of $\Phi^T \cdot \Phi$,
- compute $\bar{b} := P \cdot \Phi^T Y$, and then
- solve $Ly = \bar{b}$ via forward substitution, and
- solve $U\alpha^* = y$ via backward substitution.

```

1 # Run me, don't change me. I am providing some nice functions for you.
2 # Forward and Back substitution functions from HW04
3
4 # This is a back substitution function. It solves for x in an equation Ux = b,
5 # where U is upper triangular. The function assumes U and b are the correct
6 # sizes. You can add error checking, if you wish.
7 function backwardsub(U, b)
8     # Check if U is non-singular
9     min_diagU = minimum(abs.(diag(U)))
10    if min_diagU < 1E-10
11        return false
12    end
13    n = length(b)
14    x = Vector{Float64}(undef, n)
15    x[n] = b[n]/U[n,n]
16    for i in n-1:-1:1
17        x[i] = (b[i] - (U[i, (i+1):n])' * x[(i+1):n]) ./ U[i,i]
18    end
19    return x

```

```

20 end
21 # This is a forward substitution function. It solves for x in an equation Lx = b,
22 # where L is lower triangular. # The function assumes L and b are the correct
23 # sizes. You can add error checking, if you wish.
24 function forwardsub(L, b)
25     # Check if L is non-singular
26     min_diagL = minimum(abs.(diag(L)))
27     if min_diagL < 1E-10
28         return false
29     end
30     n = length(b)
31     x = Vector{Float64}(undef, n);
32     x[1] = b[1]/L[1,1]
33     for i = 2:n
34         x[i] = (b[i] - (L[i,1:i-1])' * x[1:i-1]) ./ L[i,i]
35     end
36     return x
37 end

```

Output

forwardsub (generic function with 1 method)

```

1 # Create the Y and the regressor matrix Phi
2 # Run me, don't change me.
3 Y = y
4 Phi = [ones(NumPts,1) x x.^2 x.^3 x.^4]

```

Abridged Output

```

100×5 Matrix{Float64}:
 1.0  0.0919935  0.0084628  0.000778522  7.16189e-5
 1.0  0.90899   0.826264   0.751066   0.682712
 1.0  1.12227   1.25948   1.41347   1.58629
 1.0  1.58257   2.50452   3.96357   6.27261
 1.0  0.74038   0.548163   0.405849   0.300483
 1.0 -0.797207   0.63554   -0.506657   0.403911
 1.0 -0.206817   0.0427734 -0.00884629 0.00182957
 1.0 -0.177557   0.0315266 -0.00559777 0.000993924
 1.0  0.815018   0.664255   0.54138    0.441234
.
.
.
 1.0 -0.402096   0.161681   -0.0650113 0.0261408
 1.0  0.396441   0.157166   0.0623069 0.024701
 1.0 -0.0500005  0.00250005 -0.000125003 6.25023e-6
 1.0  0.685263   0.469585   0.321789   0.22051
 1.0 -0.898753   0.807756   -0.725973   0.65247
 1.0 -0.805068   0.648134   -0.521792   0.420078

```

```

1 # Find alpha by solving Phi' * Y = Phi' * Phi*alpha (Ax = b) using LU decomposition
2 using LinearAlgebra
3 F = lu(Phi' * Phi)
4 L = F.L
5 U = F.U
6 P = F.P
7 #

```

```

8 # Phi' * Y = Phi' * Phi * alpha
9 # after LU Factorization of Phi'*Phi, we have
10 # P * Phi' * Y = L * U * alpha
11 #
12 y_alpha = forwardsub(L, P*Phi'*Y)
13 alpha = backwardsub(U, y_alpha)

```

Output

```

5-element Vector{Float64}:
 3.71155035678547
11.770104292873295
-2.1786034505443537
 0.488282866753122
-5.371244500774466

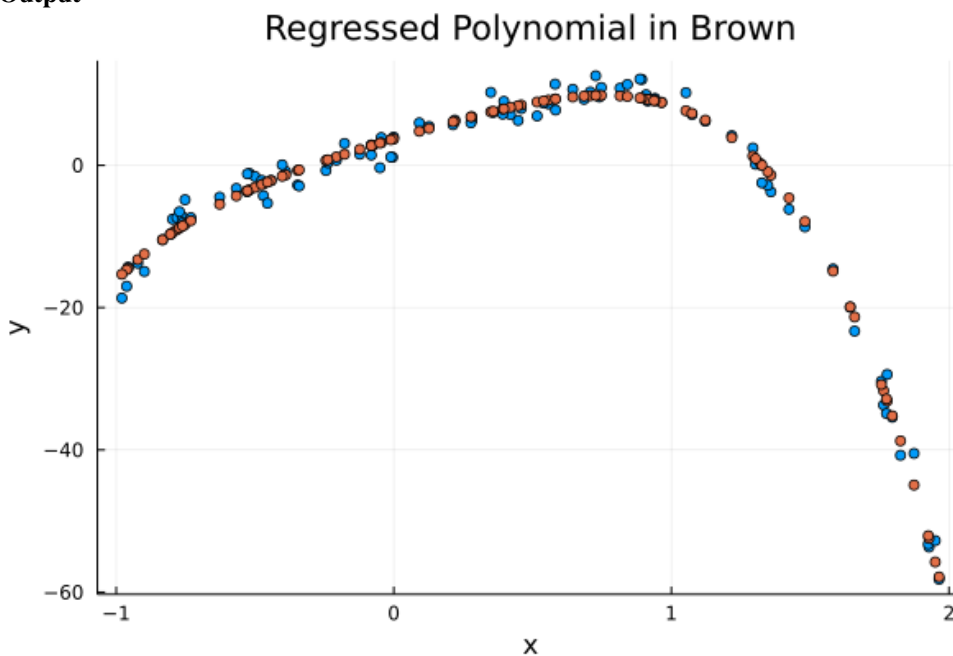
```

```

1 scatter(x, y, legend=false)
2 y_hat = Phi * alpha
3 scatter!(x, y_hat)
4 titre="Regressed Polynomial in Brown"
5 plot!(xlabel = "x", ylabel = "y", title=titre)

```

Output



6.3 Debugging

Let's build a function `LeastSquares_Solve(Phi, Y)` that takes in a regressor matrix and a set of measurements, and returns the optimal parameters and data related to the quality of the fit. Because least squares fitting relies on solving $Ax = b$ when A is square and invertible, we'll begin by building a function to compute exact solutions to square systems of linear equations. Moreover, we will build the function step-by-step and check for bugs as we go.

We start with baby steps. We only test for A being square.

```

1 # Function for solving Ax=b using LU decomposition
2 function Exact_Solve(A, b)
3     nRows, nCols = size(A)

```

```

4 # We'll check for A not square. We assume A is a matrix.
5 if ( nRows != nCols)
6     println("System of equations not properly formed")
7     return x = NaN
8 end
9 xDummy= [1;2.0]
10 return xDummy
11 end
12
13 A=[1 2; 3 4; 5 6]
14 b=[8; 9.0]
15 xAns = Exact_Solve (A, b)

```

Output

```

System of equations not properly formed
NaN

```

Next we add in a check that b has the same number of rows as A . The command `||` “double-pipes” serves as the logical `or` command. We test it on good data and bad data.

```

1 # Function for solving Ax=b using LU decomposition
2 function Exact_Solve (A, b)
3     nRows, nCols = size (A)
4     # We'll check for A not square. We assume A is a matrix.
5     if ( nRows != nCols) || (length (b) != nRows)
6         println("System of equations not properly formed")
7         return x = NaN
8     end
9     xDummy= [1;2.0]
10    return xDummy
11 end
12
13 A=[1 2; 3 4]
14 b=[8; 9.0; 11]
15 @show xAns = Exact_Solve (A, b)
16 b=[11, 13]
17 @show xAns = Exact_Solve (A, b);

```

Output

```

System of equations not properly formed
xAns = Exact_Solve(A, b) = NaN
xAns = Exact_Solve(A, b) = [1.0, 2.0]

```

Next we add in a solver via our beloved LU pipeline. We also check that the result is a valid solution of the equations.

```

1 # Function for solving Ax=b using LU decomposition
2 function Exact_Solve (A, b)
3     nRows, nCols = size (A)
4     # We'll check for A not square. We assume A is a matrix.
5     if ( nRows != nCols) || (length (b) != nRows)
6         println("System of equations not properly formed")
7         return x = NaN
8     end
9     F=lu (A)
10    y=forwardsub (F.L, F.P*b)
11    x=backwardsub (F.U, y)
12    return x

```

```

13 end
14
15 A=[1 2; 3 4]
16 b=[8; 9.0; 11]
17 @show xAns = Exact_Solve(A, b)
18 b=[11, 13]
19 @show xAns = Exact_Solve(A, b);
20 norm(A*xAns-b)

```

Output

```

System of equations not properly formed
xAns = Exact_Solve(A, b) = NaN
xAns = Exact_Solve(A, b) = [-9.0, 10.0]

```

0.0

We now add a tolerance and check for A being singular. If A is singular, we return $x=NaN$; otherwise, we implement the LU pipeline. We are very proud of our function!

```

1 # Function for solving Ax=b using LU decomposition
2 function Exact_Solve(A, b, aTol=1e-10)
3     nRows, nCols = size(A)
4     if (length(b) != nRows) || (nRows != nCols)
5         println("System of equations not properly formed")
6         return x=NaN
7     end
8     F=lu(A)
9     if minimum(abs.(diag(F.U))) < aTol
10        println("diagU has numbers that are nearly zero")
11        return x=NaN
12    else
13        y=forwardsub(F.L, F.P*b)
14        x=backwardsub(F.U, y)
15        return x
16    end
17 end
18
19 A=[1 1; 0 1e-14]
20 b=[8; 9.0]
21 @show xAns = Exact_Solve(A, b)
22 #
23 A=[1 1; 3 4]
24 b=[11, 13]
25 @show xAns = Exact_Solve(A, b)
26 norm(A*xAns-b)

```

Output

```

diagU has numbers that are nearly zero
xAns = Exact_Solve(A, b) = NaN
xAns = Exact_Solve(A, b) = [31.000000000000001, -20.000000000000007]

```

3.552713678800501e-15

Now we do a real life-size check!

```

1 using Random
2 A = randn(1000,1000)

```

```

3 b=randn(1000,1)
4 xAns = Exact_Solve(A, b);
5 norm(A*xAns-b)

```

Output

2.643871993979311e-11

And it looks great! So next, we develop a least squares solver. We use the fact that `alphaStar` satisfies

$$\Phi^T Y = \Phi^T \cdot \Phi \alpha^* \iff \Phi^T \cdot \Phi \alpha^* = \Phi^T Y$$

and thus `alphaStar` should be the exact solution to $Ax = b$, for

$$A \iff \Phi^T \cdot \Phi, b \iff \Phi^T Y, x \iff \alpha^*.$$

```

1 # Function for fitting data with Least Squares
2 function LeastSquares_Solve(Phi, Y)
3     alphaStar = Exact_Solve(Phi'*Phi, Phi'*Y) # Phi'*Y = Phi'*Phi * alpha
4     # Return some extra quantities
5     Yhat = Phi*alphaStar
6     Error = Y-Yhat
7     SqError = Error' * Error
8     RootMeanSqError = sqrt(SqError/length(Y))
9     return alphaStar, Yhat, RootMeanSqError
10 end

```

Output

LeastSquares_Solve (generic function with 1 method)

Now we run a real test!

```

1 # Generate some data for polynomial curve fitting
2 # Run me, don't change me.
3 #
4 using Random
5 Random.seed!(123456789)
6 NumPts = 100
7 x = -1 .+ 2*rand(NumPts,1)
8 a0 = 3
9 a1 = 6
10 a2 = 4
11 a3 = 5
12 a4 = -5
13 y = a4*x.^4 + a3*x.^3 + a2*x.^2 .+ a1*x.^1 .+ a0 + 0.50*randn(NumPts, 1)
14
15 using Plots
16 scatter(x, y, legend=false, label="Y")
17
18 # Create the Y and the regressor matrix Phi
19 Y = y
20 Phi = [ones(NumPts,1) x x.^2 x.^3] # Fitting a cubic to a quartic
21
22 alphaStar, Yhat, RootMeanSqError = LeastSquares_Solve(Phi, Y)
23
24 println("The mean squared error is ", RootMeanSqError)
25
26 p1 = scatter!(x, Yhat, label="Y_hat")

```

```
27 png (p1, "LinearRegressionFit")
28 display (p1)
```

Output

The mean squared error is [0.6185570543212526]

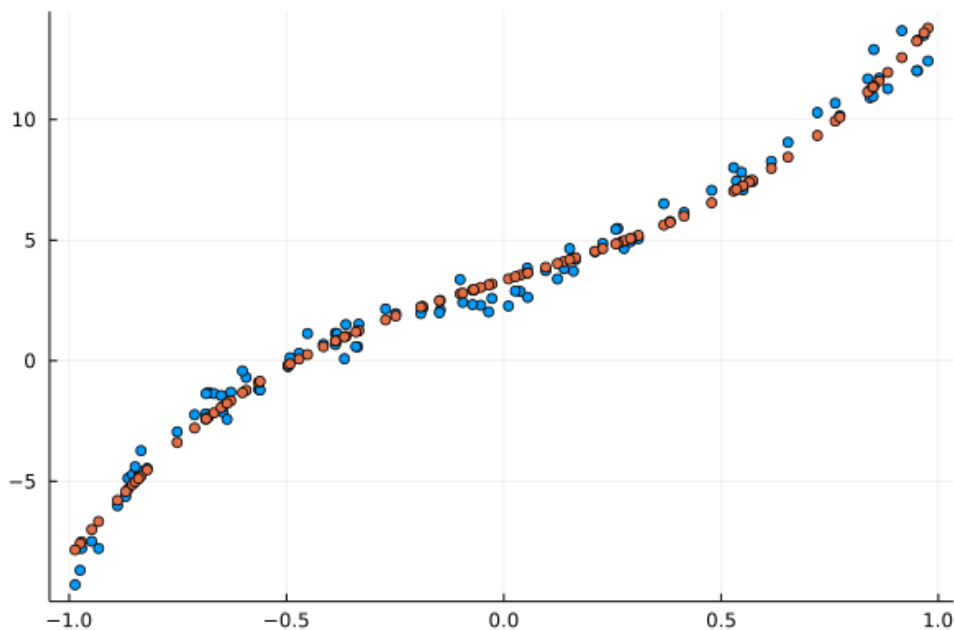


Figure 6.1: The blue dots are data while the orange dots are the fit to the data.

Secret to Building Functions is to Check Them While Building Them

Remarks:

- We broke our goal into two pieces, one solving an exact system of equations and one solving a least squares problem.
- We included error checking as we went along and tested each piece.
- By the time we got to the end, the function that was new to us, one that solves a least squares problem, was a trivial extension of something familiar to us, solving a square system of equations with an invertible matrix.
- **You can do this too, even in HW problems:**
 - Even when we say to place your code HERE, that code can include calls to other functions that you place in the same cell.
 - **Really? Yes. Build a function called `LeastSquares_Solve(Phi, Y)` that returns `alphaStar`**

```
1 # Function for fitting data with Least Squares
2 function LeastSquares_Solve (Phi, Y)
3     ### BEGIN SOLUTION
4
5     ### END SOLUTION
6     return alphaStar
7 end
```

Output

LeastSquares_Solve (generic function with 1 method)

```
1 # Function for solving Ax=b using LU decomposition
2 function Exact_Solve(A, b, aTol=1e-10)
3     nRows, nCols = size(A)
4     if (length(b) != nRows) || (nRows != nCols)
5         println("System of equations not properly formed")
6         return x=NaN
7     end
8     F=lu(A, check=false)
9     display(F)
10    #indicesDiagUsmall=findall(x->x<aTol, abs.(diag(F.U)))
11    if minimum(abs.(diag(F.U))) < aTol
12        println("diagU has numbers that are nearly zero")
13        return x=NaN
14    else
15        y=forwardsub(F.L, F.P*b)
16        x=backwardsub(F.U, y)
17        return x
18    end
19 end
20
21
22 # Function for fitting data with Least Squares
23 function LeastSquares_Solve(Phi, Y)
24     ### BEGIN SOLUTION
25     alphaStar = Exact_Solve(Phi'*Phi, Phi'*Y) # Phi'*Y = Phi'*Phi * alpha
26     ### END SOLUTION
27     return alphaStar
28 end
```

Output

LeastSquares_Solve (generic function with 1 method)

6.4 (Optional Read) Fitting a Surface to Data



We'll show that fitting a surface to data is the same as fitting a curve to data. Here, we'll continue using monomials as our basis functions. In Project 2, you will use radial basis functions, a better basis for surfaces with lots of curvature.

```
1 # run me, don't change me
2 #
3 println("Be patient. It takes time to load the Distributions package.")
4 using Distributions
5 using Plots
6 using Pkg
7 Pkg.add("PyPlot")
```

Output

Be patient. It takes time to load the Distributions package.

```
Info: Precompiling Plots [91a5bcdd-55d7-5caf-9e0b-520d859cae80]
@ Base loading.jl:1317
```

```
1 # Attempting to make the Wavefield!
2 z(x,y) = 0.2*sin(pi*x/2) - 0.2*cos(pi*y/3) + rand(Uniform(-0.03,0.03))
3
4 N = 100
5 Nsq = N^2
6 X = zeros(Nsq, 1)
7 Y = zeros(Nsq, 1)
8 Z = zeros(Nsq, 1)
9 for i = 1:N
10     for j = 1:N
11         X[(i-1)*N+j] = 10.0*i/N
12         Y[(i-1)*N+j] = 10.0*j/N
13         Z[(i-1)*N+j] = z(X[(i-1)*N+j], Y[(i-1)*N+j])
14     end
15 end
```

```

16 scatter (X, Y, Z, marker_z=Z, legend=false, color = :greens, colorbar=true, camera=(-60, 60))
17 #png("WaveFieldData")

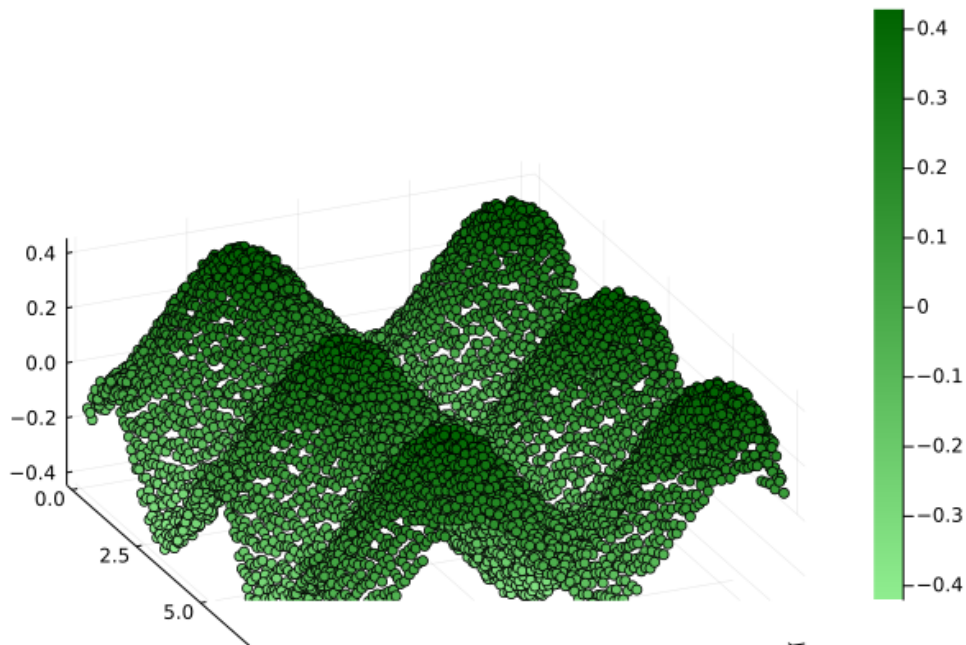
```

Output

```

Info: Precompiling GR_jll [d2c73de3-f751-5644-a686-071e5b155ba9]
@ Base loading.jl:1317
Warning: Module Cairo_jll with build ID 23135458039207652 is missing from the cache.
This may mean Cairo_jll [83423d85-b0ee-5818-9007-b63ccbeb887a] does not support
precompilation but is imported by a module that does.
@ Base loading.jl:1008
Info: Skipping precompilation since __precompile__(false).
Importing GR_jll [d2c73de3-f751-5644-a686-071e5b155ba9].
@ Base loading.jl:1025
Info: Precompiling Qt5Base_jll [ea2cea3b-5b76-57ae-a6ef-0a8af62496e1]
@ Base loading.jl:1317
Warning: Module Glib_jll with build ID 23135459152168067 is missing from the cache.
This may mean Glib_jll [7746bdde-850d-59dc-9ae8-88ece973131d] does not support
precompilation but is imported by a module that does.
@ Base loading.jl:1008
Info: Skipping precompilation since __precompile__(false).
Importing Qt5Base_jll [ea2cea3b-5b76-57ae-a6ef-0a8af62496e1].
@ Base loading.jl:1025
Warning: camera: -60° not in [0°, 90°]
@ Plots /opt/julia/packages/Plots/PomtQ/src/backends/gr.jl:1438

```



```

1 using LinearAlgebra
2
3 # Function for solving Ax=b using LU decomposition
4 function solveAxEqb (A, b, aTol=1e-10)
5     nRows, nCols = size(A)
6     if (length(b) != nRows) || (nRows != nCols)
7         println("System of equations not properly formed")
8         return x=undef
9     end

```

```

10  F=lu(A)
11  indicesDiagUsmall=findall(x->x<aTol, abs.(diag(F.U)))
12  if length(indicesDiagUsmall) > 0
13      println("diagU has numbers that are nearly zero")
14      return x=undef
15  else
16      y=forwardsub(F.L,F.P*b)
17      x=backwardsub(F.U, y)
18      return x
19  end
20 end
21
22 # Function for fitting data with Least Squares
23 function LeastSquares_Solve(Phi, Z)
24     alpha = solveAxEqb(Phi'*Phi, Phi'*Z) # Phi'*Z = Phi'*Phi * alpha
25     Zhat = Phi*alpha
26     Error = Z-Zhat
27     SqError = Error'*Error
28     MeanError = sqrt(SqError/length(Z) )
29     return Zhat, Error, SqError, MeanError
30 end

```

Output

```
LeastSquares_Solve (generic function with 1 method)
```

```

1 # Creating regressor matrix Phi as polynomial fitting
2 # We include from a constant term, linear terms, quadratic terms,
3 # and stop after adding cubic terms
4 Phi = [ones(Nsq,1) X Y X.^2 Y.^2 X.*Y X.^3 Y.^3 (X).*(Y.^2) (Y).*(X.^2)]
5 Zhat, Error, SqError, MeanError = LeastSquares_Solve(Phi, Z)
6 println("Squared error = $SqError and Mean Error = $MeanError")
7 println("The root mean squared error of our fit is $(MeanError*1e3) mm
8     or, if you prefer $(MeanError*1e2/2.54) inches")

```

Output

```

Squared error = [260.2866352193259] and Mean Error = [0.16133401229106215]
The root mean squared error of our fit is [161.33401229106215] mm
or, if you prefer [6.351732767364652] inches

```

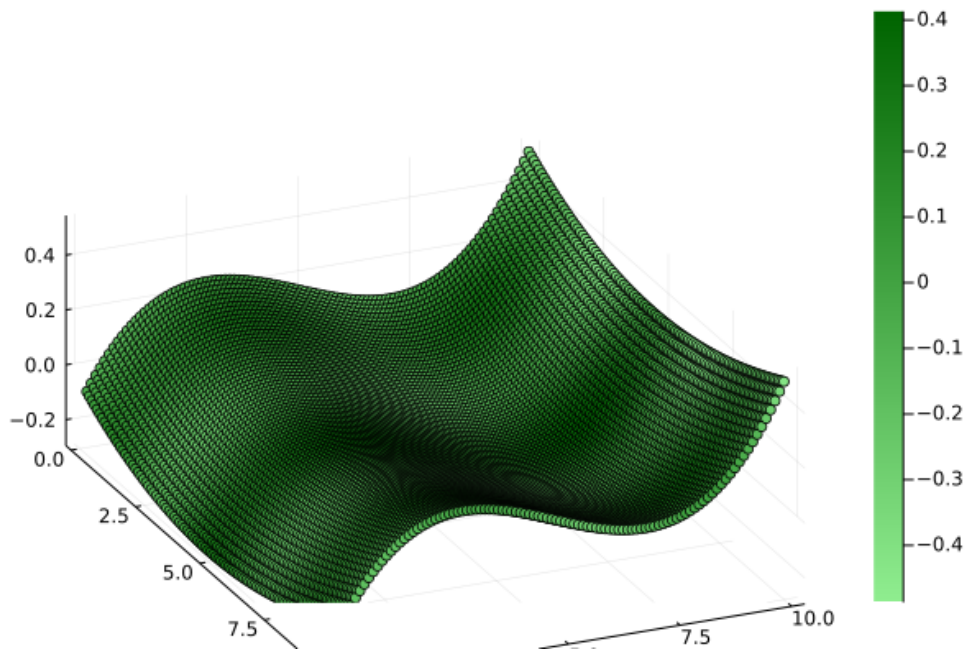
```

1 # Make a scatter plot of Zhat
2 println("Note how bad the fit is...it does not look like the wavefield at all.")
3 scatter(X, Y, Zhat, marker_z=Error, legend=false, color = :greens, colorbar=true, camera=(-60, 60))
4 #png("WaveFieldFit01")

```

Output

```
Note how bad the fit is...it does not look like the wavefield at all.
```



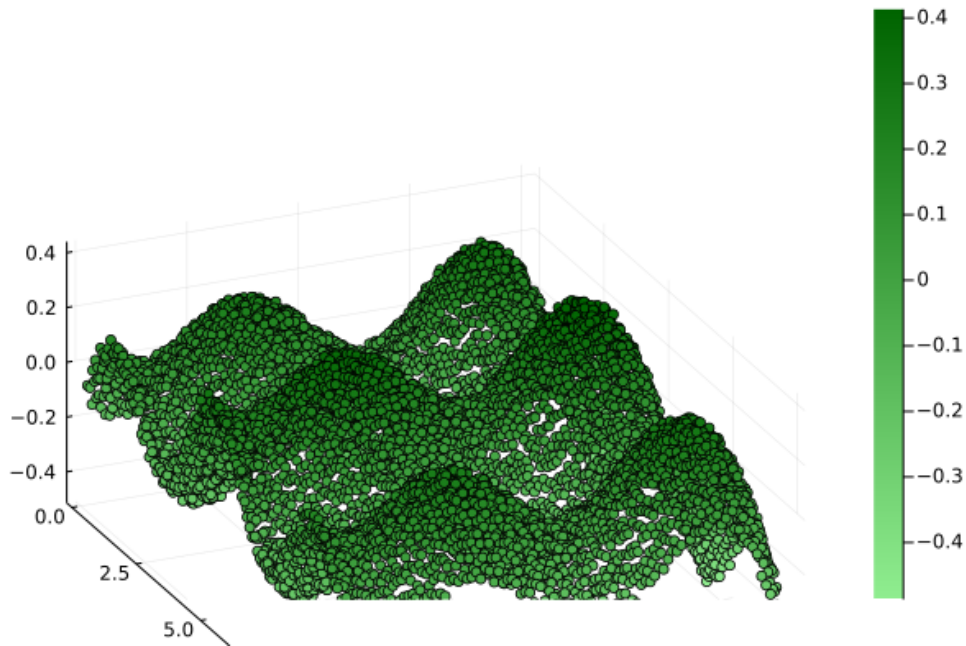
```

1 # Make a scatter plot of the error
2 println("The fit is so bad, the error looks like the wave field!!!")
3 scatter(X, Y, Error, marker_z=Error, legend=false, color = :greens, colorbar=true, camera=(-60, 60))
4 #png("WaveFieldFit01Error")

```

Output

The fit is so bad, the error looks like the wave field!!!



```

1 # We'll take another crack at it, using a method that makes it painless to include more
  terms
2

```

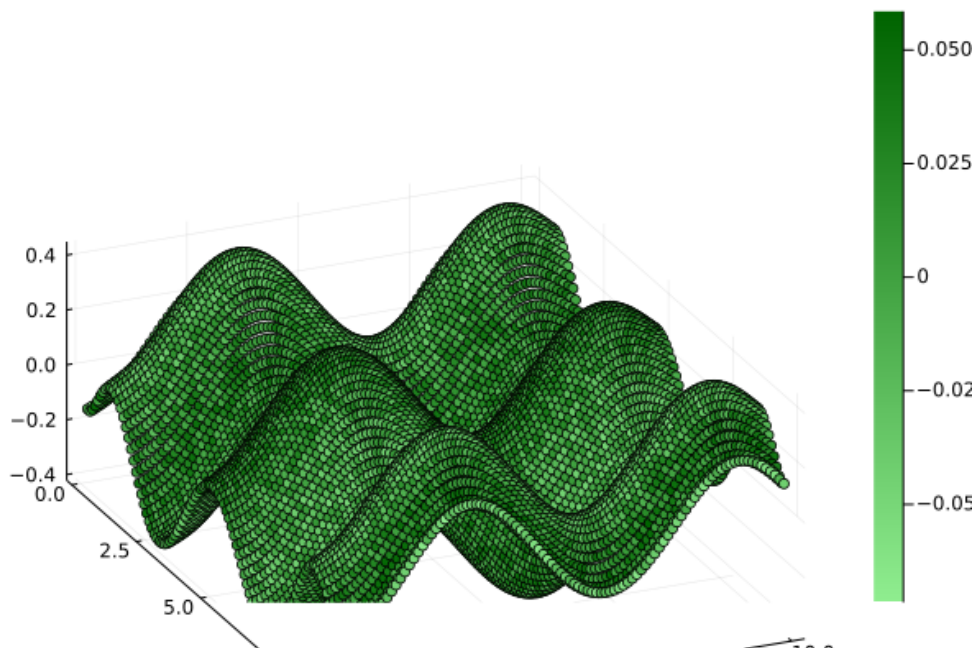
```

3 # Using for loop to build the regressor matrix Phi
4 m=8
5 Phi= Array{Float64,2} (undef, Nsq, 0)
6 avgX=mean(X)
7 avgY=mean(Y)
8 k=0
9 for i=0:m
10     for j=0:m-i
11         k=k+1
12         Phi=[Phi (X.-avgX).^i .* (Y.-avgY).^j / (factorial(i)*factorial(j))]
13     end
14 end
15 Zhat, Error, SqError, MeanError = LeastSquares_Solve(Phi, Z)
16 println("Squared error = $SqError and Mean Error = $MeanError")
17 println("Now see what you think of the fit!")
18 scatter(X, Y, Zhat, marker_z=Error, legend=false, color = :greens, colorbar=true, camera=(-60, 60))
19 #png("WaveFieldFit02")

```

Output

Squared error = [5.066187463106273] and Mean Error = [0.022508192870833218]
 Now see what you think of the fit!



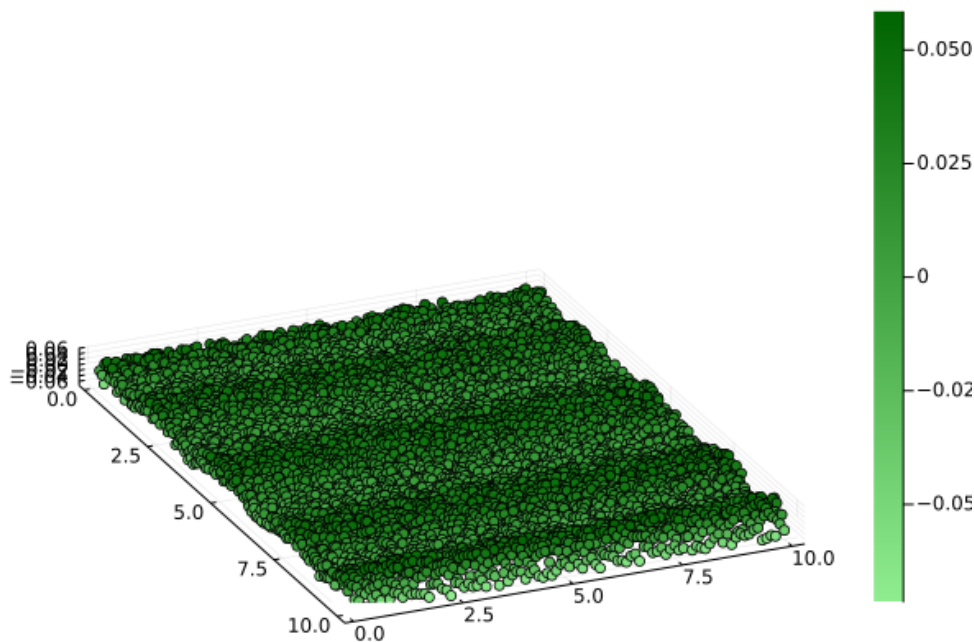
```

1 # Make a scatter plot of the error
2 println("Now, that's what I call a small error plot.")
3 scatter(X, Y, Error, marker_z=Error, legend=false, color = :greens, colorbar=true, camera=(-60, 60))
4 #png("WaveFieldFit02Error")

```

Output

Now, that's what I call a small error plot.



6.5 (Optional Read) Lab 6.5 A Potpourri of Interesting Commands

While the tools/commands covered here are not essential for ROB 101, they can make programming either more rewarding or more fun.

6.5.1 Using the Benchmark Tool: Which is Faster for Solving $Ax=b$?

```

1 using Pkg
2 Pkg.add("BenchmarkTools")
3
4 using LinearAlgebra
5
6 function backwardsub(U, b)
7     n = length(b)
8     x = Vector{Float64}(undef, n)
9     x[n] = b[n]/U[n,n]
10    for i in n-1:-1:1
11        x[i] = (b[i] - (U[i, (i+1):n])' * x[(i+1):n]) / U[i,i]
12    end
13    return x
14 end
15
16 function forwardsub(L, b)
17     n = length(b)
18     x = Vector{Float64}(undef, n)
19     x[1] = b[1]/L[1,1]
20     for i = 2:n
21         x[i] = (b[i] - (L[i, 1:i-1])' * x[1:i-1]) / L[i,i]
22     end
23     return x
24 end
25
26 function lu_solve(A, b)
27     F=lu(A)

```

```

28     y=forwardsub (F.L, b[F.p])
29     x=backwardsub (F.U, y)
30     return x
31 end
32
33 function inv_solve (A, b)
34     invA = inv (A)
35     x = invA*b
36     return x
37 end

```

Output

inv_solve (generic function with 1 method)

```

1 # We run this cell to compile the functions before using the Benchmark Tool
2 using Random
3 Random.seed! (6969420)
4 N=3
5 A = rand (N, N)
6 b = rand (N)
7 x_inv = inv_solve (A, b)
8 x_lu = lu_solve (A, b);

```

Output

Nothing.

Now, we check the relative speed of the two solvers, using the benchmark tool.

```

1 using BenchmarkTools
2 Random.seed! (6969420)
3 N=1000
4 A = rand (N, N)
5 b = rand (N)
6 @benchmark x_inv = inv_solve (A, b)

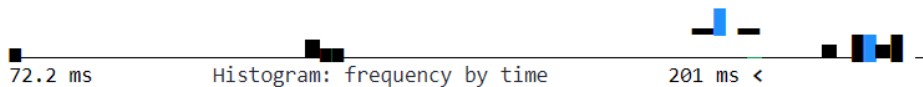
```

Output

```

BenchmarkTools.Trial: 29 samples with 1 evaluation.
Range (min ... max):  72.191 ms ... 201.125 ms  | GC (min ... max): 0.00% ... 0.00%
Time (median):        195.124 ms                | GC (median):    0.00%
Time (mean ± σ):     178.044 ms ± 36.620 ms     | GC (mean ± σ): 0.13% ± 0.31%

```



Memory estimate: 8.13 MiB, allocs estimate: 6.

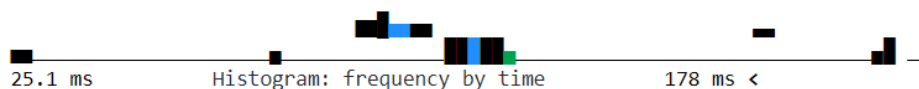
```

1 @benchmark x_lu = lu_solve (A, b)

```

Output

```
BenchmarkTools.Trial: 46 samples with 1 evaluation.
Range (min ... max): 25.054 ms ... 178.169 ms | GC (min ... max): 15.38% ... 41.82%
Time (median): 104.192 ms | GC (median): 2.11%
Time (mean ± σ): 110.632 ms ± 33.125 ms | GC (mean ± σ): 7.81% ± 15.08%
```



Memory estimate: 38.74 MiB, allocs estimate: 4007.

6.5.2 Printing with Style and Macros @

```
1 printstyled ("Haha\n", color=:light_cyan)
2 printstyled ("I love ROB 101\n", color=:red, bold=true)
```

Output Haha
I love ROB 101

```
1 ? printstyled
```

Output

search: printstyled

```
printstyled([io], xs...; bold::Bool=false, color::Union{Symbol,Int}=:normal)
Print xs in a color specified as a symbol or integer, optionally in bold.
```

color may take any of the values :normal, :default, :bold, :black, :blink, :blue, :cyan, :green, :hidden, :light_black, :light_blue, :light_cyan, :light_green, :light_magenta, :light_red, :light_yellow, :magenta, :nothing, :red, :reverse, :underline, :white, or :yellow or an integer between 0 and 255 inclusive. Note that not all terminals support 256 colors. If the keyword bold is given as true, the result will be printed in bold.

```
1 macro add_69(num)
2     return num+69
3 end
```

Output

@add_69 (macro with 1 method)

```
1 @add_69 420
```

Output

489

```
1 macro add_three_of_them(a, b, c)
2     return a+b+c
3 end
```

Output

@add_three_of_them (macro with 1 method)


```
1 @add_three_of_them 1 2 3
```

Output

```
6
```

```
1 macro print_in_yellow(x)
2     printstyled(x, color=:yellow)
3 end
```

Output

```
@print_in_yellow (macro with 1 method)
```

```
1 @print_in_yellow "My name is... My name is... My name is... SLIM SHADY"
```

Output

```
My name is... My name is... My name is... SLIM SHADY
```

Julia has an amazing list of Emojis that you can copy and paste into your cells. The symbols will not print in the latex code box below, but they will print in Julia: <https://emojidb.org/julia-and-julia-emojis>

```
1 macro aggressive_print(x)
2     s = uppercase(x)
3     s*="!!!!" # s*= appends the string between the quote marks
4             # The characters do not show here but do work in Julia; see below.
5     printstyled(s, color=:red)
6 end
```

Output

```
@aggressive_print (macro with 1 method)
```

```
1 @aggressive_print "hi nice to meet you"
```

Output

```
1 macro aggressive_print(x)
2     s = uppercase(x)
3     s*="!!!!🙄🙄🙄🙄"
4     printstyled(s, color=:red)
5 end
```

```
@aggressive_print (macro with 1 method)
```

```
1 @aggressive_print "hi nice to meet you"
```

```
HI NICE TO MEET YOU!!!!🙄🙄🙄🙄
```

6.5.3 Multiple Dispatch and Function Overloading

```
1 function add_them(a::Int, b::Int)
2     result = a+b
3     return result
4 end
5
```

```

6 function add_them (a :: Int, b :: Vector)
7     result = a .+ b
8     return result
9 end
10
11 function add_them (a :: String, b :: Int)
12     result = a*string(b)
13     return result
14 end
15
16 function add_them (a :: String, b :: Vector)
17     result = a
18     for s in b result*=string(s) end
19     return result
20 end

```

Output

add_them (generic function with 4 methods)

```

1 @show add_them(2, 3);
2 @show add_them(2, [3, 4, 5]);
3 @show add_them("mrbeast", 6000);
4 @show add_them("gluckgluck", [9, 0, 0, 0]);

```

Output

```

add_them(2, 3) = 5
add_them(2, [3, 4, 5]) = [5, 6, 7]
add_them("mrbeast", 6000) = "mrbeast6000"
add_them("gluckgluck", [9, 0, 0, 0]) = "gluckgluck9000"

```

```

1 import Base: +
2
3 function +(a :: Int, b :: Vector)
4     result = a .+ b
5     return result
6 end
7
8 function +(a :: String, b :: Int)
9     result = a*string(b)
10    return result
11 end
12
13 function +(a :: String, b :: Vector)
14    result = a
15    for s in b result*=string(s) end
16    return result
17 end

```

Output

+ (generic function with 193 methods)

```

1 @show 2 + [3, 4, 5];
2 @show "mrbeast" + 6000;
3 @show "gluckgluck" + [9, 0, 0, 0];

```

Output

`2 + [3, 4, 5] = [5, 6, 7]`

`"mrbeast" + 6000 = "mrbeast6000"`

`"gluckgluck" + [9, 0, 0, 0] = "gluckgluck9000"`

Chapter 7

Julia Lab 7: Gram-Schmidt Algorithm, Orthogonal (Basis) Vectors, and Computing the Null Space

Learning Objectives

- Master the Gram-Schmidt algorithm
- Explore some of its uses
- Regression
- Debugging, or finding and fixing errors in your code

Outcomes

- In a set of orthogonal vectors, each vector forms a right angle with respect to the other vectors in the set
- If a vector is non-zero, we can easily normalize its length to one
- In a set of orthonormal vectors, each vector has length one and forms a right angle with respect to the other vectors in the set
- Gram-Schmidt applied to a set of linearly independent vectors produces a set of orthogonal vectors, and with one small modification, we can produce a set of orthonormal vectors
- Gram-Schmidt provides an easy way to compute the null space of a matrix

Either download Lab7 from our Canvas site or open up a Jupyter notebook so that you can enter code as we go. It is suggested that you have line numbering toggled on.

This lab goes all in on the Gram-Schmidt Process. When you are done with the lab, you should feel that you “own” the algorithm.

Julia has a command for computing the inner product, aka dot product, of two vectors. Surprise, the command is called `dot`. We’ll show you how to use it.

```
1 # The dot comamnd is part of the LinearAlgebra package.
2 #
3 using LinearAlgebra
4 #
5 u = [1, 2, 3, 4, 5]
6 v = [6, 7, 8, 9, 10]
7
8 uDOTv = dot(u, v)
```

Output

130

You can easily compute the dot product without the LinearAlgebra Package.

```
1 # The dot command computes u'*v; here is the proof!
2 u'*v
```

Output

130

Normalizing the length of vectors to one is a pain by hand. But in Julia, it’s such a snap that we will soon incorporate it directly into the Gram-Schmidt process.

```
1 # The norm command is part of the LinearAlgebra Package, which we already ran
2 @show norm(v)
3 w = v / norm(v)
4 @show norm(w)
5 w
```

Output

```
norm(v) = 18.16590212458495
norm(w) = 1.0
```

```
5-element Vector{Float64}:
 0.3302891295379082
 0.3853373177942262
 0.4403855060505443
 0.4954336943068623
 0.5504818825631803
```

7.1 Orthogonal Vectors

Suppose that $\{u_1, u_2\}$ is a set of linearly independent vectors. The Gram-Schmidt Process builds from them an orthogonal set of vectors that spans the same set of vectors as $\{u_1, u_2\}$. It works as follows:

Step 1: $v_1 := u_1$

Step 2: $v_2 := u_2 - a_2v_1$, where we seek to choose a_2 such that $v_2 \bullet v_1 = 0$. We compute

$$\begin{aligned} v_2 \bullet v_1 &= (u_2 - a_2v_1) \bullet v_1 \\ &= u_2 \bullet v_1 - a_2v_1 \bullet v_1. \end{aligned}$$

If $v_1 \bullet v_1 \neq 0$, then we can set $u_2 \bullet v_1 - a_2 v_1 \bullet v_1 = 0$ and solve for a_2 , namely

$$a_2 = \frac{u_2 \bullet v_1}{v_1 \bullet v_1}.$$

Important Formula to Build $v_1 \perp v_2$ from u_1 and u_2 while Preserving Spans

$$v_1 = u_1$$

$$v_2 = u_2 - \left(\frac{u_2 \bullet v_1}{v_1 \bullet v_1} \right) v_1$$

(7.1)

$$\begin{aligned} \text{span}\{v_1\} &= \text{span}\{u_1\} \\ \text{span}\{v_1, v_2\} &= \text{span}\{u_1, u_2\} \end{aligned}$$

We illustrate the above step by step.

```

1 using Random
2 Random.seed! (2022)
3 u1 = rand (4, 1)
4 u2 = rand (4, 1)
5 U = [u1 u2]
6 @show det (U' * U) # non-zero means u1 and u2 are independent
7 # G-S on two vectors (see grey box above)
8 @show v1 = u1
9 @show v2 = u2 - ( dot (u2, v1) / dot (v1, v1) ) * v1
10 V = [v1 v2]
11 @show dot (v1, v2) # should be zero
12 println (" ") # Will print a space
13 @show det (V' * V) # non-zero means v1 and v2 are independent
14 V

```

Output

```

det (U' * U) = 0.32941782975491707
v1 = u1 = [0.7982589547203172; 0.8998485508054206; 0.15010598805721265;
0.1749482291443072]
v2 = u2 - (dot(u2, v1) / dot(v1, v1)) * v1 = [-0.04289157616244632; -0.07773177980842272;
0.3598021757771466; 0.28681029417179305]
dot (v1, v2) = -1.2642012348321895e-17

det (V' * V) = 0.32941782975491707

4×2 Matrix{Float64}:
 0.798259  -0.0428916
 0.899849  -0.0777318
 0.150106   0.359802
 0.174948   0.28681

```

```

1 v1=v1/norm (v1)
2 v2=v2/norm (v2)
3 #
4 V= [v1 v2] # now columns are orthonormal vectors
5 @show det (V' * V) # non-zero implies linearly independent
6 @show V' * V - [v1' * v1 v1' * v2; v2' * v1 v2' * v2]

```

```
7 V' * V
```

Output

```
det(V' * V) = 1.00000000000000004  
V' * V - [v1' * v1  v1' * v2; v2' * v1  v2' * v2] = [0.0 0.0; 0.0 0.0]
```

```
2x2 Matrix{Float64}:  
 1.0      -1.34268e-17  
-1.34268e-17  1.0
```

```
1 # A helper function to zero out small entries of a matrix or vector  
2 function cleanUp(A, tol=1e-10)  
3     B=copy(A)  
4     indicesSmall=findall(x->x<tol, abs.(B))  
5     B[indicesSmall]=0.0*abs.(B[indicesSmall])  
6     return B  
7 end
```

Output

```
cleanUp (generic function with 2 methods)
```

```
1 cleanUp(V' * V)
```

Output

```
2x2 Matrix{Float64}:  
 1.0  0.0  
 0.0  1.0
```

7.2 The Span Condition

Next will check that $\text{span}\{u_1, u_2\} = \text{span}\{v_1, v_2\}$, where we recall that computing the span of a set of vectors means determining “all possible linear combinations”, which seems like a lot of work! However, if we show that

$$u_1 \in \text{span}\{v_1, v_2\} \tag{7.2}$$

$$u_2 \in \text{span}\{v_1, v_2\}, \tag{7.3}$$

it then follows that $\text{span}\{u_1, u_2\} \subset \text{span}\{v_1, v_2\}$. If we also show that

$$v_1 \in \text{span}\{u_1, u_2\} \tag{7.4}$$

$$v_2 \in \text{span}\{u_1, u_2\}, \tag{7.5}$$

we'll have that $\text{span}\{v_1, v_2\} \subset \text{span}\{u_1, u_2\}$. Moreover, for any two **subsets** A and B , the conditions $A \subset B$ and $B \subset A$ are equivalent to $A = B$. Applying this to $\text{span}\{u_1, u_2\}$ and $\text{span}\{v_1, v_2\}$, yields

$$(\text{span}\{u_1, u_2\} \subset \text{span}\{v_1, v_2\}) \text{and} (\text{span}\{v_1, v_2\} \subset \text{span}\{u_1, u_2\}) \iff (\text{span}\{u_1, u_2\} = \text{span}\{v_1, v_2\}).$$

Because $u_1 = v_1$, the conditions (7.2) and (7.4) are immediate. Hence, we only need to show conditions (7.3) and (7.5), that is, u_2 can be written as a linear combination of v_1 and v_2 , and similarly, v_2 can be written as a linear combination of u_1 and u_2 .

In our textbook, we show these conditions analytically. Below, we verify them in code!

```
1 # We check the span conditions  
2 # including the obvious: u1 = v1 and v1 = u1  
3 @show u1 - v1
```



```

4 println ( " ")
5 # define
6 a1 = -dot (u2, v1) / dot (v1, v1)
7 a2 = 1
8 # check v2 = a1*u1 + a2*u2
9 @show v2 - (a1*u1 + a2*u2)
10 println ( " ")
11 # define
12 b1 = dot (u2, v1) / dot (v1, v1)
13 b2 = 1
14 # check u2 = b1*v1 + b2*v2
15 @show u2 - (b1*v1 + b2*v2)

```

Output

```

u1 - v1 = [0.0; 0.0; 0.0; 0.0]

v2 - (a1 * u1 + a2 * u2) = [0.0; 0.0; 0.0; 0.0]

u2 - (b1 * v1 + b2 * v2) = [0.0; 0.0; 0.0; 0.0]

4×1 Matrix{Float64}:
 0.0
 0.0
 0.0
 0.0

```

Here are the analytical calculations, in case you are interested,

$$v_1 = u_1$$

$$v_2 = u_2 - \left(\frac{u_2 \bullet v_1}{v_1 \bullet v_1} \right) v_1$$

we deduce that

$$\mathbf{v_2 = a_1 u_1 + a_2 u_2}$$

with

$$a_1 = - \left(\frac{u_2 \bullet v_1}{v_1 \bullet v_1} \right)$$

$$a_2 = 1$$

and

$$\mathbf{u_2 = b_1 v_1 + b_2 v_2}$$

with

$$b_1 = \left(\frac{u_2 \bullet v_1}{v_1 \bullet v_1} \right)$$

$$b_2 = 1$$

7.3 Graphical Interpretation of Orthogonal Vectors and Another way to Understand the Span Condition

Figure 7.1 provides a graphical illustration of the action of Gram-Schmidt on a pair of linearly independent vectors $\{u_1, u_2\}$. Figure 7.1-(a) shows two randomly generated vectors. Then we apply **Gram-Schmidt with normalization to produce orthonormal**

vectors $\{v_1, v_2\}$, as shown in Fig. 7.1-(b).

$$\begin{aligned} v_1 &= u_1 \\ v_1 &= v_1 / \|v_1\| \\ v_2 &= u_2 - (u_2 \cdot v_1) v_1 \\ v_2 &= v_2 / \|v_2\| \end{aligned}$$

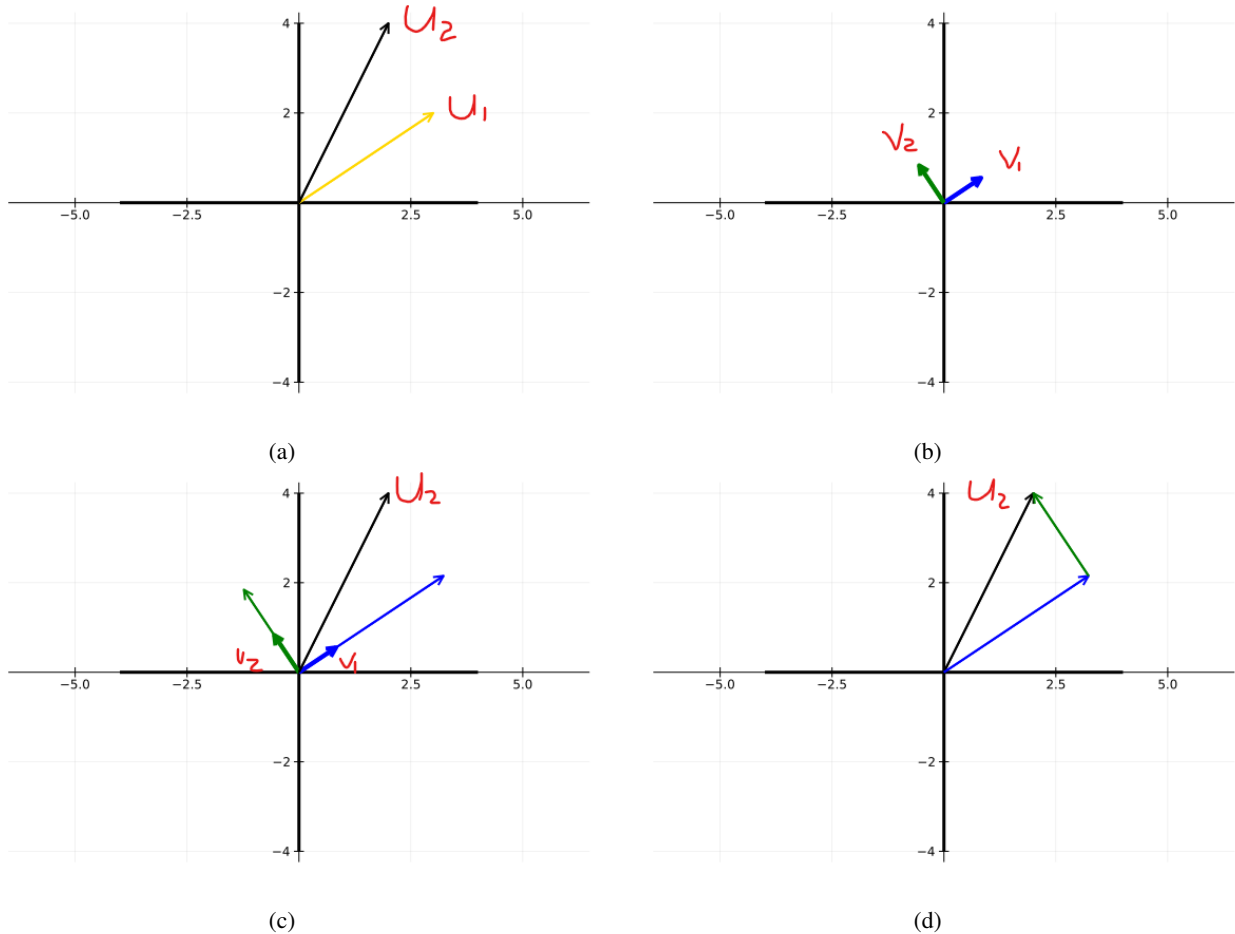


Figure 7.1: What does Gram-Schmidt do to a pair of linearly independent vectors? (a) $\{u_1, u_2\}$ are linearly independent, but they do not form a right angle. (b) $\{v_1, v_2\}$ is an **orthonormal** set produced by applying Gram-Schmidt to $\{u_1, u_2\}$. They clearly form a right angle. (c) The thin blue vector is $(u_2 \cdot v_1) \cdot v_1$. It goes by two names, the projection of u_2 along the direction v_1 or the component of u_2 along the direction v_1 . The thin green vector is $(u_2 \cdot v_2) \cdot v_2$. It goes by two names, the projection of u_2 along the direction v_2 or the component of u_2 along the direction v_2 . (d) From Gram-Schmidt, we know that $u_2 = (u_2 \cdot v_1) \cdot v_1 + (u_2 \cdot v_2) \cdot v_2$. This last plot shows the addition of the two vectors using the head to tail notion of vector addition that you may have come across in a physics course.

Because $\text{span}\{v_1, v_2\} = \text{span}\{u_1, u_2\}$, we know that we can express u_1 and u_2 as linear combinations of v_1 and v_2 . When $\{v_1, v_2\}$ are **orthonormal**, it is very easy to compute the linear combinations

$$\begin{aligned} u_1 &= \underbrace{(u_1 \cdot v_1)}_{\alpha_1} v_1 + \underbrace{(u_1 \cdot v_2)}_{\alpha_2} v_2 = \alpha_1 v_1 + \alpha_2 v_2 \\ u_2 &= \underbrace{(u_2 \cdot v_1)}_{\beta_1} v_1 + \underbrace{(u_2 \cdot v_2)}_{\beta_2} v_2 = \beta_1 v_1 + \beta_2 v_2. \end{aligned}$$

Yes, it is worth repeating: when $\{v_1, v_2\}$ are **orthonormal**, computing the coefficients in the linear combination is very easy. Figure 7.1-(c) shows the vector u_2 and the two parts of its linear combination in terms of $\{v_1, v_2\}$; namely, the thin blue vector is the

component of u_2 along v_1 , namely $(u_2 \bullet v_1)v_1$, while the thin green line is the component of u_2 along v_2 , namely $(u_2 \bullet v_2)v_2$. Figure 7.1-(d) shows that indeed, $u_2 = (u_2 \bullet v_1)v_1 + (u_2 \bullet v_2)v_2$, the sum of the two components.

When $\{v_1, v_2\}$ are **orthogonal but not orthonormal**, the expressions are still quite handy,

$$u_1 = \underbrace{\left(\frac{u_1 \bullet v_1}{v_1 \bullet v_1}\right)}_{\alpha_1} v_1 + \underbrace{\left(\frac{u_1 \bullet v_2}{v_2 \bullet v_2}\right)}_{\alpha_2} v_2 = \alpha_1 v_1 + \alpha_2 v_2$$

$$u_2 = \underbrace{\left(\frac{u_2 \bullet v_1}{v_1 \bullet v_1}\right)}_{\beta_1} v_1 + \underbrace{\left(\frac{u_2 \bullet v_2}{v_2 \bullet v_2}\right)}_{\beta_2} v_2 = \beta_1 v_1 + \beta_2 v_2.$$

```

1 # This code generates the plots in Figure 7.1
2 using Random
3 Random.seed! (31415926535897932384626433)
4 u1 = 3*randn(2,1)
5 u2 = 4*randn(2,1)
6 u1=[3; 2]
7 u2=[2; 4]
8 U = [u1 u2]
9 @show det(U' *U) # non-zero means independent
10 # G-S on two vectors (see big green box below)
11 @show v1 = u1
12 @show v2 = u2 - ( dot(u2, v1) /dot(v1, v1) ) *v1
13
14 v1 = v1/norm(v1)
15 v2 = v2/norm(v2)
16
17 t=LinRange(-4, 4,10); t=collect(t)
18 zero=0.0*t;
19
20 p1=plot(t,0.0.*t, color=:black, lw=3, legend=false, aspect_ratio=:equal, framestyle = :origin)
21 plot!(0.0.*t, t, color=:black, lw=3, legend=false)
22
23 plot!([0;u1[1:1]], [0;u1[2:2]], arrow=true, color=:gold, lw=2 )
24 plot!([0;u2[1:1]], [0;u2[2:2]], arrow=true, color=:black, lw=2 )
25
26 p2=plot(t,0.0.*t, color=:black, lw=3, legend=false, aspect_ratio=:equal, framestyle = :origin)
27 plot!(0.0.*t, t, color=:black, lw=3, legend=false)
28
29 plot!([0;v1[1:1]], [0;v1[2:2]], arrow=true, color=:blue, lw=4 )
30 plot!([0;v2[1:1]], [0;v2[2:2]], arrow=true, color=:green, lw=4 )
31
32
33 p3=plot(t,0.0.*t, color=:black, lw=3, legend=false, aspect_ratio=:equal, framestyle = :origin)
34 plot!(0.0.*t, t, color=:black, lw=3, legend=false)
35
36 u2Projv1 = ( dot(u2, v1) /dot(v1, v1) ) *v1
37 u2Projv2 = ( dot(u2, v2) /dot(v2, v2) ) *v2
38
39 plot!([0;u2Projv1[1:1]], [0;u2Projv1[2:2]], arrow=true, color=:blue, lw=2 )
40 plot!([0;u2Projv2[1:1]], [0;u2Projv2[2:2]], arrow=true, color=:green, lw=2 )
41
42 plot!([0;v1[1:1]], [0;v1[2:2]], arrow=true, color=:blue, lw=4 )
43 plot!([0;v2[1:1]], [0;v2[2:2]], arrow=true, color=:green, lw=4 )

```

```

44 plot!([0;u2[1:1]],[0;u2[2:2]], arrow=true, color=:black, lw=2 )
45
46
47 p4=plot(t,0.0.*t, color=:black, lw=3, legend=false, aspect_ratio=:equal,framestyle = :origin)
48 plot!(0.0.*t,t, color=:black, lw=3, legend=false)
49
50 plot!([0;u2[1:1]],[0;u2[2:2]], arrow=true, color=:black, lw=2 )
51
52 plot!([0;u2Projv1[1:1]],[0;u2Projv1[2:2]], arrow=true, color=:blue, lw=2 )
53 plot!([0;u2Projv2[1:1]].+u2Projv1[1:1],[0;u2Projv2[2:2]].+u2Projv1[2:2], arrow=true, color=:green,
54         lw=2 )
55
56 display(p1)
57 display(p2)
58 display(p3)
59 display(p4)

```

Output

Plots as shown in Fig. 7.1, though without the beautiful handwritten labels.

7.4 Gram-Schmidt Algorithm on a Set of four Vectors

The Gram-Schmidt Algorithm takes a set of linearly independent vectors and creates a set of orthogonal vectors that is (i) also linearly independent, and (ii), spans the same subspace as the original vectors. Here is the algorithm written out for a set of four linearly independent vectors. The hope is that you see the pattern and hence are ready to put the algorithm into a `for` loop.

Gram-Schmidt Process for 4 Vectors

Suppose that the set of vectors $\{u_1, u_2, u_3, u_4\}$ is linearly independent. Then the vectors $\{v_1, v_2, v_3, v_4\}$ are orthogonal, where

$$\begin{aligned}
 v_1 &= u_1 \\
 v_2 &= u_2 - \left(\frac{u_2 \bullet v_1}{v_1 \bullet v_1} \right) v_1 \\
 v_3 &= u_3 - \left(\frac{u_3 \bullet v_1}{v_1 \bullet v_1} \right) v_1 - \left(\frac{u_3 \bullet v_2}{v_2 \bullet v_2} \right) v_2 \\
 v_4 &= u_4 - \left(\frac{u_4 \bullet v_1}{v_1 \bullet v_1} \right) v_1 - \left(\frac{u_4 \bullet v_2}{v_2 \bullet v_2} \right) v_2 - \left(\frac{u_4 \bullet v_3}{v_3 \bullet v_3} \right) v_3
 \end{aligned}$$

Could you write out the line for a fifth vector?

```

1 using Random
2 Random.seed!(2525)
3 u1 = rand(4,1)
4 u2 = rand(4,1)
5 u3 = rand(4,1)
6 u4 = rand(4,1)
7 # G-S on four vectors (see big green box)
8 v1 = u1 # k=1
9 v2 = u2 - ( dot(u2,v1)/dot(v1,v1) ) * v1 # k=2, i = 1:1
10 v3 = u3 - ( dot(u3,v1)/dot(v1,v1) ) * v1 - ( dot(u3,v2)/dot(v2,v2) ) * v2 # k=3, i = 1:2
11 v4 = u4 - ( dot(u4,v1)/dot(v1,v1) ) * v1 - ( dot(u4,v2)/dot(v2,v2) ) * v2 - ( dot(u4,v3)/dot(v3,v3) )
    * v3 # k=4, i = 1:3

```

Output

```
4×4 Matrix{Float64}:
 2.54741  0.0      0.0      0.0
 0.0      0.183229  0.0      0.0
 0.0      0.0      0.0246929  0.0
 0.0      0.0      0.0      0.643923
```

We define $V := [v_1 \ v_2 \ v_3 \ v_4]$ and note that

$$V^T V = \begin{bmatrix} v_1^T \\ v_2^T \\ v_3^T \\ v_4^T \end{bmatrix} \cdot [v_1 \ v_2 \ v_3 \ v_4] = \begin{bmatrix} v_1^T \cdot v_1 & v_1^T \cdot v_2 & v_1^T \cdot v_3 & v_1^T \cdot v_4 \\ v_2^T \cdot v_1 & v_2^T \cdot v_2 & v_2^T \cdot v_3 & v_2^T \cdot v_4 \\ v_3^T \cdot v_1 & v_3^T \cdot v_2 & v_3^T \cdot v_3 & v_3^T \cdot v_4 \\ v_4^T \cdot v_1 & v_4^T \cdot v_2 & v_4^T \cdot v_3 & v_4^T \cdot v_4 \end{bmatrix},$$

that is,

$$[V^T V]_{ij} := v_i^T \cdot v_j = v_i \bullet v_j.$$

Therefore, if the set $\{v_1, v_2, v_3, v_4\}$ is orthogonal, we should have

$$V^T V = \begin{bmatrix} v_1^T \\ v_2^T \\ v_3^T \\ v_4^T \end{bmatrix} \cdot [v_1 \ v_2 \ v_3 \ v_4] = \begin{bmatrix} \|v_1\|^2 & 0 & 0 & 0 \\ 0 & \|v_2\|^2 & 0 & 0 \\ 0 & 0 & \|v_3\|^2 & 0 \\ 0 & 0 & 0 & \|v_4\|^2 \end{bmatrix},$$

where $\|v_i\|^2 = v_i^T \cdot v_i$. We check this.

```
1 V = [v1 v2 v3 v4]
2 cleanUp(V' * V)
```

Output

```
4×4 Matrix{Float64}:
 2.54741  0.0      0.0      0.0
 0.0      0.183229  0.0      0.0
 0.0      0.0      0.0246929  0.0
 0.0      0.0      0.0      0.643923
```

After normalization, we obtain ones on the diagonal.

```
1 # Normalize
2 V = [v1/norm(v1) v2/norm(v2) v3/norm(v3) v4/norm(v4)]
3 cleanUp(V' * V)
```

Output

```
4×4 Matrix{Float64}:
 1.0  0.0  0.0  0.0
 0.0  1.0  0.0  0.0
 0.0  0.0  1.0  0.0
 0.0  0.0  0.0  1.0
```

Here is the algorithm written closer to how it looks in a for loop.

```
1 # same algorithm, but written closer to how it looks in a for loop
2 #
3 using Random
4 Random.seed!(2525)
5 u1 = rand(4, 1)
6 u2 = rand(4, 1)
7 u3 = rand(4, 1)
```

```

8 u4 = rand(4, 1)
9 # G-S on four vectors (see big green box)
10 v1 = u1 # k=1
11 #
12 v2 = u2 # k=2
13 v2 = v2 - ( dot(u2, v1) / dot(v1, v1) ) * v1 # i=1:
14 #
15 v3 = u3 # k=3
16 v3 = v3 - ( dot(u3, v1) / dot(v1, v1) ) * v1 # i=1
17 v3 = v3 - ( dot(u3, v2) / dot(v2, v2) ) * v2 # i=2
18 #
19 v4 = u4 # k=4
20 v4 = v4 - ( dot(u4, v1) / dot(v1, v1) ) * v1 # i=1
21 v4 = v4 - ( dot(u4, v2) / dot(v2, v2) ) * v2 # i=2
22 v4 = v4 - ( dot(u4, v3) / dot(v3, v3) ) * v3 # i=3
23 # Normalize
24 V = [v1/norm(v1) v2/norm(v2) v3/norm(v3) v4/norm(v4) ]
25 cleanup(V' * V)

```

Output

```

4x4 Matrix{Float64}:
 1.0  0.0  0.0  0.0
 0.0  1.0  0.0  0.0
 0.0  0.0  1.0  0.0
 0.0  0.0  0.0  1.0

```

7.5 Gram-Schmidt Algorithm for a General Number of Linearly Independent Vectors

The Gram-Schmidt process on a set of linearly independent vectors $\{u_1, u_2, \dots, u_m\}$ is as follows

$$\begin{aligned}
 v_1 &= u_1 \\
 v_2 &= u_2 - \left(\frac{u_2 \bullet v_1}{v_1 \bullet v_1} \right) v_1 \\
 v_3 &= u_3 - \left(\frac{u_3 \bullet v_1}{v_1 \bullet v_1} \right) v_1 - \left(\frac{u_3 \bullet v_2}{v_2 \bullet v_2} \right) v_2 \\
 &\vdots \\
 v_k &= u_k - \sum_{i=1}^{k-1} \left(\frac{u_k \bullet v_i}{v_i \bullet v_i} \right) v_i \quad (\text{General Step})
 \end{aligned} \tag{7.6}$$

Gram-Schmidt Process as Pseudocode

We assume the columns of $U := [u_1 \ u_2 \ \dots \ u_m]$ are linearly independent. We build a set of orthonormal vectors as follows

```

V = [ ] # blank array
for k = 1 : Number of columns of U
    uk = U[:,k]
    vk = copy(uk)
    for i = 1:k-1
        vi = V[:,i]
        vk = vk -  $\frac{uk \bullet vi}{vi \bullet vi} \cdot vi$ 
    end
    V = [V vk/norm(vk)]
end

```

```

1 # GS in code
2 #
3 using LinearAlgebra
4 function grahm_schmidt(U)
5     nRowsU = size(U,1) # the 1 returns the number of rows
6     nColsU = size(U,2) # the 2 returns number of columns.
7     #
8     # We first create an empty matrix V that will hold all of the orthonormal vectors
9     V = Array{Float64,2}(undef, nRowsU, 0)
10    #
11    # Next, we loop over the columns of U
12    for k = 1:nColsU # loops through the columns of U
13        uk = U[:, k]
14        # next we set up for the general step of the G-S process
15        vk = copy(uk)
16        #
17        for i = 1:k-1
18            vi = V[:,i]
19            vk = vk - ( dot(uk, vi)/dot(vi, vi) ) *vi
20        end
21        #
22        V = [V vk/norm(vk)] # We add normalized columns to V
23    end
24    return V
25 end

```

Output

```
grahm_schmidt (generic function with 1 method)
```

```

1 # Friendly self test
2 Random.seed!(1999)
3 Ubig = rand(100, 27)
4 solTest = grahm_schmidt(Ubig)
5 T1=@assert isapprox(solTest[1, 1], 0.0834795, atol = 1E-6)
6 T2=@assert isapprox(solTest[100, 27], 0.0523471, atol = 1E-6)
7 T3 = @assert(norm(solTest'*solTest - I) < 1e-10 )
8 println("all nothings means likely correct")
9 [T1 T2 T3]

```

Output

```
all nothings means likely correct
```

```
1×3 Matrix{Nothing}:
 nothing nothing nothing
```

When we normalize the vectors as we go, we can slightly simplify the G-S Algorithm because $\|v_i\| = 1 \implies v_i \bullet v_i = 1$. This is highlighted below.

Gram-Schmidt Process as Pseudocode: **Take 2**

We assume the columns of $U := [u_1 \ u_2 \ \cdots \ u_m]$ are linearly independent. We build a set of spanning orthonormal vectors as follows, where we note that because we are normalizing as we go, $v_i \bullet v_i = 1$, and hence we can simplify the algorithm.

```
V=[] # blank array
for k = 1 : Number of columns of U
    uk = U[:,k]
    vk =copy(uk)
    for i = 1:k-1
        vi = V[:,i]
        # Because ||vi|| = 1 ==> vi • vi = 1
        # can replace vk = vk -  $\frac{uk \bullet vi}{vi \bullet vi} \cdot vi$  with
        vk = vk - (uk • vi) • vi
    end
    V = [V vk/norm(vk)]
end
```

7.6 Gram-Schmidt Algorithm without Assuming Linear Independence of the Starting Vectors

We can drop the assumption of linear independence if we add **one check to the code** and we keep track of the number of columns in the matrix V .

Gram-Schmidt w/o Assuming Linear Independence

We no longer assume the columns of $U := [u_1 \ u_2 \ \cdots \ u_m]$ are linearly independent. To account for this, we build a set of orthonormal vectors as follows

```
aTol = 1e-8
V=[] # blank array
for k = 1 : Number of columns of U
    uk = U[:,k]
    vk =copy(uk)
    for i = 1 : Number of columns of V
        vi = V[:,i]
        vk = vk - (uk • vi) • vi
    end
    if ||vk|| > aTol
        V = [V vk/norm(vk)]
    end
end
```

```
1 # This time we do not assume that the columns
2 # of U are linearly independent
3 function grahm_schmidt (U, aTol=1e-8)
4     ## BEGIN our code here
5     #
6     nRowsU = size (U, 1) # the 1 returns the number of rows
7     nColsU = size (U, 2) # the 2 returns number of columns.
8     #
9     # We create an empty matrix V that will hold all of the orthonormal vectors
```



```

10  V = Array{Float64,2}(undef, nRowsU, 0)
11  #
12  #start a for loop that runs the number of times that there are columns in U
13  for k in 1:nColsU
14      uk = U[:, k]
15      # next we set up for the general step of the G-S process
16      vk = copy(uk)
17      for i = 1:size(V,2) # loop over the number of columns of V
18          vi = V[:,i]
19          vk = vk - ( dot(uk, vi)/dot(vi, vi) ) *vi
20      end
21      #
22      if norm(vk) > aTol          # Trick? Not really. We only add a normalized vk
23          V = [V vk/norm(vk)]    # if vk is not approximately the zero vector. Make
sense?
24      end                          # And we ignore the vector otherwise
25  end
26  #
27  ## END our code here
28  return V #columns are orthonormal vectors
29 end

```

Output

gramm_schmidt (generic function with 2 methods)

Now we use our function to create a set of linearly independent vectors that span the columns of a wide matrix (more columns than rows). In Chapter 10 of our textbook, we will call such vectors a basis.

While there are several ways to approach a solution to this, but we'll use Gram-Schmidt and build an orthonormal set of vectors because (1) they are guaranteed to be linearly independent, AND (2) we know that GS is span preserving (after we discard the dependent vectors).

```

1  A=randn(5, 50)
2
3  V=gramm_schmidt(A)

```

Output

```

5×5 Matrix{Float64}:
 0.00322536  0.104034 -0.0666101 -0.0199444  0.992135
-0.739886  -0.485495 -0.294256  0.358629  0.0407671
 0.545578  -0.822757  0.0995872  0.0829251  0.0928525
 0.215882  0.256771  0.164351  0.927604  0.00205493
-0.329088  -0.10301  0.933835  -0.0605141  0.0733508

```

```

1  # check the columns of V are orthonormal
2  println("That's a very nice identity matrix!")
3  cleanUp(V'*V)

```

Output

That's a very nice identity matrix!

```

5×5 Matrix{Float64}:
 1.0  0.0  0.0  0.0  0.0
 0.0  1.0  0.0  0.0  0.0

```

```

0.0  0.0  1.0  0.0  0.0
0.0  0.0  0.0  1.0  0.0
0.0  0.0  0.0  0.0  1.0

```

7.7 Orthogonal Complement of a Set of Vectors

Consider a set of vectors $\{u_1, u_2, \dots, u_p\} \subset \mathbb{R}^n$, not necessarily linearly independent. Our goal is to compute its **orthogonal complement**, that is, the set of all vectors in \mathbb{R}^n that are orthogonal to each of the vectors u_k , $1 \leq k \leq p$. **The standard notation for the orthogonal complement of a set of vectors is**

$$\{u_1, u_2, \dots, u_p\}^\perp := \{x \in \mathbb{R}^n \mid u_k \bullet x = 0, 1 \leq k \leq p\}.$$

The symbol “ \perp ” is read “perp” and is short for perpendicular. It is placed as a right superscript on a set to denote the **orthogonal complement** of the set.

Step 1: We first apply G-S to $\{u_1, \dots, u_p\}$, yielding

$$\text{span}\{u_1, \dots, u_p\} = \text{span}\{v_1, \dots, v_r\},$$

where r is the number of linearly independent vectors in the set $\{u_1, \dots, u_p\}$, and the vectors $\{v_1, \dots, v_r\}$ are orthogonal or orthonormal (it’s our choice).

Step 2: Let $\{e_1, \dots, e_n\}$ be the columns of the $n \times n$ identity matrix, so that $\text{span}\{e_1, \dots, e_n\} = \mathbb{R}^n$. We continue applying G-S to $\{v_1, \dots, v_r, e_1, \dots, e_n\}$, to produce an orthogonal (or orthonormal) basis for all of \mathbb{R}^n , namely

$$\mathbb{R}^n = \text{span}\{v_1, \dots, v_k, e_1, \dots, e_n\} = \text{span}\{v_1, \dots, v_r, v_{r+1}, \dots, v_n\}.$$

Step 3: From G-S, $\{v_1, \dots, v_r\} \perp \{v_{r+1}, \dots, v_n\}$. From Step 1, we then conclude

$$\{v_1, \dots, v_r\} \perp \{v_{r+1}, \dots, v_n\} \iff \{u_1, \dots, u_p\} \perp \{v_{r+1}, \dots, v_n\}.$$

In other words,

$$\{u_1, \dots, u_p\}^\perp = \text{span}\{v_{r+1}, \dots, v_n\}.$$

Here is the idea implemented in code, with the vectors $\{u_1, \dots, u_p\}$ stacked up as the columns of a matrix U .

```

1 function myOrthogonalComplement (U, aTol=1e-8)
2     n, p=size (U)
3     myI=zeros (n, n) +I
4     U=[copy (U) myI]
5     V=Array{Float64, 2} (undef, n, 0)
6     # Build an orthonormal basis for the column span of transpose(A)
7     # It is not assumed that the columns are linearly independent in R^m
8     for k = 1:p
9         vi=U[:, k]
10        for i=1:size (V, 2)
11            vi= vi- (vi' *V[:, i]) *V[:, i]
12        end
13        norm_vi=sqrt (vi' *vi)
14        if norm_vi > aTol
15            V=[V vi/norm_vi]
16        end
17    end
18    dimColSpanU=size (V, 2)

```

```

19 # Now, we complete the above basis for the column span to a basis for all of R^n.
20 # Gram-Schmidt will make sure that these extra vectors are orthonormal to
21 # the column span of U, and hence they form the orthogonal complement of U
22 for k = p+1:n+p
23     vi=U[:,k]
24     for i=1:size(V,2)
25         vi= vi-(vi'*V[:,i])*V[:,i]
26     end
27     norm_vi=sqrt(vi'*vi)
28     if norm_vi > aTol
29         V=[V vi/norm_vi]
30     end
31 end
32 # We could easily have combined the two for loops, but we separated them so we could
33 # explain what is being done at each part of the computations
34 dimOrthogonalComplementU=n-dimColSpanU
35 if dimOrthogonalComplementU > 0
36     orthogonalComplementU=V[:,(dimColSpanU+1):end]
37 else
38     orthogonalComplementU=0.0*myI[:,1]
39 end
40 return orthogonalComplementU
41 end

```

Output

myOrthogonalComplement (generic function with 2 methods)

```

1 # Let's check the above
2 using Random
3 n=11; p=5
4 U=randn(n,p)
5 U = [U U] # make the columns dependent
6 M = myOrthogonalComplement(U)
7 r=size(M,2)
8 a=rand(r,1)
9 # form a random linear combination and check
10 # that it is orthogonal to the columns of U
11 x=M*a
12 for i = 1:size(U,2)
13     ui_col=U[:,i]
14     @show dot(ui_col,x) # transpose(ai_row) perpendicular to x ?
15 end
16 M

```

Output

```

dot(ui_col, x) = 1.0585650443511322e-15
dot(ui_col, x) = -1.1548457756088937e-15
dot(ui_col, x) = -2.7211455760065846e-15
dot(ui_col, x) = -8.282159261185579e-16
dot(ui_col, x) = -5.519198040808542e-15
dot(ui_col, x) = 1.0585650443511322e-15
dot(ui_col, x) = -1.1548457756088937e-15
dot(ui_col, x) = -2.7211455760065846e-15
dot(ui_col, x) = -8.282159261185579e-16
dot(ui_col, x) = -5.519198040808542e-15

```

11x6 Matrix{Float64}:

```

0.892501      7.23201e-17  -2.04308e-16  ...   4.24411e-16   4.62381e-16
-0.18157     0.767575   -5.26253e-16  ...   9.35674e-16   9.06197e-16
-0.153321    -0.0970827  0.237338     ...   9.15058e-17   5.29968e-16
0.191072     0.394995   -0.183774    ...   -8.92336e-16  -7.47455e-16
0.00534674  -0.298358  0.441247     ...   0.373253      1.46137e-16
0.0506156   -0.0377994  0.157102     ...   -0.411495     0.379856
-0.165052   -0.0948882 -0.376516    ...   0.400254      -0.437411
0.192435    0.101522   0.0771978    ...   -0.229906     -0.678507
0.0966389   -0.0995346  0.145413     ...   0.0980089     -0.264283
0.0818378   0.354351   0.560105     ...   0.487769      0.0562148
0.166008    -0.0124015 -0.455994    ...   0.480386      0.361945

```

7.8 Null Space of a Matrix using the Orthogonal Complement

Null Space of A Consists of Vectors Orthogonal to the Rows of A

Let A be an $n \times m$ matrix so that its rows are m -vectors.

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{21} & a_{22} & \cdots & a_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nm} \end{bmatrix} =: \begin{bmatrix} a_1^{\text{row}} \\ a_2^{\text{row}} \\ \vdots \\ a_n^{\text{row}} \end{bmatrix},$$

with $(a_i^{\text{row}})^\top \in \mathbb{R}^m$ for $1 \leq i \leq n$. Then

$$x \in \text{null}(A) \iff Ax = 0 \iff \left(\begin{bmatrix} a_1^{\text{row}} \\ a_2^{\text{row}} \\ \vdots \\ a_n^{\text{row}} \end{bmatrix} x = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \right) \iff \left(\begin{bmatrix} a_1^{\text{row}} \cdot x \\ a_2^{\text{row}} \cdot x \\ \vdots \\ a_n^{\text{row}} \cdot x \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \right).$$

However, recalling that $\langle x, y \rangle = x \bullet y := x^\top \cdot y$, we have that

$$a_i^{\text{row}} \cdot x = (a_i^{\text{row}})^\top \bullet x = \langle (a_i^{\text{row}})^\top, x \rangle, 1 \leq i \leq n,$$

and therefore,

$$x \in \text{null}(A) \iff x \perp (a_i^{\text{row}})^\top, 1 \leq i \leq n.$$

And we conclude that,

$$\text{null}(A) = \{(a_1^{\text{row}})^\top, \dots, (a_n^{\text{row}})^\top\}^\perp$$

Remark: “ \top ” denotes the transpose of a vector or matrix, while “ \perp ” denotes the orthogonal complement of a set of vectors.

```

1 # Our shortest function?
2 function myNullSpace(A, aTol=1e-8)
3     return myOrthogonalComplement(A', aTol)
4 end

```

Output

myNullSpace (generic function with 2 methods)

```

1 # Let's check the above
2 using LinearAlgebra
3 using Random
4 A=randn(3,5)
5 a=rand(1,2)
6 V = myNullSpace(A)
7 @show size(V)
8 # null space will have at least dimension 5 - 3 = 2
9 # form a random linear combination
10 @show x=a[1]*V[:,1] + a[2]*V[:,2]
11 for i = 1:size(A,1)
12     ai_row=A[i:i,:]
13     @show ai_row*x # transpose(ai_row) perpendicular to x ?
14 end
15 V

```

Output

```
size(V) = (5, 2)
```

```
x = a[1] * V[:, 1] + a[2] * V[:, 2] = [0.12274665165761715, 0.3105435916158756,
-0.37315446243641986, 0.022442592224478974, -0.5069618813368804]
```

```
ai_row * x = [1.5671029920338576e-16]
ai_row * x = [2.6975713230706956e-16]
ai_row * x = [-4.890711401061546e-16]
```

```
5×2 Matrix{Float64}:
 0.258452  -1.83849e-16
-0.0221855  0.603877
-0.203597  -0.519957
-0.556498  0.539292
-0.762607  -0.27229
```

7.9 Debugging: the Worst Case is When We Have No Error Messages

The standard Gram-Schmidt function we have been using assumes the columns of the input matrix are linearly independent. Our goal is to extend the function so that it works for inputs that include dependent vectors. Here is our strategy:

- Start with a function that works when the columns of the input matrix are linearly independent.
- Apply some dependent vectors and see what happens.
- Hope that the errors suggest solutions!

```

1 function grahm_schmidt(U)
2     nRowsU, nColsU = size(U)
3     V = Array{Float64,2}(undef, nRowsU, 0)
4     #
5     # Next, we loop over the columns of U
6     for k = 1:nColsU # loops through the columns of U
7         uk = U[:, k]
8         # next we set up for the general step of the G-S process
9         vk = copy(uk)
10        #
11        for i = 1:k-1
12            vi = V[:, i]

```

```

13     vk = vk - ( dot(uk, vi) / dot(vi, vi) ) * vi
14     end
15     #
16     V = [V vk/norm(vk)] # We do the normalization as we add a column to V
17     end
18     return V #columns are orthonormal vectors
19 end

```

Output

gramm_schmidt (generic function with 1 method)

```

1 Random.seed!(101010101)
2 U = randn(3,5) # More columns than rows, hence cannot all be independent.
3 V = gramm_schmidt(U)

```

Output

```

3×5 Matrix{Float64}:
 0.988747  -0.0127323  0.149056  0.0790081  0.0790081
 0.14833   -0.0460092  -0.987867  -0.870475  -0.870475
 0.0194357  0.99886   -0.0436029  -0.48583   -0.48583

```

The result is the worst possible thing for us! There is no error message. Our function returned five vectors, even though we know that is impossible. **Who would have thought that we'd be missing error messages being thrown?**

We have to come up with something to check on our own. Let's check if $V^T \cdot V = I$, the identity matrix.

```

1 display(V' * V)
2 cleanUp(V' * V)

```

Output

```

5×5 Matrix{Float64}:
 1.0          -4.11175e-17  3.87164e-16  -0.0604414  -0.0604414
 -4.11175e-17  1.0          3.06669e-16  -0.446233   -0.446233
 3.87164e-16  3.06669e-16  1.0          0.892874    0.892874
 -0.0604414   -0.446233    0.892874     1.0          1.0
 -0.0604414   -0.446233    0.892874     1.0          1.0

```

```

5×5 Matrix{Float64}:
 1.0          0.0          0.0          -0.0604414  -0.0604414
 0.0          1.0          0.0          -0.446233   -0.446233
 0.0          0.0          1.0          0.892874    0.892874
 -0.0604414  -0.446233    0.892874     1.0          1.0
 -0.0604414  -0.446233    0.892874     1.0          1.0

```

We see that we have not produced a matrix of orthogonal vectors, which is good, because we did not think it would work, but if we had not carefully checked, we would have been fooled, because, hey, our function did produce output! However, you have to remember the ol' programming adage, **Garbage In, Garbage Out!**

An idea: We should only add a vector to V when its norm is sufficiently large. Let's see if that helps. To save space, we also include the call to our function in the cell.

```

1 function gramm_schmidt(U, aTol=1e-10)
2     nRowsU, nColsU = size(U)
3     V = Array{Float64, 2}(undef, nRowsU, 0)
4     #
5     # Next, we loop over the columns of U

```

```

6   for k = 1:nColsU # loops through the columns of U
7       uk = U[:, k]
8       # next we set up for the general step of the G-S process
9       vk = copy(uk)
10      #
11      for i = 1:k-1
12          vi = V[:, i]
13          vk = vk - ( dot(uk, vi) / dot(vi, vi) ) * vi
14      end
15      #
16      if norm(vk) > aTol
17          V = [V vk/norm(vk)] # We do the normalization as we add a column to V
18      end
19  end
20  return V #columns are orthonormal vectors
21 end
22
23 Random.seed!(101010101)
24 U = randn(3,5) # More columns than rows, hence cannot all be independent.
25 V = gram_schmidt(U)

```

Output

BoundsError: attempt to access 3×3 Matrix{Float64} at index [1:3, 4]

Stacktrace:

```

[1] throw_boundserror(A::Matrix{Float64}, I::Tuple{Base.Slice{Base.OneTo{Int64}}, Int64})
    @ Base ./abstractarray.jl:651
[2] checkbounds
    @ ./abstractarray.jl:616 [inlined]
[3] _getindex
    @ ./multidimensional.jl:831 [inlined]
[4] getindex
    @ ./abstractarray.jl:1170 [inlined]
[5] gram_schmidt(U::Matrix{Float64}, aTol::Float64)
    @ Main ./In[19]:12
[6] gram_schmidt(U::Matrix{Float64})
    @ Main ./In[19]:2
[7] top-level scope
    @ In[19]:25
[8] eval
    @ ./boot.jl:360 [inlined]
[9] include_string(mapexpr::typeof(REPL.softscope), mod::Module, code::String,
filename::String)
    @ Base ./loading.jl:1094

```

This is progress. We have an error message! The message `Main ./In[19]:12` tells us to look around line 12. We insert appropriate display commands.

```

1 function gram_schmidt(U, aTol=1e-10)
2     nRowsU, nColsU = size(U)
3     V = Array{Float64, 2}(undef, nRowsU, 0)
4     #
5     # Next, we loop over the columns of U
6     for k = 1:nColsU # loops through the columns of U
7         uk = U[:, k]
8         # next we set up for the general step of the G-S process
9         vk = copy(uk)

```

```

10     #
11     for i = 1:k-1
12         @show i
13         @show size(V)
14         vi = V[:,i]
15         @show vi
16         vk = vk - ( dot(uk, vi) / dot(vi, vi) ) * vi
17     end
18     #
19     if norm(vk) > aTol
20         V = [V vk/norm(vk)] # We do the normalization as we add a column to V
21     end
22 end
23 return V #columns are orthonormal vectors
24 end
25
26 Random.seed!(101010101)
27 U = randn(3,5) # More columns than rows, hence cannot all be independent.
28 V = grahm_schmidt(U)

```

Output

```

i = 1
size(V) = (3, 1)
vi = [0.9887468476333943, 0.14833045243820744, 0.01943574476324906]
i = 1
size(V) = (3, 2)
vi = [0.9887468476333943, 0.14833045243820744, 0.01943574476324906]
i = 2
size(V) = (3, 2)
vi = [-0.012732300292710743, -0.04600918846453677, 0.9988598716066689]
i = 1
size(V) = (3, 3)
vi = [0.9887468476333943, 0.14833045243820744, 0.01943574476324906]
i = 2
size(V) = (3, 3)
vi = [-0.012732300292710743, -0.04600918846453677, 0.9988598716066689]
i = 3
size(V) = (3, 3)
vi = [0.14905555952154834, -0.9878670110173289, -0.04360285219348423]
i = 1
size(V) = (3, 3)
vi = [0.9887468476333943, 0.14833045243820744, 0.01943574476324906]
i = 2
size(V) = (3, 3)
vi = [-0.012732300292710743, -0.04600918846453677, 0.9988598716066689]
i = 3
size(V) = (3, 3)
vi = [0.14905555952154834, -0.9878670110173289, -0.04360285219348423]
i = 4
size(V) = (3, 3)
BoundsError: attempt to access 3x3 Matrix{Float64} at index [1:3, 4]

```

Stacktrace:

```

[1] throw_boundserror(A::Matrix{Float64}, I::Tuple{Base.Slice{Base.OneTo{Int64}}, Int64})
    @ Base ./abstractarray.jl:651
[2] checkbounds

```



```

@ ./abstractarray.jl:616 [inlined]
[3] _getindex
@ ./multidimensional.jl:831 [inlined]
[4] getindex
@ ./abstractarray.jl:1170 [inlined]
[5] grahm_schmidt(U::Matrix{Float64}, aTol::Float64)
@ Main ./In[20]:14
[6] grahm_schmidt(U::Matrix{Float64})
@ Main ./In[20]:2
[7] top-level scope
@ In[20]:28
[8] eval
@ ./boot.jl:360 [inlined]
[9] include_string(mapexpr::typeof(REPL.softscope), mod::Module, code::String,
filename::String)
@ Base ./loading.jl:1094

```

Now the message `BoundsError: attempt to access 3×3 MatrixFloat64 at index [1:3, 4]` means something to us. We see that V is 3×3 , and yet we are trying to extract out its 4-th column via `V[:, i]` (because we see that $i = 4$). Why would we be doing that? Well, the counter for $i = 1:k-1$ does not change with the size of V . So we fix that on line 11.

```

1 function grahm_schmidt(U, aTol=1e-10)
2     nRowsU, nColsU = size(U)
3     V = Array{Float64, 2}(undef, nRowsU, 0)
4     #
5     # Next, we loop over the columns of U
6     for k = 1:nColsU # loops through the columns of U
7         uk = U[:, k]
8         # next we set up for the general step of the G-S process
9         vk = copy(uk)
10        #
11        for i = 1:size(V, 2)
12            @show i
13            @show size(V)
14            vi = V[:, i]
15            @show vi
16            vk = vk - ( dot(uk, vi) / dot(vi, vi) ) * vi
17        end
18        #
19        if norm(vk) > aTol
20            V = [V vk/norm(vk)] # We do the normalization as we add a column to V
21        end
22    end
23    return V #columns are orthonormal vectors
24 end
25
26 Random.seed!(101010101)
27 U = randn(3,5) # More columns than rows, hence cannot all be independent.
28 V = grahm_schmidt(U)

```

Output

```

i = 1
size(V) = (3, 1)
vi = [0.9887468476333943, 0.14833045243820744, 0.01943574476324906]
i = 1
size(V) = (3, 2)
vi = [0.9887468476333943, 0.14833045243820744, 0.01943574476324906]

```

```

i = 2
size(V) = (3, 2)
vi = [-0.012732300292710743, -0.04600918846453677, 0.9988598716066689]
i = 1
size(V) = (3, 3)
vi = [0.9887468476333943, 0.14833045243820744, 0.01943574476324906]
i = 2
size(V) = (3, 3)
vi = [-0.012732300292710743, -0.04600918846453677, 0.9988598716066689]
i = 3
size(V) = (3, 3)
vi = [0.14905555952154834, -0.9878670110173289, -0.04360285219348423]
i = 1
size(V) = (3, 3)
vi = [0.9887468476333943, 0.14833045243820744, 0.01943574476324906]
i = 2
size(V) = (3, 3)
vi = [-0.012732300292710743, -0.04600918846453677, 0.9988598716066689]
i = 3
size(V) = (3, 3)
vi = [0.14905555952154834, -0.9878670110173289, -0.04360285219348423]

3x3 Matrix{Float64}:
 0.988747  -0.0127323  0.149056
 0.14833   -0.0460092  -0.987867
 0.0194357 0.99886   -0.0436029

```

It works! We should go back and remove all of the display commands, because they are annoying!

```

1 display(V' * V)
2 cleanUp(V' * V)

```

Output

```

3x3 Matrix{Float64}:
 1.0          -4.16334e-17  3.93023e-16
-4.16334e-17  1.0          3.1225e-16
 3.93023e-16  3.1225e-16  1.0

```

```

3x3 Matrix{Float64}:
 1.0  0.0  0.0
 0.0  1.0  0.0
 0.0  0.0  1.0

```

Reminder on Debugging

- Taken From the Pro-tip page at Georgia Tech for a graduate-level course: https://cse6040.gatech.edu/fa22/pro_tips.html
- Debugging is an iterative process where you must trace the undesired behavior back to the root cause. Sometimes it's simple; other times it can be frustratingly complex.
- Identify what is wrong. This needs to be as precise as possible. If you don't understand the error in the traceback - Google it!. Check the syntax, parameters, and inputs of any function calls.
- Identify where the wrong thing got set. This will usually be an assignment or a function call.
- Rinse and repeat until you have found and corrected the root cause.

7.10 (Optional Read) Comparing Julia's Native Null Space Command with Our Own

Exercise 7.1 Explain how you would check that Julia's native (builtin) function `nullspace(A)` and our function `myNullspace(A)` actually provide basis vectors for the same subspace. Alternatively, how would you show that they compute different things altogether?

Hint: The key thing is linear independence!

Solution: We set $V1 = \text{nullspace}(A)$ and $V2 = \text{myNullspace}(A)$. Then $\text{span}\{V1\} = \text{span}\{V2\}$ if, and only if, $[V1 \ V2]$ has the same number of linearly independent columns as $V1$ and $V2$. We can check these using our LDLT function from LAB5. ■

```
1 function ldlRob101(A)
2     epsilon = 1e-12
3     M = A' * A
4     n, m = size(A)
5     A_reduced = M
6     L = Array{Float64, 2}(undef, m, 0)
7     Id = zeros(m, m) + I
8     P = Id
9     D = zeros(m, m)
10    for i=1:m
11        ii = argmax(diag(A_reduced[i:m, i:m]))
12        mrow = ii[1] + (i-1)
13        if ! (i == mrow)
14            P[[i, mrow], :] = P[[mrow, i], :]
15            A_reduced[[i, mrow], :] = A_reduced[[mrow, i], :]
16            A_reduced[:, [i, mrow]] = A_reduced[:, [mrow, i]]
17        end
18        if (i > 1)
19            L[[i, mrow], :] = L[[mrow, i], :]
20        end
21        pivot = A_reduced[i, i]
22        if ! isapprox(pivot, 0, atol=epsilon)
23            D[i, i] = pivot
24            C = A_reduced[:, i] / pivot
25            L = [L C]
26            A_reduced = A_reduced - (C * pivot * C')
27        else
28            L = [L Id[:, i:m]]
29            break
30        end
31    end
32    diagD = diag(D)
33    return L, P, D, diagD
34 end
```

Output

```
ldlRob101 (generic function with 1 method)
```

```
1 using Random
2 A = randn(3, 5)
3 #
4 V1 = nullspace(A)
5 V2 = myNullspace(A)
6
7 L, P, D, diagD1 = ldlRob101(V1' * V1)
8 @show diagD1
```

```

9 L, P, D, diagD2 =ldltROB101 (V2' *V2)
10 @show diagD2
11 L, P, D, diagD12 =ldltROB101 ([V1 V2]' * [V1 V2])
12 @show diagD12

```

Output

```

diagD1 = [1.0000000000000004, 1.0]
diagD2 = [1.0, 0.9999999999999998]
diagD12 = [2.0000000000000013, 1.9999999999999993, 0.0, 0.0]

```

```

4-element Vector{Float64}:
 2.0000000000000013
 1.9999999999999993
 0.0
 0.0

```

Each of the above sets has 2 linearly independent vectors. Hence, our command and the native command agree. We'll next do a larger set of random vectors and see if they still agree!

```

1 using Random
2 A=randn(11, 50)
3 #
4 V1 = nullspace(A)
5 V2 = myNullspace(A)
6
7 tol=1e-8
8 L, P, D, diagD1 =ldltROB101 (V1' *V1)
9 numIndeVectorsV1=length ( findall (x->x>tol, diagD1) )
10 @show numIndeVectorsV1
11 L, P, D, diagD2 =ldltROB101 (V2' *V2)
12 numIndeVectorsV2=length (findall (x->x>tol, diagD2) )
13 @show numIndeVectorsV2
14 L, P, D, diagD12 =ldltROB101 ([V1 V2]' * [V1 V2])
15 numIndeVectorsV12=length (findall (x->x>tol, diagD12) )
16 @show numIndeVectorsV12

```

Output

```

numIndeVectorsV1 = 39
numIndeVectorsV2 = 39
numIndeVectorsV12 = 39

```

39

7.11 (Optional Read) Null Space of a Matrix without Mentioning the Orthogonal Complement

Our goal here is to show how to actually compute the null space of a matrix through the use of the Gram-Schmidt Process.

Null Space of A Consists of Vectors Orthogonal to the Rows of A

Let A be an $n \times m$ matrix so that its rows are m -vectors.

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{21} & a_{22} & \cdots & a_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nm} \end{bmatrix} =: \begin{bmatrix} a_1^{\text{row}} \\ a_2^{\text{row}} \\ \vdots \\ a_n^{\text{row}} \end{bmatrix},$$

with $(a_i^{\text{row}})^\top \in \mathbb{R}^m$ for $1 \leq i \leq n$. Then

$$x \in \text{null}(A) \iff Ax = 0 \iff \left(\begin{bmatrix} a_1^{\text{row}} \\ a_2^{\text{row}} \\ \vdots \\ a_n^{\text{row}} \end{bmatrix} x = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \right) \iff \left(\begin{bmatrix} a_1^{\text{row}} \cdot x \\ a_2^{\text{row}} \cdot x \\ \vdots \\ a_n^{\text{row}} \cdot x \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \right).$$

However, recalling that $\langle x, y \rangle = x \bullet y := x^\top \cdot y$, we have that

$$a_i^{\text{row}} \cdot x = (a_i^{\text{row}})^\top \bullet x = \langle (a_i^{\text{row}})^\top, x \rangle, 1 \leq i \leq n,$$

and therefore,

$$x \in \text{null}(A) \iff x \perp (a_i^{\text{row}})^\top, 1 \leq i \leq n.$$

```

1 # Let's check the above
2 using LinearAlgebra
3 using Random
4 A=randn(3, 5)
5 a=rand(1, 2)
6 V=nullspace(A)
7 x=a[1]*V[:, 1] + a[2]*V[:, 2]
8 for i = 1:size(A, 1)
9     ai_row=A[i:i, :]
10    @show dot(ai_row, x)
11 end
    
```

Output

```

dot(ai_row, x) = 8.942832355282709e-17
dot(ai_row, x) = -6.58380309139146e-17
dot(ai_row, x) = 1.1163776819499002e-16
    
```

How is the observation

$$x \in \text{null}(A) \iff x \perp (a_i^{\text{row}})^\top, 1 \leq i \leq n$$

at all useful to us? We lay it out in three steps:

Step 1: We first apply G-S to $\{(a_1^{\text{row}})^\top, \dots, (a_n^{\text{row}})^\top\}$, yielding

$$\text{span}\{(a_1^{\text{row}})^\top, \dots, (a_n^{\text{row}})^\top\} = \text{span}\{v_1, \dots, v_k\},$$

where k is the number of linearly independent vectors in the set $\{(a_1^{\text{row}})^\top, \dots, (a_n^{\text{row}})^\top\}$, and the vectors $\{v_1, \dots, v_k\}$ are orthogonal or orthonormal (it's our choice).

Step 2: We continue applying G-S to $\{v_1, \dots, v_k, e_1, \dots, e_m\}$, to produce an orthogonal (or orthonormal) basis for all of \mathbb{R}^m , namely

$$\mathbb{R}^m = \text{span}\{v_1, \dots, v_k, e_1, \dots, e_m\} = \text{span}\{v_1, \dots, v_k, v_{k+1}, \dots, v_m\}.$$

Step 3: From G-S, $\{v_1, \dots, v_k\} \perp \{v_{k+1}, \dots, v_m\}$. From Step 1, we then conclude

$$\{v_1, \dots, v_k\} \perp \{v_{k+1}, \dots, v_m\} \iff \{(a_1^{\text{row}})^\top, \dots, (a_n^{\text{row}})^\top\} \perp \{v_{k+1}, \dots, v_m\}.$$

In other words,

$$\boxed{\text{null}(A) = \text{span}\{v_{k+1}, \dots, v_m\}.$$

Here is the idea implemented in code.

```

1 function myNullspace (A, aTol=1e-8)
2     n, m=size (A)
3     myI=zeros (m, m)+I
4     U=[copy (A') myI]
5     V=Array{Float64, 2} (undef, m, 0)
6     # Build an orthonormal basis for the column span of transpose(A)
7     # It is not assumed that the columns are linearly independent in R^m
8     for k = 1:n
9         vi=U[:, k]
10        for i=1:size (V, 2)
11            vi= vi- (vi' *V[:, i]) *V[:, i]
12        end
13        norm_vi=sqrt (vi' *vi)
14        if norm_vi > aTol
15            V=[V vi/norm_vi]
16        end
17    end
18    dimColSpanAtranspose=size (V, 2)
19    # Now, we complete the above basis for the column span to a basis for all of R^m.
20    # Gram-Schmidt will make sure that these extra vectors are orthonormal to
21    # the column span of transpose(A), and hence they form a basis for null space of A
22    for k = n+1:n+m
23        vi=U[:, k]
24        for i=1:size (V, 2)
25            vi= vi- (vi' *V[:, i]) *V[:, i]
26        end
27        norm_vi=sqrt (vi' *vi)
28        if norm_vi > aTol
29            V=[V vi/norm_vi]
30        end
31    end
32    # We could easily have combined the two for loops, but we separated them so we could
33    # explain what is being done at each part of the computations
34    dimNullSpaceA=m-dimColSpanAtranspose
35    if dimNullSpaceA > 0
36        nullSpaceA=V[:, (dimColSpanAtranspose+1):end]
37    else
38        nullSpaceA=0.0*myI[:, 1]
39    end
40    return nullSpaceA
41 end

```

Output

myNullspace (generic function with 2 methods)

```

1 # Let's check the above
2 using LinearAlgebra

```

```

3 using Random
4 A=randn(3,5)
5 a=rand(1,2)
6 V = myNullspace(A)
7 # null space will have at least dimension 5 - 3 = 2
8 # form a random linear combination
9 x=a[1]*V[:,1] + a[2]*V[:,2]
10 for i = 1:size(A,1)
11     ai_row=A[i:i,:]
12     @show dot(ai_row,x) # transpose(ai_row) perpendicular to x ?
13 end

```

Output

```

dot(ai_row, x) = -1.3712315398837382e-15
dot(ai_row, x) = 3.4078577063098528e-15
dot(ai_row, x) = 1.5850056569889014e-15

```


Chapter 8

Julia Lab 8: Basis Vectors, Dimension, Coordinates, Eigenvectors and Eigenvalues

Learning Objectives

- Understand basis vectors and why they are so useful.
- Coming to grips with eigenvalues and eigenvectors.

Outcomes

- Basis vectors provide a compact means of defining a subspace
- Every vector in a subspace can be written as a linear combination of a set of basis vectors
- Because of the Gram-Schmidt Algorithm, once we find any basis for a subspace, we know how to generate one where the vectors are orthonormal
- The dimension of a subspace is the number of vectors in any basis.
- Even though basis vectors for a subspace are not unique, the number of vectors in a basis is unique
- Learn the `\` command in Julia for solving linear systems of equations. It will automatically use the best method available when solving the equation, such as LU or QR, without you having to think about it.
- A non-zero vector v is an eigenvector of a square matrix A if there exists a number λ such that $Av = \lambda v$. The number λ is called an eigenvalue. We will sometimes use e-value and e-vector for short.
- An $n \times n$ matrix has n^2 entries. It seems positively amazing that there can exist vectors such that multiplying v on the left by A is the same as multiplying v by a scalar, λ .
- Such an amazing property must have huge consequences for applied Linear Algebra, and it does!
- We'll learn the `eigen()` command/function in Julia to compute eigenvalues and eigenvectors of a matrix.

Either download Lab8 from our Canvas site or open up a Jupyter notebook so that you can enter code as we go. It is suggested that you have line numbering toggled on.

This lab goes all in on basis vectors.

8.1 Basis Vectors, Linear Independence, and Span

Basis Vectors and Dimension

Suppose that V is a subspace of \mathbb{R}^n . Then $\{v_1, v_2, \dots, v_k\}$ is a **basis for V** if

1. the set $\{v_1, v_2, \dots, v_k\}$ is linearly independent, and
2. $\text{span}\{v_1, v_2, \dots, v_k\} = V$, that is, every vector in V can be expressed as a linear combination of $\{v_1, v_2, \dots, v_k\}$.

The **dimension of V** is k , the number of basis vectors.

We note that the above definition applies to \mathbb{R}^n **because** \mathbb{R}^n is a subset of itself and it is closed under linear combinations. In particular, \mathbb{R}^n has dimension n , or we say that \mathbb{R}^n is an n -dimensional vector space.

Remark 8.1 We can check **linear independence** in two ways: define $A := [v_1 \ v_2 \ \dots \ v_k]$ and

- check $\det(A^\top \cdot A)$ is not equal to zero, or even better,
- do the LU factorization of $A^\top \cdot A$ and check that there are no zeros on the diagonal of U .

For the **span condition**, we want to check if a vector v can be written as a **linear combination** of $\{v_1, v_2, \dots, v_k\}$ or not. This is really asking if the equation

$$c_1 v_1 + c_2 v_2 + \dots + c_k v_k = \underbrace{[v_1 \ v_2 \ \dots \ v_k]}_A \underbrace{\begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_k \end{bmatrix}}_c = v$$

has a solution or not. In Chapter 8.2 of our textbook, we showed a cool way for checking this condition and for finding a solution, if it exists. We'll review that shortly.

```
1 using LinearAlgebra
2
3 # Determine linear independence
4 function is_independent(A, aTol=1e-10)
5     if false # change to true to use this version of the test
6         test = det(A' * A) # non-zero means columns of A are independent
7     else
8         F = lu(A' * A, check=false)
9         test = minimum(abs.(diag(F.U))) # non-zero means columns of A are independent
10    end
11    return !isapprox(test, 0.0, atol=aTol) # return true when test
12                                           # is large enough in magnitude
13 end
```

Output

```
is_independent (generic function with 2 methods)
```

```
1 # Check e1 and e2 are linearly independent, where
2 e1 = [0; 1]
3 e2 = [1; 0]
4 A = [e1 e2]
5 @show is_independent(A), size(A)
```

```

6 #
7 using Random
8 Random.seed!(2000)
9 A = randn(12, 12) # random square matrix
10 @show is_independent(A), size(A)
11 #
12 A = randn(12, 15) # random wide matrix (more columns than rows)
13 @show is_independent(A), size(A)
14 #
15 #
16 A = randn(12, 8) # random tall matrix (more rows than columns)
17 @show is_independent(A), size(A); # semicolon suppresses repeated output

```

Output

```

(is_independent(A), size(A)) = (true, (2, 2))
(is_independent(A), size(A)) = (true, (12, 12))
(is_independent(A), size(A)) = (false, (12, 15))
(is_independent(A), size(A)) = (true, (12, 8))

```

Least Squares Solutions to Linear Equations

Here are the main results on solutions to $Ax = b$ that minimize the squared error $\|Ax - b\|^2$.

- (a) $A^T A$ is invertible if, and only if, the columns of A are linearly independent.
- (b) If $A^T A$ is invertible, then there is a unique vector $x^* \in \mathbb{R}^m$ achieving $\min_{x \in \mathbb{R}^m} \|Ax - b\|^2$ and it satisfies the equation

$$(A^T A) x^* = A^T b. \quad (8.1)$$

- (c) Therefore, if $A^T A$ is invertible,

$$x^* = (A^T A)^{-1} A^T b \iff x^* = \arg \min_{x \in \mathbb{R}^m} \|Ax - b\|^2 \iff (A^T A) x^* = A^T b. \quad (8.2)$$

As you might guess by now, your instructors prefer that for large systems of equations, you solve (8.1) to obtain the least squares solution and avoid doing the inverse. For small systems, we'll cut you some slack.

Useful Remark for Solving Tall Linear Equations

Suppose that A is a “tall matrix” (more rows than columns) and suppose that $Ax = b$ has a solution (hence, b is a linear combination of the columns of A). Then, if the columns of A are linearly independent, you can compute the solution using (8.1) and the squared error will be zero, meaning x^* really is a solution to the equation because

$$\text{the squared error is zero} \iff \|Ax^* - b\|^2 = 0 \iff \|Ax^* - b\| = 0 \iff Ax^* - b = 0 \iff Ax^* = b.$$

Remark 8.2 The matrix $(A^T \cdot A)$ is square and invertible when $\{v_1, v_2, \dots, v_k\}$ is linearly independent. Hence we can solve for c by $(A^T \cdot A)c = A^T w$. This equation can be solved in many ways:

- using the inverse command, which we dislike;
- using LU or QR Factorization, which we love a lot; or
- using a new Julia trick involving the backslash command: $c = (A' * A) \backslash (A' * w)$ where here, the backslash command solves the equation $(A^T \cdot A)c = A^T w$ using what Julia finds to be the most efficient means at its disposal, typically one of LU or QR, but not always.

Exercise 8.3 Write a function to express a given vector w as a linear combination of basis vectors $\{v_1, v_2, \dots, v_k\}$, that is,

$$w = c_1 v_1 + c_2 v_2 + \dots + c_k v_k \iff Ac = w$$

where $A := [v_1 \ v_2 \ \dots \ v_k]$ and $c = \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_k \end{bmatrix}$.

Solution:

```

1 function expressLinearCombination(A, w, aTol=1e-4)
2     # Assume columns of V are linearly independent
3     c = (A'*A)\(A'*w) # (A'*A) c = A'*w
4     # You end here
5     if norm(A*c-w) > aTol
6         c = NaN # c = NaN will indicate that no solution exists
7     end
8     return c
9 end

```

Output

expressLinearCombination (generic function with 2 methods)

```

1 # A is a tall matrix, meaning more rows than columns.
2 A = [ -0.432768    0.038062   -0.187462
3       -0.0418031  -0.787585    0.0321116
4         0.640989   -0.036278    0.306395
5         0.601849    0.11336   -0.19077
6       -0.0801513  0.598622   -0.00674714
7       -0.177342   0.0758032   0.912968]
8 w = [-0.91903  -1.52064  1.48762  0.256259  1.09685  2.71317]'
9 w2 = [2.24379  1.67673  0.56574  0.67676  1.74866  -22.167922]'
10 #
11 c=expressLinearCombination(A, w)
12 if isnan(c[1])
13     println("There is no solution, and hence w is not a linear combo of columns of A")
14 end
15 @show c # should be a column vector with three components
16 @show norm(A*c-w)
17 #
18 c2=expressLinearCombination(A, w2)
19 @show c2
20 if isnan(c2[1])
21     println("There is no solution, and hence w2 is not a linear combo of columns of A")
22 end

```

Output

```

c = [1.0000011718917985; 2.0000006213876516; 3.0000020275852974]
norm(A * c - w) = 2.28426074545448e-6
c2 = NaN
There is no solution, and hence w2 is not a linear combo of columns of A

```

```

1 # Run me and note that our function works to find more than
2 # one linear combination at a time

```

```

3 c=expressLinearCombination(A, [w 2*w])
4 @show norm(A*c[:,1] - w), norm(A*c[:,2] - 2*w), norm(A*c-[w 2*w])
5 c

```

Output

```

(norm(A * c[:, 1] - w), norm(A * c[:, 2] - 2w), norm(A * c - [w 2w])) =
(2.28426074540208e-6, 4.56852149080416e-6, 5.107762304797088e-6)

```

```

3×2 Matrix{Float64}:
 1.0  2.0
 2.0  4.0
 3.0  6.0

```



8.2 Vector Space Coordinates and Vector Representations

A Basis Defines Coordinates

Suppose that V is a k -dimensional subspace of \mathbb{R}^n with basis $\{v_1, v_2, \dots, v_k\}$. We note that if $k = n$, then V is all of \mathbb{R}^n .

Due to the linear independence property of a basis, each $x \in V$ can be expressed (uniquely) as a linear combination of basis vectors

$$x = c_1 v_1 + c_2 v_2 + \dots + c_k v_k.$$

Stacking the coefficient c_1, c_2, \dots, c_k into a column vector yields

$$[x]_{\{v_1, \dots, v_k\}} := \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_k \end{bmatrix},$$

which forms the **coordinates** of $x \in V$ associated to the basis $\{v_1, v_2, \dots, v_k\}$.

Remark 8.4 Let's consider \mathbb{R}^2 , $\{v_1 = e_1, v_2 = e_2\}$, the so-called natural basis vectors. The span property of the vectors $\{e_1, e_2\} \subset \mathbb{R}^2$ is easy to check because, for any vector $x \in \mathbb{R}^2$, it can be written as a **linear combination** of $\{e_1, e_2\}$ per

$$x := \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = x_1 \begin{bmatrix} 1 \\ 0 \end{bmatrix} + x_2 \begin{bmatrix} 0 \\ 1 \end{bmatrix} = x_1 e_1 + x_2 e_2.$$

Therefore the coordinates of x associated to the basis $\{e_1, e_2\}$ are simply (x_1, x_2) . With other bases, the concept of representing vectors can be more interesting. We will limit ourselves to cases that we can visualize.

The following function will plot the coordinates associated with a given pair of basis vectors in \mathbb{R}^2 and plot a point that has been expressed in the given basis vectors.

```

1 using Plots
2
3 # Function of plotting a 2-d vector v using basis vectors {v1, v2} in R^2
4 function plot_with_basis(v1, v2, v)
5     t = LinRange(-4, 4, 10)
6     t = collect(t)
7     titre = "Basis Vectors"
8     p1 = plot(t, 0.0.*t, framestyle=:origin, aspect_ratio=1,
9             title=titre, legend=false, color=:black, lw=3)
10    p1 = plot!(0.0.*t, t, color=:black, lw=3)

```

```

11 for k = -4:4
12     X = k*v1[1] .+ t.*v2[1]
13     Y = k*v1[2] .+ t.*v2[2]
14     p1 = plot!(X, Y, color=:gray)
15 end
16 for k = -4:4
17     X = k*v2[1] .+ t.*v1[1]
18     Y = k*v2[2] .+ t.*v1[2]
19     p1 = plot!(X, Y, color=:gray)
20 end
21 p1 = plot!([0, v1[1]], [0, v1[2]], arrow=true, color=:blue, lw=4)
22 p1 = plot!([0, v2[1]], [0, v2[2]], arrow=true, color=:green, lw=4)
23 #
24 # Plot the vector v
25 titre = "Vector $v"
26 p2 = plot(t, 0.0.*t, framestyle=:origin, aspect_ratio=1,
27     title=titre, legend=false, color=:black, lw=3)
28 p2 = plot!(0.0.*t, t, color=:black, lw=3)
29 for k = -4:4
30     X = k*v1[1] .+ t.*v2[1]
31     Y = k*v1[2] .+ t.*v2[2]
32     p2 = plot!(X, Y, color=:gray)
33 end
34 for k = -4:4
35     X = k*v2[1] .+ t.*v1[1]
36     Y = k*v2[2] .+ t.*v1[2]
37     p2 = plot!(X, Y, color=:gray)
38 end
39 pt = v[1]*v1 + v[2]*v2
40 p2 = plot!([0, v[1]*v1[1]], [0, v[1]*v1[2]], color=:blue, lw=4)
41 p2 = plot!([v[1]*v1[1], v[1]*v1[1]+v[2]*v2[1]], [v[1]*v1[2], v[1]*v1[2]+v[2]*v2[2]], color
=:green, lw=4)
42 p2 = scatter!([pt[1]], [pt[2]], color=:red)
43 plot(p1, p2, layout=(1, 2))
44 end

```

Output

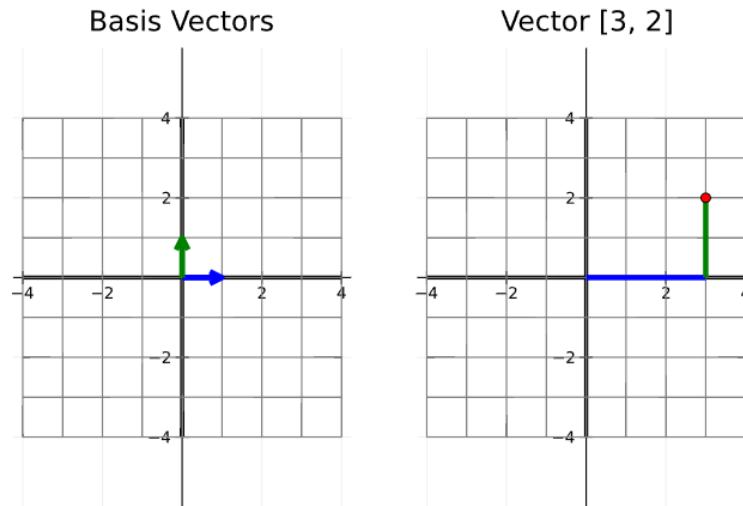
plot_with_basis (generic function with 1 method)

```

1 # We can plot a vector using the natural basis vectors
2 e1 = [1; 0]
3 e2 = [0; 1]
4 x = [3; 2] # means x = 3 e1 + 2 e2
5 plot_with_basis(e1, e2, x)

```

Output

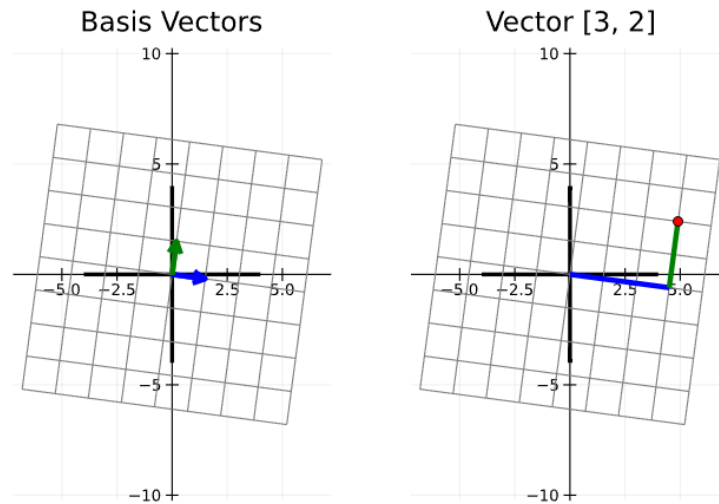


```

1 # We can plot a vector x with respect to a different set of basis vectors in R^2
2 v1 = [1.5; -0.2]
3 v2 = [0.2; 1.5]
4 x = [3; 2] # means x = 3 v1 + 2 v2
5 plot_with_basis(v1, v2, x)

```

Output



Next, we show an example in \mathbb{R}^3

```

1 z(x, y) = 2x+y
2 x = -1:1
3 y = -1:1
4 p1=plot(x, y, z, st=:surface, legend = false)
5 titre = "A 2D subspace in R3. All points satisfy 0 = 2x + y - z"
6 pl = plot!(titre = titre, xlabel="X", ylabel = "Y", zlabel="Z")
7 display(p1)
8 #
9 # Next, add basis vectors to the plot
10 A = [2 1 -1]
11 V = nullspace(A) # columns will be orthonormal vectors
12 #
13 v1=-V[:, 1]; v2=V[:, 2];

```

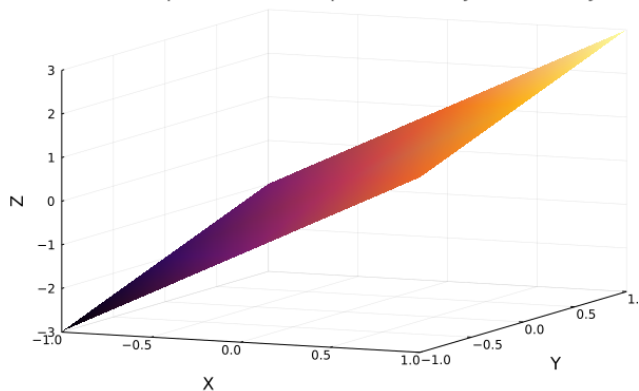
```

14 p2 = plot!([0, v1[1]], [0, v1[2]], [0, v1[3]], arrow=true, color=:blue, lw=4)
15 p2 = plot!([0, v2[1]], [0, v2[2]], [0, v2[3]], arrow=true, color=:green, lw=4)
16 titre = "Orthonormal basis vectors on our subspace"
17 p2 = plot! (title = titre)
18 display (p2)
19 #
20 # Finally, show grid lines on the plot
21 # In other words, show coordinate lines on the plot
22 t = LinRange(-1, 1, 10)
23 t = collect (t)
24 for k = -3:3
25     X = .33*k*v1[1] .+ t.*v2[1]
26     Y = .33*k*v1[2] .+ t.*v2[2]
27     Z = .33*k*v1[3] .+ t.*v2[3]
28     p3 = plot! (X, Y, Z, color=:gray)
29 end
30 for k = -3:3
31     X = .33*k*v2[1] .+ t.*v1[1]
32     Y = .33*k*v2[2] .+ t.*v1[2]
33     Z = .33*k*v2[3] .+ t.*v1[3]
34     p3 = plot! (X, Y, Z, color=:gray)
35 end
36 titre = "Basis vectors define coordinate lines on subspace"
37 p3 = plot! (title = titre)
38 display (p3)

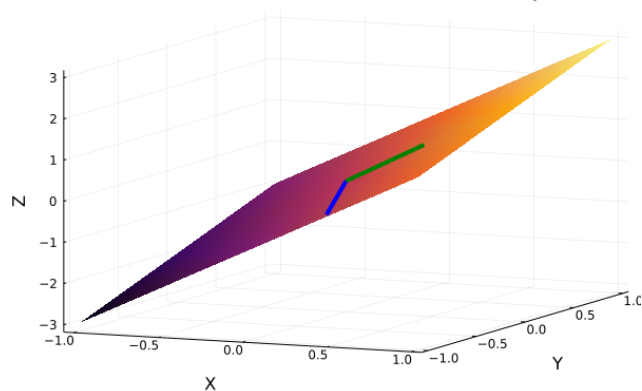
```

Output

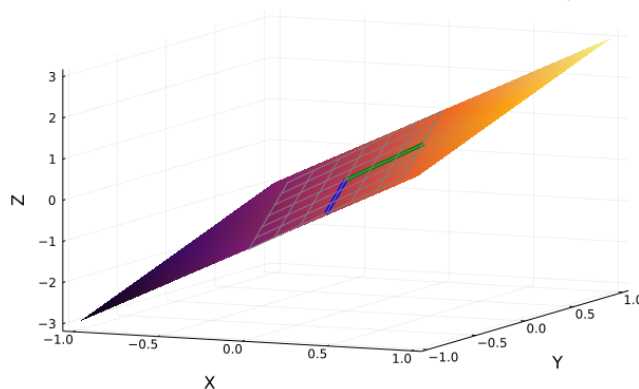
A 2D subspace in \mathbb{R}^3 . All points satisfy $0 = 2x + y - z$



Orthonormal basis vectors on our subspace



Basis vectors define coordinate lines on subspace



Exercise 8.5 Find a basis for the subspace of \mathbb{R}^3 defined by

$$V := \{(x, y, z) \mid z = 3x + 2y\}.$$

Solution:

```
1 ## BEGIN YOUR SOLUTION HERE
2 A = [3 2 -1]
3 Vbasis = nullspace(A)
4 ### END YOUR SOLUTION HERE
```

Output

```
3×2 Matrix{Float64}:
-0.534522  0.267261
 0.841427  0.0792865
 0.0792865  0.960357
```

```
1 # Friendly self test
2 T1 = @assert isapprox(A*Vbasis, [0 0], atol=1e-6)
3 println("all nothings means likely correct")
4 [T1]
```

Output

```
all nothings means likely correct

1-element Vector{Nothing}:
 nothing
```



8.3 Eigenvectors and Eigenvalues

In ROB 101, we are limiting ourselves to real eigenvalues and real eigenvectors. The Appendix of our textbook covers the case of complex eigenvalues and complex eigenvectors.

Eigen Stuff: Incomplete Definition but Good Enough to Get us Started

Let A be an $n \times n$ matrix with real coefficients. A scalar $\lambda \in \mathbb{R}$ is an **eigenvalue** of A , if there exists a non-zero vector $v \in \mathbb{R}^n$ such that $Av = \lambda v$. Any such vector v is called an **eigenvector** associated with λ .

We note that if v is an eigenvector, then so is αv for any $\alpha \neq 0$, and therefore, eigenvectors are not unique. The true definition is given in the Appendix of our textbook.

Finding Eigenvectors and Eigenvalues with Julia

We do not care to compute eigenvalues or eigenvectors by hand in ROB 101. We use Julia, instead!

```
1 Random.seed!(876543212345678);
2 A=randn(4,4)
3 A=A'+A # It's a fact that symmetric matrices have real eigenvalues
4 E=eigen(A)
5 @show E.values
6 E.vectors
```

Output

```
E.values = [0.06287200462929299, 0.6813033999332612, 2.9738855645273268,
4.4839915456638]
```

```
4 x 4 Matrix{Float64}:
 0.339074  0.0385456 -0.71411  -0.61122
 0.551488  0.528452  -0.226828  0.604275
-0.191731  0.824533   0.318536  -0.426521
 0.737651 -0.19849   0.58063  -0.281676
```

In the above, each column of `E.vectors` contains an e-vector of the matrix A . In fact, for $1 \leq i \leq$ number of columns of A , we have

$$A * E.vectors[:,i] = E.values[i] * E.vectors[:,i]$$

When the Eigenvalues are Real and Distinct, the Eigenvectors form a Basis of \mathbb{R}^n

Let A be an $n \times n$ matrix with real coefficients. If the eigenvalues $\{\lambda_1, \dots, \lambda_n\}$ are real and **distinct**, that is, $\lambda_i \neq \lambda_j$ for all $1 \leq i \neq j \leq n$, then the eigenvectors $\{v_1, \dots, v_n\}$ are real and provide a basis of \mathbb{R}^n .

The case of complex eigenvalues and eigenvectors is covered in the Appendix of our textbook. An interesting tidbit is that symmetric matrices always have real eigenvalues. Moreover, their eigenvectors can always be selected to form an orthogonal matrix.

```
1 using Random
2 Random.seed!(8765432123456)
3
4 # Symmetric matrices have real eigenvalues
5 A = randn(4,4)
6 A = A+A'
7
8 # Compute eigenvalues and eigenvectors
9 E = eigen(A)
10 @show E.values
11 E.vectors
```

Output

```
E.values = [-3.6258168896748075, -0.5790181480494843, 0.10365049366010881,
1.4790240511628157]
```

```
4x4 Matrix{Float64}:
 0.376015  0.798978 -0.133442  0.449933
 0.53203  0.223155  0.10347  -0.81021
 0.586088 -0.478348 -0.630802  0.17255
 0.481724 -0.288131  0.757348  0.333687
```

Next we show that the eigenvectors computed by Julia are orthonormal. Now, because $E.vectors' * E.vectors = I_{4 \times 4}$, we also see that the vectors are linearly independent. Why? Because $\det(I) = 1 \neq 0$.

```
1 cleanUp (E . vectors' * E . vectors)
```

Output

```
4x4 Matrix{Float64}:
 1.0  0.0  0.0  0.0
 0.0  1.0  0.0  0.0
 0.0  0.0  1.0  0.0
 0.0  0.0  0.0  1.0
```

Eigenvalues and Eigenvectors Explain how a Square Matrix acts on a Vector

Let A be an $n \times n$ real matrix with real eigenvalues $\{\lambda_1, \dots, \lambda_n\}$ that are **distinct**, that is, $\lambda_i \neq \lambda_j$ for all $1 \leq i \neq j \leq n$. It then follows that the eigenvectors $\{v_1, \dots, v_n\}$ provide a basis for \mathbb{R}^n . Let $x \in \mathbb{R}^n$ be arbitrary and write it as a linear combination of the basis of eigenvectors

$$x = \alpha_1 v_1 + \alpha_2 v_2 + \dots + \alpha_n v_n. \quad (8.3)$$

Then because $Av_i = \lambda_i v_i$,

$$Ax = \alpha_1 \lambda_1 v_1 + \alpha_2 \lambda_2 v_2 + \dots + \alpha_n \lambda_n v_n. \quad (8.4)$$

If we apply A to both sides of (8.4), we obtain

$$\begin{aligned} A^2 x &= \alpha_1 \lambda_1 A v_1 + \alpha_2 \lambda_2 A v_2 + \dots + \alpha_n \lambda_n A v_n \\ &= \alpha_1 (\lambda_1)^2 v_1 + \alpha_2 (\lambda_2)^2 v_2 + \dots + \alpha_n (\lambda_n)^2 v_n, \end{aligned} \quad (8.5)$$

where $A^2 := A \cdot A$ and we have used again, $Av_i = \lambda_i v_i$. Moreover, using this fact iteratively yields that, for all $k \geq 2$,

$$A^k x = \alpha_1 (\lambda_1)^k v_1 + \alpha_2 (\lambda_2)^k v_2 + \dots + \alpha_n (\lambda_n)^k v_n. \quad (8.6)$$

Equation (8.6) may not seem very important at first glance, but it is really a gold mine! We next illustrate this for the special case that $x = v_i$, one of the eigenvectors of a matrix A .

For context, in Project 3, we will model a Segway as a “discrete-time dynamic system” of the form

$$x_{k+1} = Ax_k.$$

When $\|x_k\| \rightarrow \infty$ as the time index k gets large, we’ll interpret that as the Segway falling over! On the other hand, $\|x_k\|$ staying “small” for all $k > 0$ will correspond to the Segway remaining upright.

```
1 using Random
2 Random.seed! (876543212345678)
3 using LinearAlgebra
4 #
5 # Recall: Symmetric matrices have real eigenvalues
6 #
7 A = randn(4, 4)
8 A = A' * A
9 #
10 # Compute eigenvalues and eigenvectors
11 E = eigen(A)
12 @show E.values
13 E.vectors
```

Output

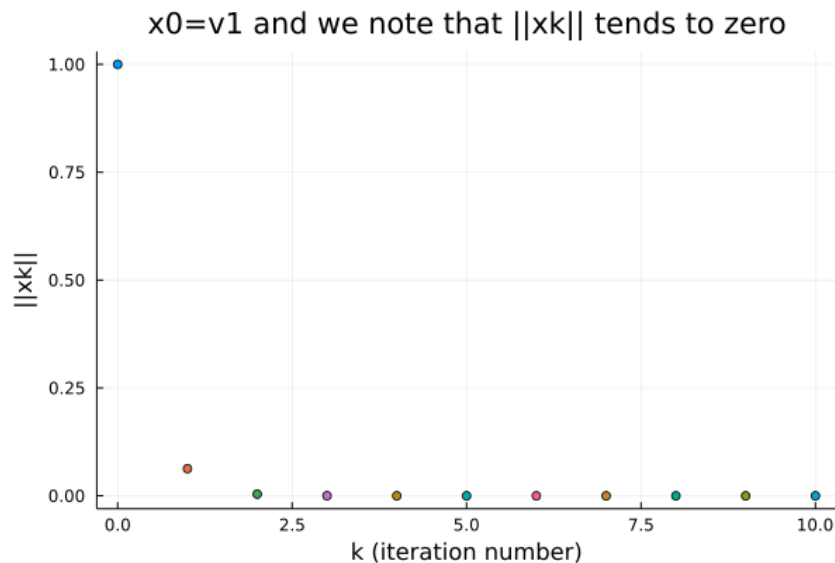
E.values = [0.06287200462929261, 0.6813033999332586, 2.9738855645273254, 4.483991545663797]

```
4×4 Matrix{Float64}:
 0.339074  0.0385456 -0.71411  0.61122
 0.551488  0.528452 -0.226828 -0.604275
-0.191731  0.824533  0.318536  0.426521
 0.737651 -0.19849  0.58063  0.281676
```

In the following cell, we use the 4×4 matrix A defined above and see what happens for different initial conditions, x_0 . **It is important to observe that the e-values of A are E.values = [0.063, 0.68, 2.97, 4.48] and note that the first two have magnitude less than one and the second two have magnitude greater than one.**

```
1 # When x0 equals the eigenvector v1, we can observe norm(xk) goes to 0
2 using Plots
3 x = E.vectors[:,1]
4 p = scatter([0], [norm(x)], legend=false, xlabel="k (iteration number)", ylabel="||xk||")
5 for k = 1:10
6     x = A*x
7     # Remark xk = A^k * x0, where x0 = E.vectors[:,1]
8     p = scatter!([k], [norm(x)])
9 end
10 titre="x0=v1 and we note that ||xk|| tends to zero"
11 plot!(title = titre)
12 png("x0Equalsv1")
13 display(p)
```

Output



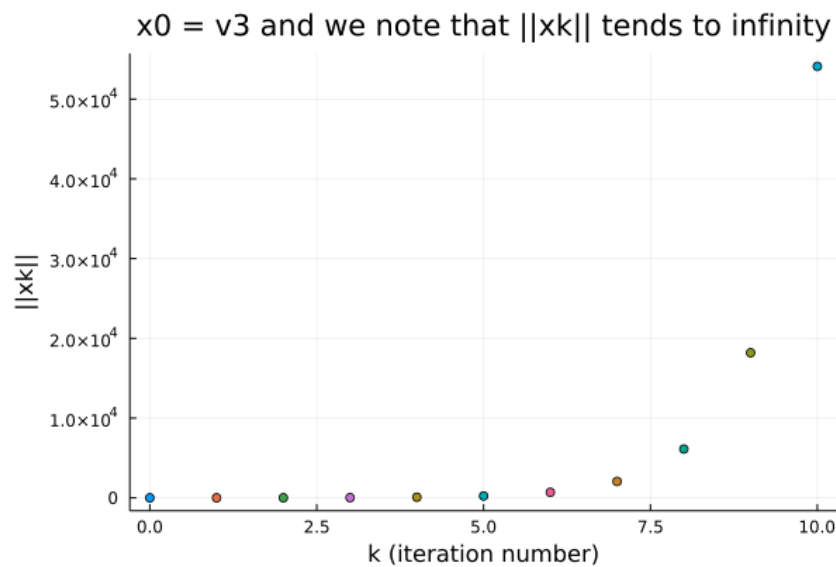
```
1 # When x0 equals the eigenvector v3, we can observe norm(xk) goes to infinity
2 x = E.vectors[:,3]
3 p = scatter([0], [norm(x)], legend=false, xlabel="k (iteration number)", ylabel="||xk||")
4 for k = 1:10
5     x = A*x
6     # Remark xk = A^k * x0, where x0 = E.vectors[:,3]
7     p = scatter!([k], [norm(x)])
```

```

8 end
9 titre="x0 = v3 and we note that ||xk|| tends to infinity"
10 plot!(title = titre)
11 png("x0Equalsv3")
12 display(p)

```

Output



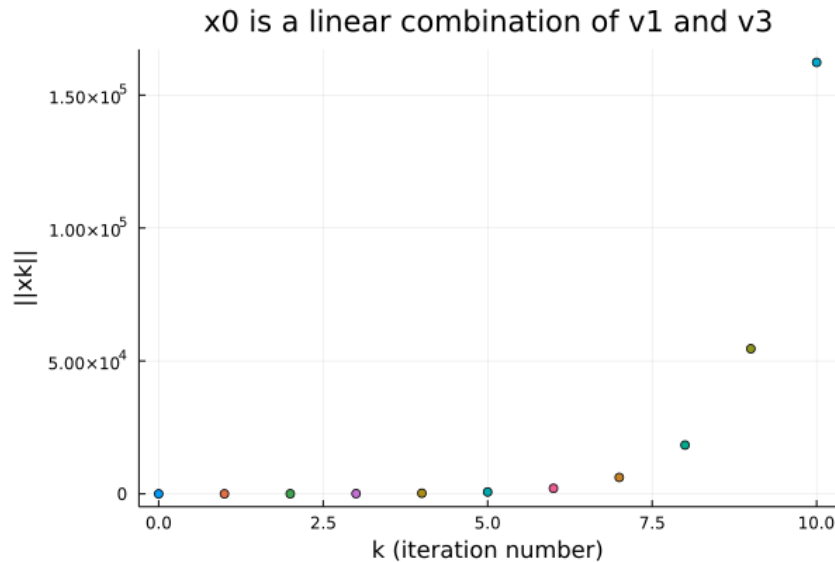
Can you “guess” what will happen when we take $x_0 = 2v_1 + 3v_3$? What happens if you add a very large quantity to a much smaller quantity? The larger quantity dominates!

```

1 # What will happen when x0 = 2v1+3v3?
2 x = 2*E.vectors[:,1] + 3*E.vectors[:,3]
3 p = scatter([0], [norm(x)], legend=false, xlabel="k (iteration number)", ylabel="||xk||")
4 for k = 1:10
5     x = A*x
6     # Remark xk = A^k * x0, where x0 = 2*E.vectors[:,1] + 3*E.vectors[:,3]
7     p = scatter!([k], [norm(x)])
8 end
9 titre="x0 is a linear combination of v1 and v3"
10 plot!(title = titre)
11 png("x0EqualsLinearCombv1andv3")
12 display(p)

```

Output



Here is what is happening. When $x_0 = v_1$, where v_1 is the eigenvector corresponding to eigenvalue λ_1 , we have

$$x_1 = Ax_0 = Av_1 = \lambda_1 v_1.$$

It follows that

$$x_2 = Ax_1 = A(\lambda_1 v_1) = \lambda_1 Av_1 = (\lambda_1)^2 v_1.$$

If $|\lambda_1| < 1$, then $(\lambda_1)^2$ is smaller and higher powers result in even smaller magnitudes.

In general, $x_k = (\lambda_1)^k v_1$. Therefore, when $|\lambda_1| < 1$, $\|x_k\| \rightarrow 0$ as $k \rightarrow \infty$ because $|\lambda_1|^k \rightarrow 0$ as $k \rightarrow \infty$. Similarly, when $x_0 = v_3$, we have $x_k = (\lambda_3)^k v_3$. Therefore, because $|\lambda_3| > 1$, $\|x_k\| \rightarrow \infty$.

Exercise 8.6 Describe the set of all initial conditions x_0 for which x_k stays bounded in our example.

Hint: The set is a two-dimensional subspace of \mathbb{R}^4 .

Solution: We'll call the set S_{bdd} , where bdd is short for bounded. Then

$$S_{\text{bdd}} = \text{span}\{v_1, v_2\}$$

because the eigenvalues associated with v_1 and v_2 have magnitude less than one. ■

Exercise 8.7 Repeat the exercise for the following matrix.

Hint: Note that the magnitude of $\lambda_1 = -3.6$ is $|-3.6| = 3.6!$

```

1 using Random
2 Random.seed!(8765432123456)
3
4 # Symmetric matrices have real eigenvalues
5 A = randn(4, 4)
6 A = A+A'
7
8 # Compute eigenvalues and eigenvectors
9 E = eigen(A)
10 @show E.values
11 E.vectors

```

Output

```
E.values = [-3.6258168896748075, -0.5790181480494843, 0.10365049366010881,
1.4790240511628157]
```

```
4×4 Matrix{Float64}:
 0.376015  0.798978 -0.133442  0.449933
 0.53203  0.223155  0.10347  -0.81021
 0.586088 -0.478348 -0.630802  0.17255
 0.481724 -0.288131  0.757348  0.333687
```

Solution:

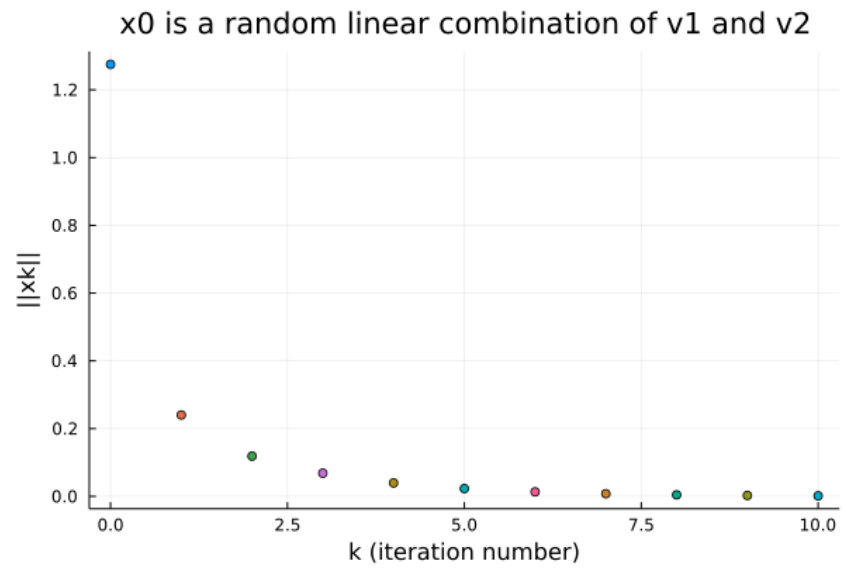
```
1 # Solution here
2 #v1 =
3 #v2 =
4 ## PLACE YOUR SOLUTION HERE
5 v1 = E.vectors[:,2]
6 v2 = E.vectors[:,3]
7 [v1 v2]
```

Output

```
4×2 Matrix{Float64}:
 0.0385456 -0.71411
 0.528452 -0.226828
 0.824533  0.318536
-0.19849  0.58063
```

```
1 # To confirm this, we take a random linear combination of the vectors v1 and v2
2 c=randn(2,1)
3 x0=c[1]*v1 + c[2]*v2
4 # What will happen when x0 = c[1]*v1 + c[2]*v2 ???
5 x = x0
6 p = scatter([0], [norm(x)], legend=false, xlabel="k (iteration number)", ylabel="||xk||")
7 for k = 1:10
8     x = A*x
9     # Remark xk = A^k * x0, where x0 = 2*E.vectors[:,1] + 3*E.vectors[:,3]
10    p = scatter!([k], [norm(x)])
11 end
12 titre="x0 is a random linear combination of v1 and v2"
13 plot!(title = titre)
14 png("x0EqualsRandomLinearCombv1andv2")
15 display(p)
```

Output



Chapter 9

Julia Lab 9: Modeling, Simulating, and Controlling a Mobile Robot

Learning Objectives

- Equations with derivatives are called (ordinary) differential equations, or ODEs for short.
- We model mobile robots with ODEs.
- Simulation means to solve an ODE numerically with a computer.
- Balancing is a key problem in robotics.

Outcomes

- Learn about states and controls in ODE models.
- Learn Euler's method for numerically approximating solutions to ODEs.
- Apply Euler's method to simulate several simple examples.
- Learn about the double inverted pendulum.
- Simulate the double inverted pendulum via Euler's method.
- For linear models, learn how to compute control sequences that drive the system from an initial state to a desired final state.
- Feedback means that the values in a control sequence are defined to depend on state of the system.
- Our main feedback control method involves least squares solutions of underdetermined equations from Chapter 9 or our textbook.
- Model Predictive Control (MPC) is based on iteratively computing new least squares solutions as the state of the system evolves with time. <https://www.youtube.com/watch?v=8U0xiOkDcmw>
- Balance the double inverted pendulum via MPC.

Either download Lab9 from our Canvas site or open up a Jupyter notebook so that you can enter code as we go. It is suggested that you have line numbering toggled on.

9.1 Motivation

- MS-Degree-level feedback control of the double inverted pendulum on a cart, <https://www.youtube.com/watch?v=B6vr1x6KDaY>. While our work will be simpler than what is shown in this video, we'll still launch you on the path to designing your own feedback control for interesting robot-like devices.
- Advanced Undergrad-level feedback control of the double inverted pendulum, https://www.youtube.com/watch?v=W5q_eYd9bfY. You need at least EECS 460.
- A self-balancing stick, another cool feedback control problem: <https://www.youtube.com/watch?v=woCdjbsjBpg>. This system depends on controlling angular momentum, something you learn about in Physics 140.
- The feedback control of the Michigan Cassie bipedal robot combines ideas from the above with Differential Geometry, <https://www.youtube.com/watch?v=utWqXZwTIbQ>. Differential Geometry is a generalized form of calculus that works on surfaces, such as spheres, and more exotic things, like a Klein Bottle.
- Professor Grizzle started his work on bipedal robots with the simpler robot, Rabbit; see <https://www.youtube.com/watch?v=3m-64ZdkuAg> and <https://www.youtube.com/watch?v=tHiqlSQsUnA>.
- Step by step, pun intended, this led to MABEL running at 3 m/s https://www.youtube.com/watch?v=x1Owk6_xpWo.
- And then Cassie on the Wave Field, <https://youtu.be/V36DCsc6iio?t=84>.

Terms in the Model ...Oh my!

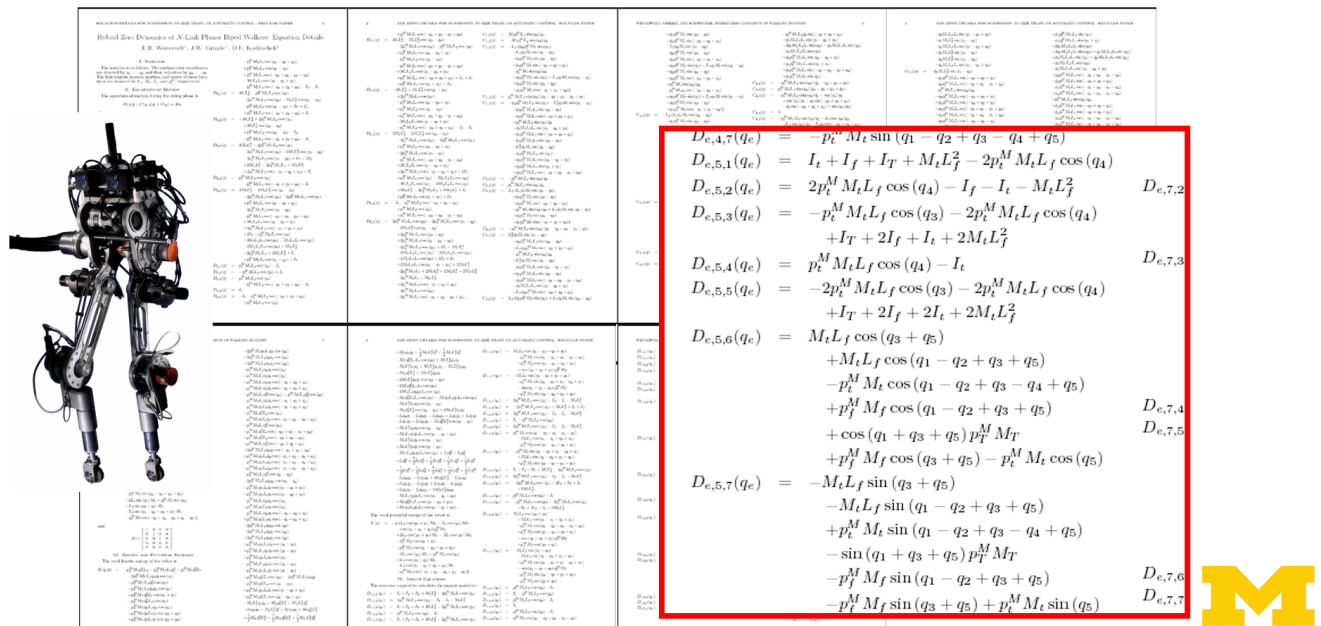


Figure 9.1: Terms in the ODE that describes (models) the bipedal robot named “Rabbit” by its French designers are shown in this figure. A model of the Cassie or Digit bipedal robots is more than 100 times larger. By combining good mathematics and programming skills, Prof. Grizzle’s students use these models to design bipedal walking and running gaits; see <https://www.youtube.com/user/DynamicLegLocomotion>.

9.2 What is an ODE?

Equations with a derivative in them are called **ordinary differential equations** or **ODEs for short**. Here is what one may look like,

$$\dot{x}(t) = f(x(t), u(t)) \quad (9.1)$$

where t is time, $x(t)$ is a vector of variables that we call “states”, $\dot{x}(t) := \frac{dx(t)}{dt}$ is the rate of change of $x(t)$ with respect to time, and $u(t)$ is a vector of input variables, such as a motor torque, that we can “control”. This lab goes all in on ODEs. Why?

- In Robotics, we use differential equations to understand the motion of robots, such as Cassie Blue, or in the case of Project #3, a Segway.
- Because we have studied iterative methods for solving equations, such as the Bisection Method, the Newton-Raphson Algorithm, and Gradient Descent, we are well positioned to approach the solution of differential equations through the lens of iteration.
- The topic of differential equations is typically delayed until a third- or fourth-semester Calculus course. Such courses focus almost exclusively on closed-form solutions to differential equations, which is a gnarly subject. We, however, will take a numerical approach and thereby allow you to see physics and calculus in a new light.
- The differential equations associated with robot models, such as Rabbit shown in Fig. 9.1, do not possess closed-form solutions! The equations have solutions, but literally, it has been proven that the solutions cannot be expressed in terms of trigonometric functions, polynomials, ratios, and radicals of such functions. While closed-form solutions are nice when they exist, as a Robotician, one has to face reality and deal with the challenges at hand. For more information, see https://en.wikipedia.org/wiki/Three-body_problem#/media/File:Three-body_Problem_Animation_with_COM.gif and https://en.wikipedia.org/wiki/Three-body_problem.

In Robotics, we typically deal with ODEs that look like this

$$D(q(t))\ddot{q}(t) + C(q(t), \dot{q}(t))\dot{q}(t) + G(q(t)) = Bu(t), \quad (9.2)$$

where

- t represents time in seconds,
- $q(t)$ is an n -vector, consisting of positions and angles,
- $\dot{q}(t) := \frac{dq(t)}{dt}$ is the rate of change of q ,
- $\ddot{q}(t) := \frac{d^2q(t)}{dt^2} = \frac{d\dot{q}(t)}{dt}$ is the rate of change of $\dot{q}(t) = \frac{dq(t)}{dt}$,
- D is an $n \times n$ matrix whose entries depend on q ,
- C is an $n \times n$ matrix whose entries depend on q and \dot{q} ,
- G is $n \times 1$ vector whose entries depend on q ,
- B is an $n \times m$ matrix, and
- u is an m -vector of controls.

Figure 9.1 shows in very small font some of the terms in the various matrices and vectors for a specific robot. The marriage of math and software allows us to analyze such models for very complicated robots.

We can rewrite (9.2) in the form (9.1) by defining the state as

$$x(t) := \begin{bmatrix} x_1(t) \\ x_2(t) \end{bmatrix} := \begin{bmatrix} q(t) \\ \dot{q}(t) \end{bmatrix},$$

so that

$$\dot{x}(t) = \begin{bmatrix} x_2(t) \\ (D(x_1(t)))^{-1} (-C(x_1(t), x_2(t))x_2(t) - G(x_1(t)) + Bu(t)) \end{bmatrix} =: f(x(t), u(t)). \quad (9.3)$$

In later Robotics courses, you will skip the rewriting step and learn how to work directly with (9.2) and some of its generalizations!

9.3 Euler's Method for Numerically Solving ODEs

We assume that you have not yet taken any Calculus, and therefore in particular, we assume that you have not yet taken a course on Differential Equations¹. We will emphasize in ROB 101 a very simple numerical method for (approximately) solving differential equations that is due to Euler. You can read more about it in case you are interested:

- Wikipedia https://en.wikipedia.org/wiki/Euler_method
- Khan Academy <https://www.khanacademy.org/math/ap-calculus-bc/bc-differential-equations-new/bc-7-5/v/eulers-method>

The basic idea is based on our forward difference method for computing a derivative of a function. We will assume our function is $x(t)$ and we have an ODE

$$\frac{dx(t)}{dt} = f(x(t)), \quad (9.4)$$

where t represents time.

To develop a numerical approximation to the ODE, we recall that

$$\frac{dx(t)}{dt} \approx \frac{x(t+dt) - x(t)}{dt}. \quad (9.5)$$

Hence, our ODE can be approximated as

$$\frac{x(t+dt) - x(t)}{dt} \approx f(x(t)). \quad (9.6)$$

Solving for $x(t+dt)$ and replacing the approximation sign with an equals sign give us

$$x(t+dt) = x(t) + dt \cdot f(x(t)), \quad (9.7)$$

which is a formula to compute x at time $t+dt$ from knowledge of x at time t and the function in our ODE. Hence, if we know x at some initial time t_0 has initial value x_0 , we can place (9.7) in a `for loop` and compute x at future values of t , namely $t+dt$, $t+2dt$, etc. The value of x at its initial time is called an **initial condition**.

Euler's Method

Consider an ODE $\dot{x}(t) = f(x(t))$ and define a time vector $t = [t_0, t_1, t_2, \dots, t_f]$, where $t_{k+1} = t_k + dt$, for $k = 0, 1, \dots, N$, and t_f is the final time. Then a numerical approximation to the solution of the ODE for $x(t_0) = x_0$ at times $t_0 \leq t_k \leq t_f$ can be computed by a simple `for loop`

```
for k = 0 : N - 1
    xk+1 = xk + dt · f(xk)
    tk+1 = tk + dt
end
```

In Julia, it looks like this

```
1 dimX = length(x0)
2 N = 1 + floor(Int64, tf/dt) # Number of time steps between t0 and tf
3 x = zeros(dimX, N); t = zeros(1, N)
4 x[:, 1] = x0
5 t[1] = t0
6 for k = 1:(N-1) # start at 1 because arrays in Julia are one-indexed
7     #xkp1 = xk + dt*f(xk)
8     x[:, k+1] = x[:, k] + dt*f(x[:, k])
9     t[k+1] = t[k] + dt
10 end
```

¹At Michigan, that would be Math 216.

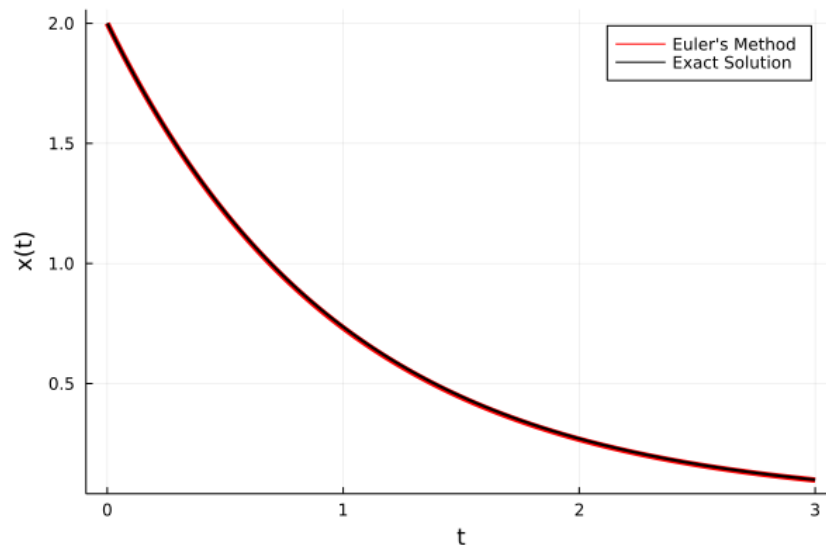
As an illustration, let's use Euler's method to solve the ODE

$$\frac{dx(t)}{dt} = -x(t), \quad (9.8)$$

with initial value $x_0 = 2$ at time $t_0 = 0$. We will compare our numerical solution with the known exact solution, $x(t) = e^{-t}x_0$, for two value of dt .

```
1 # first order linear differential equation
2 # dx/dt = - x
3 function SimpleOde(x)
4     return - x
5 end
6
7 x0 = 2.0; t0 = 0.0; # Initial values
8 dt = 0.01; tf=3.0 # dt and final time tf
9 N = 1 + floor(Int64, tf/dt) # want to return an integer
10 x = zeros(1,N); t = zeros(1,N)
11 x[1] = x0
12 t[1] = t0
13 for k = 1:(N-1)
14     #xkpl = xk + dt*f(xk) # Euler's method
15     x[k+1] = x[k] + dt*SimpleOde(x[k])
16     t[k+1] = t[k] + dt
17 end
18 y = x0*exp.(-t) # closed form solution from Calculus
19
20 t=t'; x=x'; y=y' #for plotting
21 p1 = plot(t,x, xlabel = "t", ylabel = "x(t)", lw=4, color = "red", labels = "Euler's Method" )
22 p1 = plot!(t, y, lw=2, color = "black", labels = "Exact Solution" )
23 png("FirstUseOfEulerMethod")
24 display(p1)
```

Output



Wow! That's incredible, our computed Euler's numerical approximation to the ODE is indistinguishable from the true solution! Yeah, but we got lucky. For more complicated ODEs, you should really use better approximations than Euler's Method, but hey, this is ROB 101 and we just want to learn the basics.

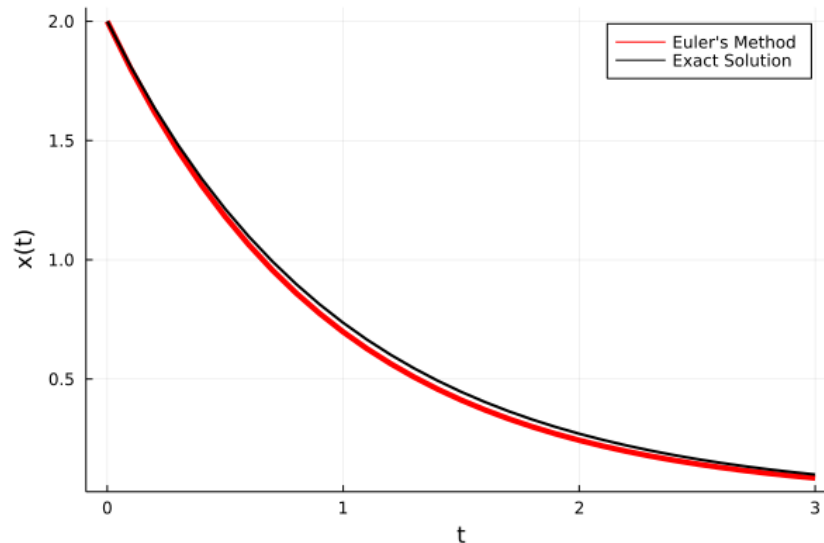
In the above plot, we set $dt = 0.01$ seconds, or 10 ms. To see what we mean by getting lucky, let's increase the value of dt by a factor of ten, namely, $dt = 0.1$ seconds. The resulting plot is still pretty satisfying, because we are still close to the true answer. The challenge is that in real engineering problems, we do not know the "true answer" and we're using numerical methods to "approximate it" without having a closed-form answer against which to check how good of a job we are doing. Here we still did OK.

```

1 # first order linear differential equation
2 # dx/dt = - x
3 function SimpleOde(x)
4     return - x
5 end
6
7 x0 = 2.0; t0 = 0.0; dt = 0.1; tf=3.0
8 N = 1 + floor(Int64, tf/dt) # want to return an integer
9 x = zeros(1,N); t = zeros(1,N)
10 x[1] = x0
11 t[1] = t0
12 for k = 1:(N-1)
13     #xkp1 = xk + dt*f(xk)
14     x[k+1] = x[k] + dt*SimpleOde(x[k])
15     t[k+1] = t[k] + dt
16 end
17 y = x0*exp.(-t) # closed form solution
18
19 t=t'; x=x'; y=y' #for plotting
20 p1 = plot(t,x, xlabel = "t", ylabel = "x(t)", lw=4, color = "red", labels = "Euler's Method" )
21 p1 = plot!(t, y, lw=2, color = "black", labels = "Exact Solution" )
22 png("UseOfEulerMethodLarge_dt")
23 display(p1)

```

Output



The above shows that large values of dt lead to errors even for simple ODEs. You might think that by taking dt sufficiently small, you can always find a good approximation. That's not totally false, but (a) computing the solution will take a long time, and even more importantly, you can run into other "numerical problems". To learn more, you need to take the ODE course!

Next we compute a solution to a well-known linear oscillator,

$$\begin{aligned}
 \dot{x}_1 &= x_2 \\
 \dot{x}_2 &= -\pi^2 x_1,
 \end{aligned} \tag{9.9}$$

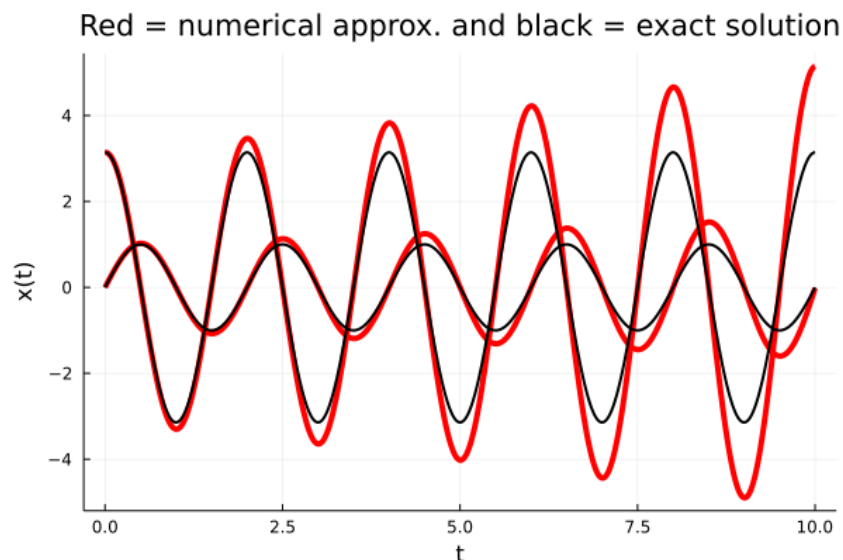
where Euler's method does a great job for several cycles, but then starts diverging from the true solution. We emphasize that more sophisticated numerical solution schemes for ODEs are well known and are routinely used in practice. Here, in ROB 101, we are just getting a taste for what is an ODE and how can one even conceive of computing solutions.

```

1 # linear differential equation
2 # where x is a vector
3 #
4 # dx/dt = [x2; -pi^2 * x1]
5 function LessSimpleOde(x)
6     return [x[2]; - pi^2*x[1]]
7 end
8
9 x0 = [0;pi]; t0 = 0.0; dt = 0.01; tf=10
10 dimX = length(x0)
11 N = 1 + floor(Int64,tf/dt) # want to return an integer
12 x = zeros(dimX,N); t = zeros(1,N)
13 x[:,1] = x0
14 t[1] = t0
15 for k = 1:(N-1)
16     #xkp1 = xk + dt*f(xk)
17     x[:,k+1] = x[:,k] + dt*LessSimpleOde(x[:,k])
18     t[k+1] = t[k] + dt
19 end
20 y = [sin.(pi*t); pi*cos.(pi*t)] # closed form solution
21 #y = [cos.(pi*t);-pi*sin.(pi*t)] # closed form solution for x0=[1;0]
22
23 titre = "Red = numerical approx. and black = exact solution"
24 t=t'; x=x'; y = y' # for plotting purposes
25
26 p1 = plot(t, x, xlabel = "t", ylabel = "x(t)", lw=4, color = "red", title = titre, legend = false)
27 p1 = plot!(t, y, lw=2, color = "black")
28
29 png("UseOfEulerMethodSinusoid")
30 display(p1)

```

Output



In the above solution, we took the initial condition as $t_0 = 0$ and

$$x_0 = \begin{bmatrix} 0 \\ \pi \end{bmatrix}.$$

For those who have taken Calc I, you can check that for this initial condition,

$$x(t) = \begin{bmatrix} x_1(t) \\ x_2(t) \end{bmatrix} = \begin{bmatrix} \sin(\pi t) \\ \pi \cos(\pi t) \end{bmatrix}$$

satisfies the initial condition $x(t_0) = x(0) = x_0 = [0 \ \pi]^\top$ and

$$\begin{aligned} \frac{d}{dt}x_1(t) &= \frac{d}{dt}\sin(\pi t) = \pi \cos(\pi t) = x_2(t) \\ \frac{d}{dt}x_2(t) &= \frac{d}{dt}\pi \cos(\pi t) = -\pi^2 \sin(\pi t) = -\pi^2 x_1(t). \end{aligned}$$

This is what it means to be a solution to the ODE. Specifically, $x(t_0) = x_0$ and $\dot{x}(t) = f(x(t))$. If instead you take the initial condition as

$$x_0 = \begin{bmatrix} 1 \\ 0 \end{bmatrix},$$

then $x_1(t) = \cos(\pi t)$ and $x_2(t) = -\pi \sin(\pi t)$ are the solution to the ODE.

Returning to the importance of numerical solutions of ODES, our point would be that no one knows a closed-form solution to the ODE

$$\dot{x}(t) = x_1(t) \cdot \cos(e^{x_1(t)}) - (x(t))^5, x(0) = -\sqrt{2},$$

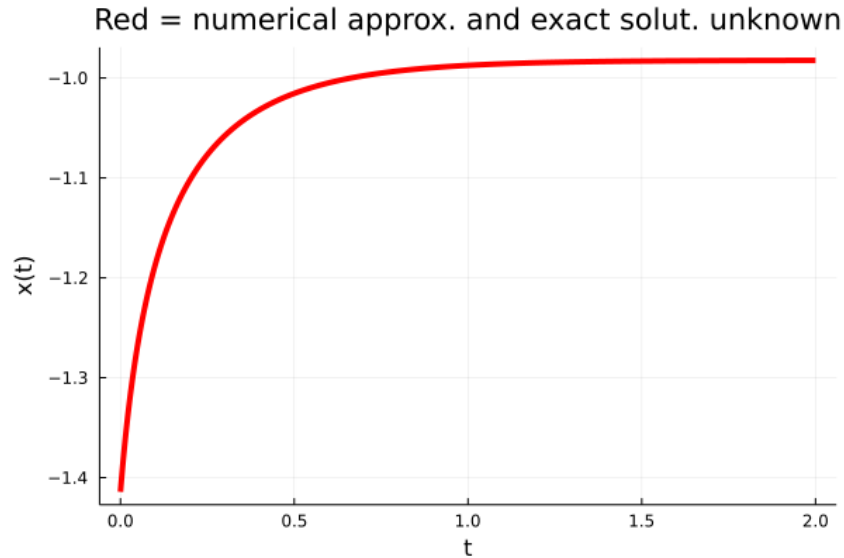
and yet we can compute it numerically.

```

1 # nonlinear differential equation
2 #
3 #
4 # dx/dt = x * cos(exp(x)) - x^5
5 function NonlinearOde(x)
6     return x * cos(exp(x)) - x^5
7 end
8
9 x0 = -sqrt(2); t0 = 0.0; dt = 0.001; tf=2
10 N = 1 + floor(Int64, tf/dt) # want to return an integer
11 x = zeros(1, N); t = zeros(1, N)
12 x[1] = x0
13 t[1] = t0
14 for k = 1:(N-1)
15     #xkpl = xk + dt*f(xk)
16     x[k+1] = x[k] + dt*NonlinearOde(x[k])
17     t[k+1] = t[k] + dt
18 end
19
20 titre = "Red = numerical approx. and exact solut. unknown"
21 t=t'; x=x' # for plotting
22 p1 = plot(t, x, xlabel = "t", ylabel = "x(t)", lw=4, color = "red", title = titre, legend = false)
23 png("UseOfEulerMethodNonlinear")
24 display(p1)

```

Output



9.4 Discrete-time Approximations of Continuous-time Linear ODEs with Control Variables

A linear ODE with control variables is typically written as

$$\dot{x}(t) = A_c x(t) + B_c u(t), \quad (9.10)$$

where $t \geq t_0$ represents time, $x(t) \in \mathbb{R}^n$ is the state vector, and $u(t) \in \mathbb{R}^m$ is a vector of control inputs, that is, the variables you can change by commanding motor torques, for example. From this information, you can deduce that A_c is $n \times n$ and B_c is $n \times m$. The initial value of the state is denoted $x_0 = x(t_0)$. We'll discuss how to make intelligent choices for the control signal in the next section.

To obtain a **discrete-time approximation** of the ODE (9.10), we use Euler's method and replace the time derivative with an approximation,

$$\frac{x(t+dt) - x(t)}{dt} = A_c x(t) + B_c u(t) \iff x(t+dt) = x(t) + dt \cdot (A_c x(t) + B_c u(t)). \quad (9.11)$$

If for $k = 0, 1, 2, \dots$ we define $t_k := t_0 + kdt$, $x_k := x(t_k)$, $u_k := u(t_k)$, we obtain

$$\begin{aligned} x_{k+1} &= x_k + dt \cdot A_c x_k + dt \cdot B_c u_k \\ x_{k+1} &= \underbrace{(I + dt \cdot A_c)}_A x_k + \underbrace{dt \cdot B_c}_B u_k \\ x_{k+1} &= Ax_k + Bu_k, \end{aligned} \quad (9.12)$$

where, $A := I + dt \cdot A_c$ and $B := dt \cdot B_c$ are $n \times n$ and $n \times m$, respectively.

Euler's Method takes us from a Linear ODE to Iterating Linear Equations

A typical linear ODE looks like this.

$$\dot{x}(t) = A_c x(t) + B_c u(t).$$

With Euler's Method, we quickly approximate $\dot{x}(t)$ via

$$\frac{x(t+dt) - x(t)}{dt} = A_c x(t) + B_c u(t).$$

When for $k = 0, 1, 2, \dots$ we define $t_k := t_0 + kdt$, $x_k := x(t_k)$, $u_k := u(t_k)$, we obtain

$$x_{k+1} = Ax_k + Bu_k,$$

where $A := I + dt \cdot A_c$ and $B := dt \cdot B_c$, which we can solve with a `for loop`.

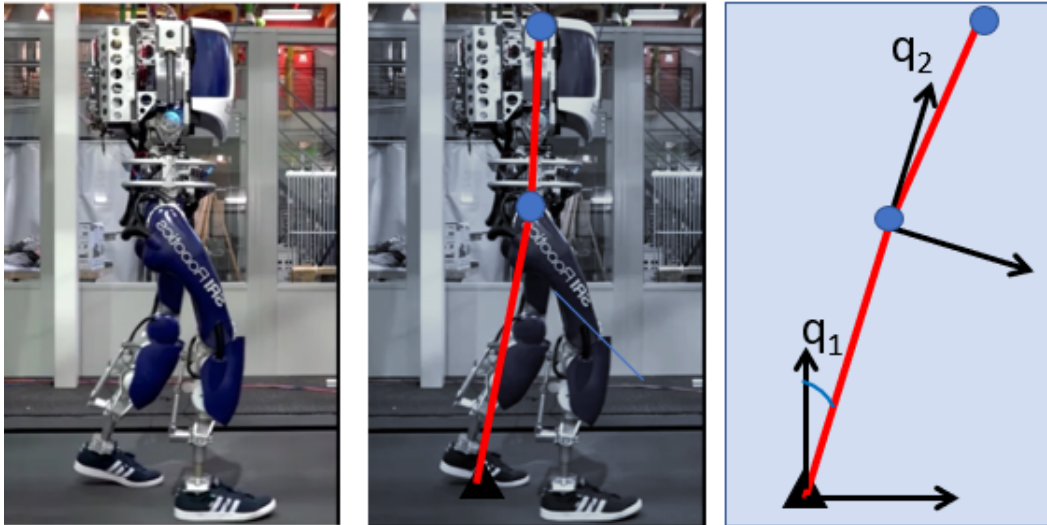


Figure 9.2: Left is the bipedal robot, Durus, from Prof. Ames' group at Caltech. Center is a crude pendulum approximation of the robot. Right is a double inverted pendulum that we will balance in this lab! We assume the bottom link is actuated via a motor, while the middle link is a frictionless revolute joint. All angles are positive when measured clockwise from the vertical axis. A linear approximation of the double inverted pendulum's ODE model is given in (9.13).

Why is Euler's method important? A scary ODE now looks like one of our iterative algorithms, say the Bisection Method, Newton's Algorithm, or Gradient Descent. Once we convert the ODE (9.10) into a discrete-time model (9.12), we can compute solutions just by iterating with a `for loop`, which is quite remarkable. We'll illustrate this on a **double inverted pendulum** that approximates a legged robot; see Fig. 9.2. Here is a linear ODE model that is valid for small deviations of the angles from zero, where $x_1 = q_1$ and $x_2 = q_2$ are the two angles of the pendula in radians and $x_3 = \dot{q}_1$ and $x_4 = \dot{q}_2$ are the angular rates in radians per second.

$$\underbrace{\begin{bmatrix} \dot{x}_1(t) \\ \dot{x}_2(t) \\ \dot{x}_3(t) \\ \dot{x}_4(t) \end{bmatrix}}_{\dot{x}(t)} = \underbrace{\begin{bmatrix} 0.0000 & 0.0000 & 1.0000 & 0.0000 \\ 0.0000 & 0.0000 & 0.0000 & 1.0000 \\ 39.2400 & -31.3920 & 0.0000 & 0.0000 \\ -78.4800 & 133.4160 & 0.0000 & 0.0000 \end{bmatrix}}_{A_c} \underbrace{\begin{bmatrix} x_1(t) \\ x_2(t) \\ x_3(t) \\ x_4(t) \end{bmatrix}}_{x(t)} + \underbrace{\begin{bmatrix} 0.0000 \\ 0.0000 \\ 3.2000 \\ -9.6000 \end{bmatrix}}_{B_c} u(t). \quad (9.13)$$

If we take $dt = 0.1$, then the associated discrete-time model becomes

$$\underbrace{\begin{bmatrix} x_{1,k+1} \\ x_{2,k+1} \\ x_{3,k+1} \\ x_{4,k+1} \end{bmatrix}}_{x_{k+1}} = \underbrace{\begin{bmatrix} 1.0000 & 0.0000 & 0.1000 & 0.0000 \\ 0.0000 & 1.0000 & 0.0000 & 0.1000 \\ 3.9240 & -3.1392 & 1.0000 & 0.0000 \\ -7.8480 & 13.3416 & 0.0000 & 1.0000 \end{bmatrix}}_A \underbrace{\begin{bmatrix} x_{1,k} \\ x_{2,k} \\ x_{3,k} \\ x_{4,k} \end{bmatrix}}_{x_k} + \underbrace{\begin{bmatrix} 0.0000 \\ 0.0000 \\ 0.3200 \\ -0.9600 \end{bmatrix}}_B u_k. \quad (9.14)$$

```

1 using LinearAlgebra
2 # Linear Double Inverted Pendulum Model
3 F=dyn_mod_2LinkPendulum([0;0],[0;0])
4 A11=zeros(2,2); A22=zeros(2,2)
5 A12=zeros(2,2)+I; A21=-inv(F.D)*F.JacG
6 #
7 # Continuous-time terms in the ODE model
8 Ac = [A11 A12; A21 A22]
9 Bc = [zeros(2,1); inv(F.D)*F.B]
10
11 # Discrete time model
12 dt = 0.1
13 A = I + dt*Ac
14 B = dt*Bc;

```

We now compute the evolution of the double inverted pendulum when the initial condition does not correspond to the upright equilibrium position and the motor torque is identically zero.

```

1 # Run me, don't change me
2 x0=[pi/16; -pi/16; 0; 0]
3 t0 = 0.0; tf=1
4 N = 1 + floor(Int64, tf/dt) # want to return an integer
5 x = zeros(length(x0), N); t = zeros(1, N)
6 x[:, 1] = x0
7 t[1] = t0
8 for k = 1:(N-1)
9     #xkp1 = xk + dt*f(xk)
10    x[:, k+1] = A*x[:, k]
11    t[k+1] = t[k] + dt
12 end

```

Output

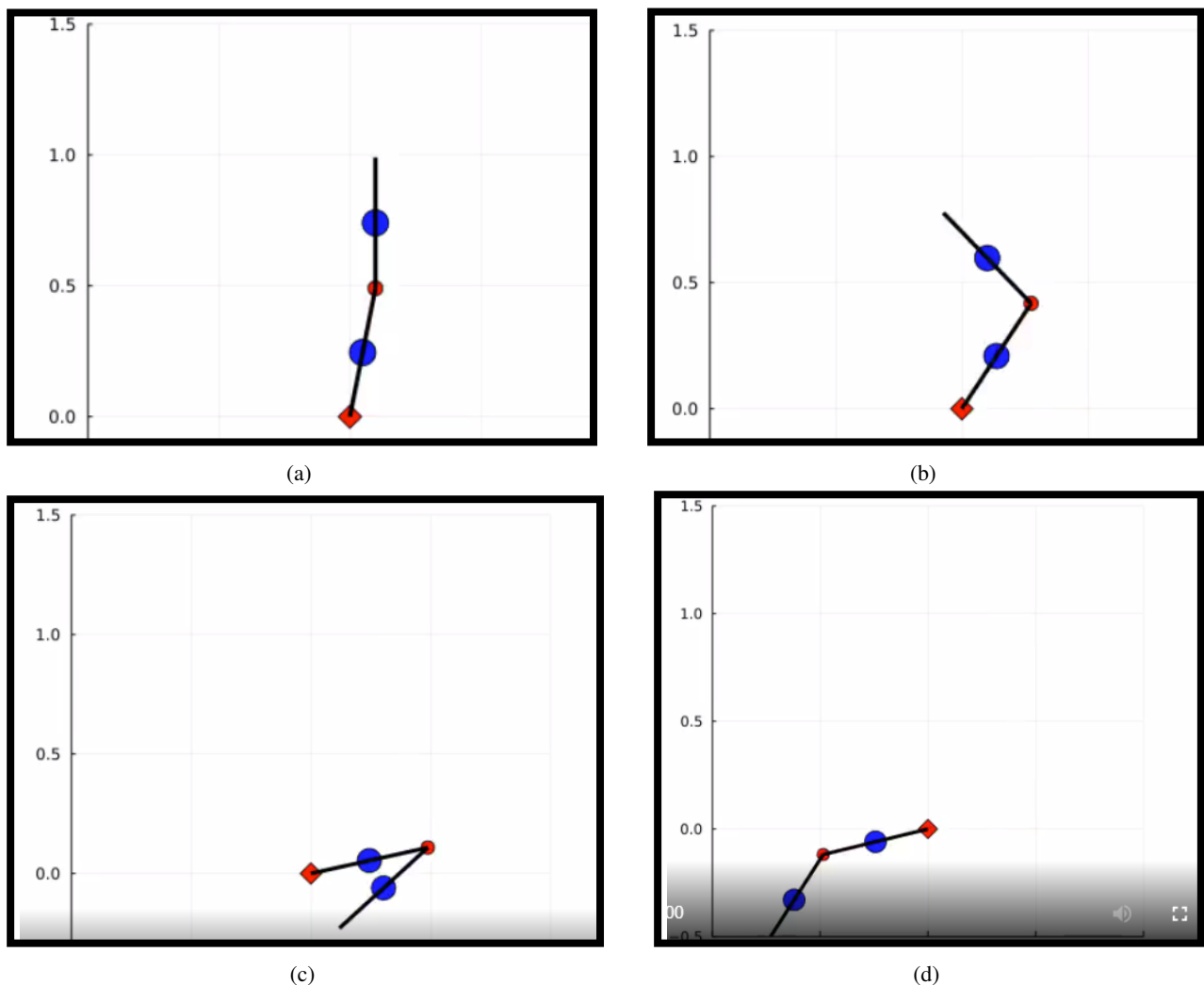


Figure 9.3: Evolution of the double inverted pendulum when $u_k \equiv 0.0$. The images are the pendulum's configurations at $t_0 = 0.0$, $t_1 = 0.1$, $t_2 = 0.2$ and $t_3 = 0.3$ seconds. Without the proper inputs, the pendulum falls over. Are we surprised?

Just for fun, we give the nonlinear model of the double inverted pendulum. Your author generated it using Lagrangian mechanics, a technique that is taught in advanced Physics courses or advanced ME courses on dynamics.

```

1 function    dyn_mod_2LinkPendulum (q, dq)
2 #DYN_MOD_2LINKPENDULUM
3 #20-Jun-2022 12:32:57
4 #
5 # Author: Grizzle
6 #
7 #
8 # Model NOTATION: Spong and Vidyasagar, page 142, Eq. (6.3.12)
9 #            $D(q)*ddq + C(q,dq)*dq + G(q) = B*tau$ 
10 #
11 #
12 g, L1, L2, m1, m2=modelParameters();
13 #
14 #
15 #
16 #
17 q1=q[1]; q2=q[2];
18 dq1=dq[1]; dq2=dq[2];
19 #
20 #
21 #
22 #
23 D=zeros(2,2);
24 D[1,1]=(m2*(8*L1^2 + 2*L2^2 + 8*L1*L2*cos(q2)))/8 + (L1^2*m1)/4;
25 D[1,2]=(L2*m2*(L2 + 2*L1*cos(q2)))/4;
26 D[2,1]=(L2*m2*(L2 + 2*L1*cos(q2)))/4;
27 D[2,2]=(L2^2*m2)/4;
28 #
29 #
30 #
31 #
32 C=zeros(2,2);
33 C[1,1]=-(L1*L2*dq2*m2*sin(q2))/2;
34 C[1,2]=-(L1*L2*m2*sin(q2)*(dq1 + dq2))/2;
35 C[2,1]=(L1*L2*dq1*m2*sin(q2))/2;
36 #
37 #
38 #
39 #
40 G=zeros(2,1);
41 G[1]=-(g*m2*((L2*sin(q1 + q2))/2 + L1*sin(q1)) - (L1*g*m1*sin(q1)))/2;
42 G[2]=-(L2*g*m2*sin(q1 + q2))/2;
43 #
44 #
45 #
46 #
47 B=zeros(2,1);
48 B[1,1]=1;
49 #
50 #
51 JacG=zeros(2,2);
52 JacG[1,1]=-(g*m2*((L2*cos(q1 + q2))/2 + L1*cos(q1)) - (L1*g*m1*cos(q1)))/2;
53 JacG[1,2]=-(L2*g*m2*cos(q1 + q2))/2;
54 JacG[2,1]=-(L2*g*m2*cos(q1 + q2))/2;
55 JacG[2,2]=-(L2*g*m2*cos(q1 + q2))/2;
56 #

```

```

57 #
58 return (D=D, C=C, G=G, B=B, JacG=JacG)
59 end

```

9.5 Predicting the State Vector at Future Instants of Time

Now that we have a means to compute the state at the next time instance based on the current state and current input, we can also iterate forward and **predict the state at some future time**, say N steps ahead, as a function of a hypothesized input sequence, $\{u_0, u_1, \dots, u_{N-1}\}$, as follows

$$\begin{aligned}
x_0 &= \text{given} \\
x_1 &= Ax_0 + Bu_0 \\
x_2 &= Ax_1 + Bu_1 = A(Ax_0 + Bu_0) + Bu_1 = A^2x_0 + ABu_0 + Bu_1 \\
x_3 &= Ax_2 + Bu_2 = A(A^2x_0 + ABu_0 + Bu_1) + Bu_2 = A^3x_0 + A^2Bu_0 + ABu_1 + Bu_2 \\
&\vdots \\
x_N &= A^N x_0 + A^{N-1}Bu_0 + A^{N-2}Bu_1 + \dots + ABu_{N-2} + Bu_{N-1}
\end{aligned} \tag{9.15}$$

We now focus on that last line of (9.15). We will see that it is a gold mine. We note that

$$x_N = A^N x_0 + A^{N-1} \cdot Bu_0 + A^{N-2} \cdot Bu_1 + \dots + A \cdot Bu_{N-2} + Bu_{N-1} \tag{9.16}$$

$$x_N = \underbrace{A^N}_{S} x_0 + \underbrace{[A^{N-1} \cdot B \quad A^{N-2} \cdot B \quad \dots \quad A \cdot B \quad B]}_M \cdot \underbrace{\begin{bmatrix} u_0 \\ u_1 \\ \vdots \\ u_{N-2} \\ u_{N-1} \end{bmatrix}}_{u_{\text{seq}}}, \tag{9.17}$$

where for compactness of notation, we define

$$S := A^N \tag{9.18}$$

$$M := [A^{N-1} \cdot B \quad A^{N-2} \cdot B \quad \dots \quad A \cdot B \quad B], \text{ and} \tag{9.19}$$

$$u_{\text{seq}} := \begin{bmatrix} u_0 \\ u_1 \\ \vdots \\ u_{N-2} \\ u_{N-1} \end{bmatrix}. \tag{9.20}$$

This leads to our most important equations so far:

$$\begin{array}{c}
x_N = Sx_0 + Mu_{\text{seq}} \\
\Downarrow \\
x_N - Sx_0 = Mu_{\text{seq}} \\
\Downarrow \\
Mu_{\text{seq}} = (x_N - Sx_0).
\end{array} \tag{9.21}$$

If the rows of M are linearly independent, then we can solve for a vector u_{seq} of minimum norm that satisfies (9.21) using our knowledge of underdetermined linear equations. We illustrate this on our discrete-time model of the double inverted pendulum in (9.14).

```

1 N=8
2 x0=[pi/16; -pi/16; 0; 0]

```

```

3 t0=0
4
5 # Predict ahead
6 S=A^N
7 M=B
8 for k = 1:N-1
9     M=[A*M B]
10 end
11
12 xGoal = [0.;0.;0.;0.]
13 # Solve for input such that x[N]=xGoal using least squares
14 if true
15     # Cheating method to solve least squares..we use the inverse function :( oh no....
16     useq = (M') * inv(M*(M')) * (xGoal-S*x0)
17 else # better way to do it
18     useq = minNormUnderdetermined(M, xGoal-S*x0)
19 end
20 # Verify we have a good solution
21 @show xGoal
22 @show xFinal=S*x0+M*useq
23 #
24 # Now, solve for x[k], 0 <= k <= N
25 x = zeros(length(x0),N+1); t = zeros(1,N+1)
26 x[:,1]=x0
27 for k = 1:N
28     x[:,k+1] = A*x[:,k] + B*useq[k]
29     t[k+1] = t[k] + dt
30 end
31 @show x[:,end]
32 animateInvPendulum(x)

```

Output

```

xGoal = [0.0, 0.0, 0.0, 0.0]
xFinal = S * x0 + M * useq =
[6.481984013362307e-5, -0.00020456325910345186, 0.0014950509685149882, -0.004721057710412424]
x[:,end] = [6.481984022717204e-5, -0.00020456325908854315, 0.0014950509688680391,
-0.004721057705593168]

```

Snapshots of the pendulum's evolution are given in Fig. 9.4.

Remark: The least squares solution to underdetermined equations is studied in Chapter 9 of our textbook. The key results are as follows: Consider an underdetermined system of linear equations $Mu_{\text{seq}} = x_{\text{Goal}} - Sx_k$. If the rows of M are linearly independent (equivalently, the columns of M^\top are linearly independent), then

$$u_{\text{seq}}^* = \arg \min_{Mu_{\text{seq}}=(x_{\text{Goal}}-Sx_k)} \|u_{\text{seq}}\|^2 \iff u_{\text{seq}}^* = M^\top \alpha \text{ and } M \cdot M^\top \alpha = (x_{\text{Goal}} - Sx_k). \quad (9.22)$$

We do the QR Factorization of M^\top instead of M itself, so that

$$M^\top = Q \cdot R.$$

Then,

$$u_{\text{seq}}^* = \arg \min_{Mu_{\text{seq}}=(x_{\text{Goal}}-Sx_k)} \|u_{\text{seq}}\|^2 \iff u_{\text{seq}}^* = Q\beta \text{ and } R^\top \beta = (x_{\text{Goal}} - Sx_k). \quad (9.23)$$

We note that R^\top is lower triangular, and thus $R^\top \beta = (x_{\text{Goal}} - Sx_k)$ can be solved via forward substitution. See Chapter 9 for more details.

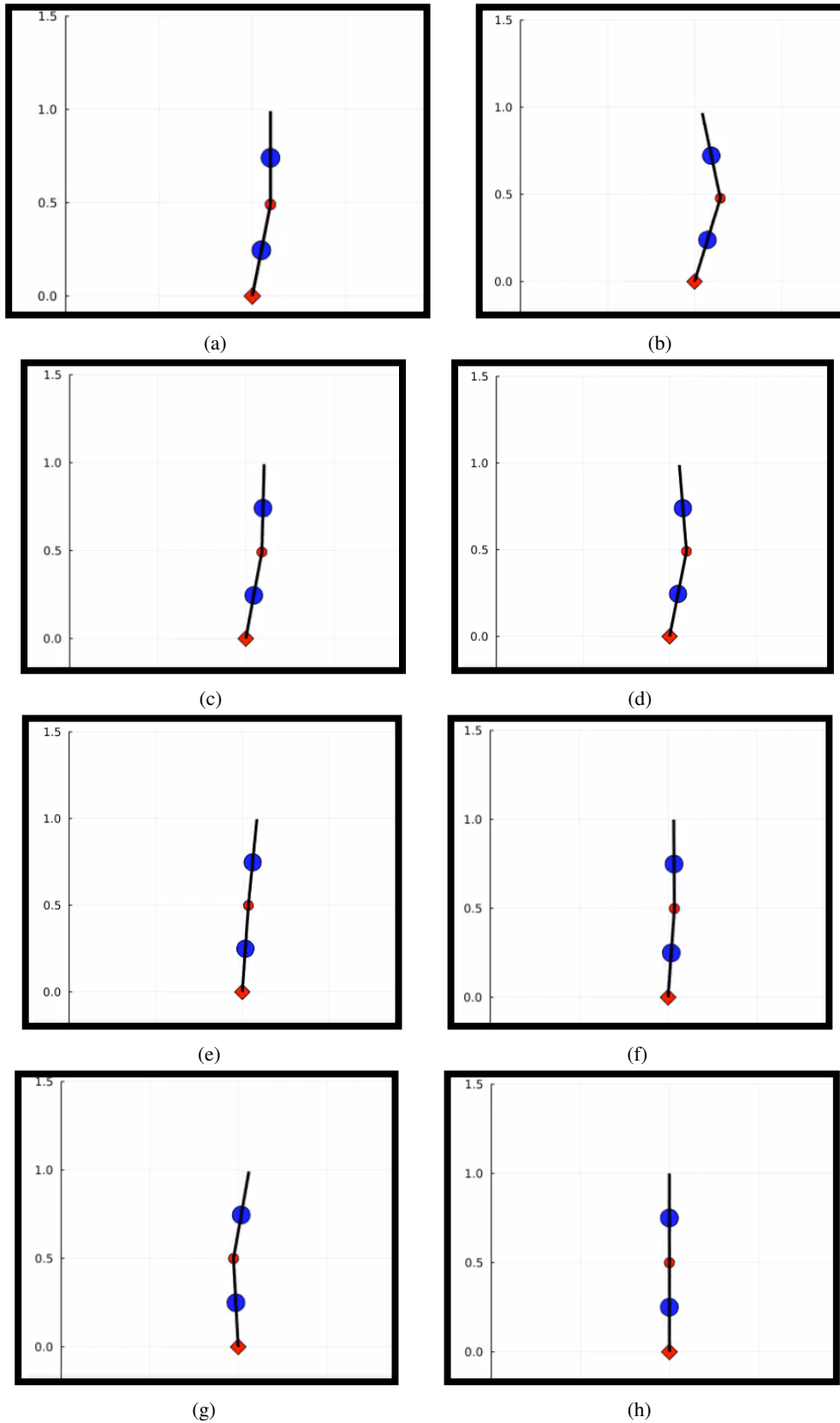


Figure 9.4: Evolution of the double inverted pendulum when the sequence u_0, \dots, u_7 is chosen to drive the pendulum to the upright position. The images are the pendula's configurations at $t_0 = 0.0, t_1 = 0.1, t_2 = 0.2, \dots, t_7 = 0.7$ seconds. With the proper inputs, the pendulum balances nicely. Are we happy?

```

1 # The QR pipeline for minimum norm solution of Ax=b when
2 # the ROWS of A are linearly independent
3 # (yes, the columns of A' are linearly independent)
4
5 function forwardsub(L, b)
6     (nr, nc) = size(L)
7     x = Vector{Float64}(undef, nc)
8     if minimum(abs.(diag(L))) < 1e-8
9         println("L is close to singular. I will not solve this problem")
10        return x
11    else
12        x[1] = b[1] / L[1,1]
13        for i = 2:nc
14            x[i] = ( b[i] - L[i,1:i-1]' * x[1:i-1] ) ./ L[i,i]
15        end
16        return x
17    end
18 end
19
20 # Call me with A = M and b = (xGoal - Sxk)
21 function minNormUnderdetermined(A, b)
22     # Solves for minimum norm x in Ax = b
23     # using the QR factorization and forward substitution. Returns xStar.
24     # xStar = arg min x' x
25     # subject to Ax = b
26     F = qr(A')
27     Q = Matrix(F.Q)
28     R = Matrix(F.R)
29     beta = forwardsub(R', b)
30     xStar = Q*beta
31     return xStar
32 end

```

Output

minNormUnderdetermined (generic function with 1 method)

9.6 (Optional Read:) For What Values of N are the Rows of M Linearly Independent?

We write a simple script to check for the rows of $M := [A^{N-1} \cdot B \ \dots \ A \cdot B \ B]$ to be linearly independent.

```

1 for N = 1:10
2     M=B
3     for k = 1:N-1
4         M=[A*M B]
5     end
6     detMMtrans = det(M*M')
7     rankM = rank(M, rtol=1e-6)
8     println("For N = \$N, det(M*M') = \$detMMtrans, rank(M) = \$rankM")
9 end

```

Output

For N = 1, det(M*M') = 0.0, rank(M) = 1
For N = 2, det(M*M') = -5.816113682013453e-32, rank(M) = 2
For N = 3, det(M*M') = -2.294447422015386e-17, rank(M) = 3

For $N = 4$, $\det(M \cdot M') = 0.001629280421367469$, $\text{rank}(M) = 4$
 For $N = 5$, $\det(M \cdot M') = 0.13025026505817638$, $\text{rank}(M) = 4$
 For $N = 6$, $\det(M \cdot M') = 9.088750148559221$, $\text{rank}(M) = 4$
 For $N = 7$, $\det(M \cdot M') = 431.06039658516085$, $\text{rank}(M) = 4$
 For $N = 8$, $\det(M \cdot M') = 19417.82334475286$, $\text{rank}(M) = 4$
 For $N = 9$, $\det(M \cdot M') = 772663.0420936232$, $\text{rank}(M) = 3$
 For $N = 10$, $\det(M \cdot M') = 3.0107394769364707e7$, $\text{rank}(M) = 3$

The “theoretically correct” result is that for all $N \geq 1$, $\text{rank}(M) = \min\{N, 4\}$. As we see, the “numerical rank” of M “deteriorates” for $N > 8$. Hence, our value of $N = 8$ used for Fig. 9.4 now makes more sense.

9.7 Feedback Control of Discrete-time Linear Models via Least Squares for Underdetermined Systems of Equations

In this section, we get very close to real Robotics.

9.7.1 Getting the Pendulum Upright is not Enough

To get started, we need to observe that the final value of the state x in Fig. 9.4 is nearly zero, but is not exactly equal to zero,

$$x_{\text{final}} \approx [6.48e-5, -0.0002, 0.001495, -0.0047].$$

Since the pendulum is not exactly at the equilibrium, it will fall once we are no longer providing inputs that strive to keep it upright, as the next code block verifies. If we continue computing $x(kdt)$ for $8 < k \leq 16$ with $u_k = 0$, we end up with $x(16dt) \approx [0.94, -2.97, 21.72, -68.59]$. Ooops! The pendulum falls.

Even if we had achieved $x_{\text{final}} \equiv [0.0, 0.0, 0.0, 0.0]$, and thus the pendula are in perfect equilibrium, would that be enough? What would happen if a slight gust of air passed over the device? Or a slight vibration of the platform on which the device is setting caused it to wiggle just a tiny bit? Or what if the parameters in the model that were used for computing u_{seq} were just a bit different than the real device? In all of these cases, the pendula would fall.

```

1 # Here we simulate for longer than we control
2 N=8 # Planning horizon
3 x0=[pi/16; -pi/16; 0; 0]
4 t0=0
5 Nsim = 16 # Simulation duration
6 # Predict ahead
7 S=A^N
8 M=B
9 for k = 1:N-1
10     M=[A*M B]
11 end
12
13 xGoal = [0.;0.;0.;0.]
14 # Solve for input such that x[N]=xGoal using least squares
15 if true
16     # Cheating method to solve least squares..we use the inverse function :( oh no....
17     useq = (M') * inv(M*(M')) * (xGoal-S*x0)
18 else # better way to do it
19     useq = minNormUnderdetermined(M, xGoal-S*x0)
20 end
21 # Verify we have a good solution
22 @show xGoal
23 @show xFinal=S*x0+M*useq
24 #
25 # Now, solve for x[k], 0 <= k <= N

```

```

26 x = zeros (length (x0) , Nsim+1) ; t = zeros (1, Nsim+1)
27 x[:, 1]=x0
28
29 for k = 1:Nsim
30     if k > N
31         x[:, k+1] = A*x[:, k] # uk = 0
32     else
33         x[:, k+1] = A*x[:, k] + B*useq[k]
34     end
35     t[k+1] = t[k] + dt
36 end
37 @show x[:, end]
38 animateInvPendulum (x)

```

Output

```

xGoal = [0.0, 0.0, 0.0, 0.0]
xFinal = S * x0 + M * useq =
[6.481984013362307e-5, -0.00020456325910345186, 0.0014950509685149882,
-0.004721057710412424]
[N Nsim] = [8 16]
x[:, end] = [0.93888952433862, -2.965304039678654, 21.715899365969108, -68.5855850968567]

```

9.7.2 Keeping the Pendulum Upright is Our Goal

Let's summarize what we have learned so far:

- If we choose a fixed number of time steps, N_{horiz} , large enough so that the rows of M are linearly independent, we know theoretically how to compute a sequence of control values

$$u_{\text{seq}} = u_0, u_1, \dots, u_{N_{\text{horiz}}-1}$$

that drives the pendulum from an initial condition x_0 to a goal state that we are choosing as the upright equilibrium for the double inverted pendulum.

- In our numerical simulations, even for the linear model, our calculations have a small amount of numerical error so that u_{seq} does not drive the double inverted pendulum to the exact equilibrium. We get close, but if we continue the computation of the solution of the double inverted pendulum with zero as the control input, it tips over and falls. The same thing would happen on a real robot, by the way, so our issue is not limited to numerical simulations.
- We chose $N_{\text{horiz}} = 8$. One possibility is to make it bigger, say 16, or 20, or why not 100? Well, no matter how big we make it, the pendulum will fall over when our control sequence ends.
- Also, our control sequence is computed on the basis of our linear model, which we are assuming to be correct. If the pendula were perturbed by a person, or by wind, or by friction in the joints, then our predicted behavior from the model would be incorrect. You can check that these scenarios once again lead to the pendula falling.
- If you are thinking that well, we could measure $x_{N_{\text{horiz}}}$, compute a new control sequence u_{seq} , apply it for the next N_{horiz} time steps, and repeat this process, then you have just invented what is called **Model Predictive Control or MPC for short!**

From x_k to x_{k+N} as a function of the control sequence.

What happens if you know the state at time k and want to predict its value at some future time $k + N$? It works like this

$$x_{k+N} = Sx_k + Mu_{\text{seq}}, \quad (9.24)$$

where this time,

$$u_{\text{seq}} := \begin{bmatrix} u_k \\ u_{k+1} \\ \vdots \\ u_{k+N-2} \\ u_{k+N-1} \end{bmatrix},$$

while S and M are still given by (9.18) and (9.19). **In other words, the formula does not change.**

The fix for the falling pendulum is to compute new controls that drive the pendulum to its goal state, the upright equilibrium.

In the modern version of Model Predictive Control, aka MPC, we do not wait the full N_{horiz} time steps before computing a new control sequence, u_{seq} . Instead, **at each instant of time kdt** , we measure the current state x_k of the double pendulum, compute a new sequence of control commands, $u_{\text{seq}} = [u_k, u_{k+1}, \dots, u_{k+N_{\text{horiz}}-1}]$, for the given horizon length, N_{horiz} , that steers the double inverted pendulum to its equilibrium, **apply the first value in the sequence which gets us to time step $(k + 1)dt$** , and repeat the process all over again (means, put this process in a `for loop`).

MPC: Closed-loop Control via Least-squares Optimization

Model Predictive Control works as follows.

- Set a goal, x_{Goal} .
- Set a (relatively short-term) planning horizon for control, consisting of N_{horiz} time steps;
- FOR** $k = 0 : N$ **as long as you want to balance the pendula**

- measure the system's current state, x_k ;
- using (9.23), solve

$$u_{\text{seq}}^* := \arg \min_{Mu_{\text{seq}} = (x_{\text{Goal}} - Sx_k)} \|u_{\text{seq}}\|^2,$$

where S and M are given in (9.18) and (9.19) with $N = N_{\text{horiz}}$;

- define $u_k := u_{\text{seq}}[1]$ and apply it to the actuators of your system (that is, the motor driving the base of the double inverted pendulum).
- Either let physics do its thing and compute x_{k+1} for you, or, in the case of a computer simulation, apply the control u_k to your mathematical model to compute $x_{k+1} = Ax_k + Bu_k$, and then wash, rinse, and repeat.

END

9.7.3 Relevant Code for MPC

Here we give code that supports MPC control.

```
1 function myMPC(A, B, xk, nControlHoriz, xGoal)
2     # xk is the current position and velocity of the pendulum links
3     # k is the time index. Hence, tk = k dt
4     (nr, nc) = size(A)
5     nControlHoriz = max(nControlHoriz, nr) # Cannot be smaller than nr
6     S = A^nControlHoriz
7     M = Array{Float64, 2} (undef, nr, 0)
8     for i=1:nControlHoriz
```

```

9      M=[A*M B]
10     end
11     C=zeros(nr, nr)+I
12     uControl=minNormUnderdetermined(C*M, C*(xGoal-S*xk))
13     uk=uControl[1]
14     return uk
15 end

```

Output

myMPC (generic function with 1 method)

The next code block simulates the linear system $x_{k+1} = Ax_k + Bu_k$ under MPC control.

```

1 function LinSimMPC(A, B, x0, nControlHoriz, Nsim, xGoal)
2     (rA, cA) = size(A)
3     N=1+Nsim # because x0 is included
4     xTraj = Array{Float64, 2}(undef, rA, 0)
5     uTraj = Array{Float64, 2}(undef, 1, 0)
6     xTraj=[xTraj x0]
7     for k = 1:N
8         xk=xTraj[:, k]
9         uk=myMPC(A, B, xk, nControlHoriz, xGoal)
10        uTraj=[uTraj uk]
11        xkp1 = A*xk+B*uk
12        xTraj = [xTraj xkp1]
13    end
14    return (xTraj=xTraj, uTraj=uTraj)
15 end

```

Output

LinSimMPC (generic function with 1 method)

The next code block simulates a nonlinear system $x_{k+1} = x_k + dt f(x_k, u_k)$ under MPC control.

```

1 # 2 link inverted pendulum
2 function f(x1, x2, u=[0.0])
3     F=dyn_mod_2LinkPendulum(x1, x2)
4     Kfric=0*diagm([.1; .1])
5     # Use the backslash command to solve for x2 dot
6     dx2=F.D\(-F.C*x2-F.G-Kfric*x2+F.B*u)
7     return dx2
8 end
9
10 # Apply our linear MPC solution to the real nonlinear model
11
12 function NonLinSimMPC(f, dt, A, B, x0, nControlHoriz, Nsim, xGoal)
13     rA = length(x0)
14     N=1+Nsim # because x0 is included
15     xTraj = Array{Float64, 2}(undef, rA, 0)
16     uTraj = Array{Float64, 2}(undef, 1, 0)
17     xTraj=[xTraj x0]
18     for k = 1:N
19         xk=xTraj[:, k]
20         uk=myMPC(A, B, xk, nControlHoriz, xGoal)
21         uTraj=[uTraj uk]

```

```

22     dx2 = f(xk[1:2], xk[3:4], uk)
23     xkp1 = xk + dt*[xk[3:4]; dx2]
24     xTraj = [xTraj xkp1]
25 end
26 return (xTraj=xTraj, uTraj=uTraj)
27 end

```

Output

NonLinSimMPC (generic function with 1 method)

9.7.4 Balancing the Double Inverted Pendulum with MPC

Here we run the previous code blocks for the Double Inverted Pendulum. To show that our approach is practical, we'll also implement on the "real" nonlinear model of the device!

```

1 xGoal=zeros(4,1)
2 x0=[pi/16; -pi/16; 0; 0]
3 t0=0
4 tf=3
5 @show dt
6 # dt is defined above
7 Nsim = 1 + floor(Int64, tf/dt) # want to return an integer
8 nControlHoriz=8
9 F=LinSimMPC(A, B, x0, nControlHoriz, Nsim, xGoal)
10 G=NonLinSimMPC(f, dt, A, B, x0, nControlHoriz, Nsim, xGoal);

```

Output

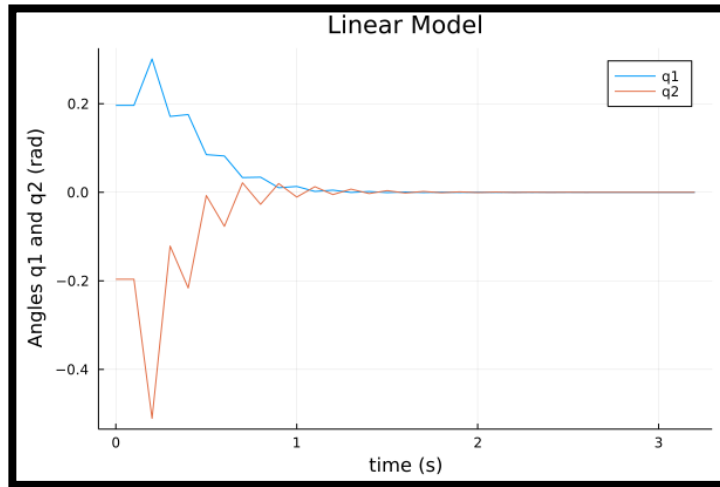
dt = 0.1

```

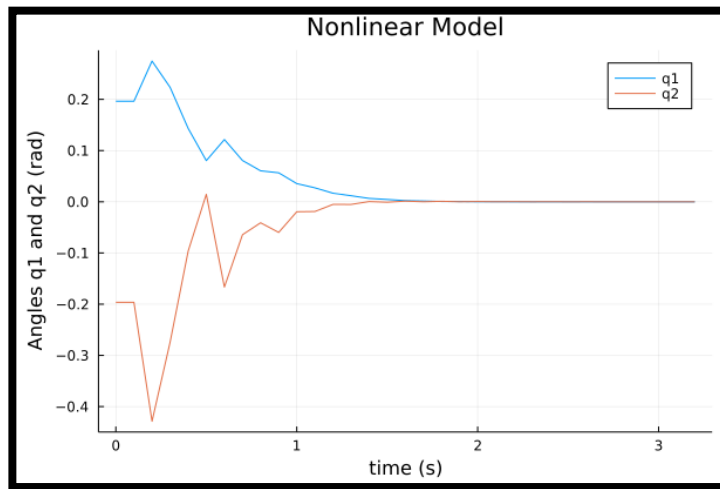
1 X=F.xTraj # Linear model
2 X=X[1:2, :]
3 t=dt*collect(0:size(X,2)-1)
4 using Plots
5 titre = "Linear Model"
6 Legend = ["q1" "q2"]
7 p1=plot(t, X', xlabel="time (s)", ylabel="Angles q1 and q2 (rad)", title=titre, label = Legend)
8 X=G.xTraj # Nonlinear model
9 X=X[1:2, :]
10 titre = "Nonlinear Model"
11 p2=plot(t, X', xlabel="time (s)", ylabel="Angles q1 and q2 (rad)", title=titre, label = Legend)
12 #
13 Ulin=F.uTraj; Unon=G.uTraj
14 U=[Ulin; Unon]
15 t=t[1:end-1]
16 p3=plot(t, U', title="Motor Torques", label=["u linear model" "u nonlinear model"])
17 display(p1)
18 display(p2)
19 display(p3)

```

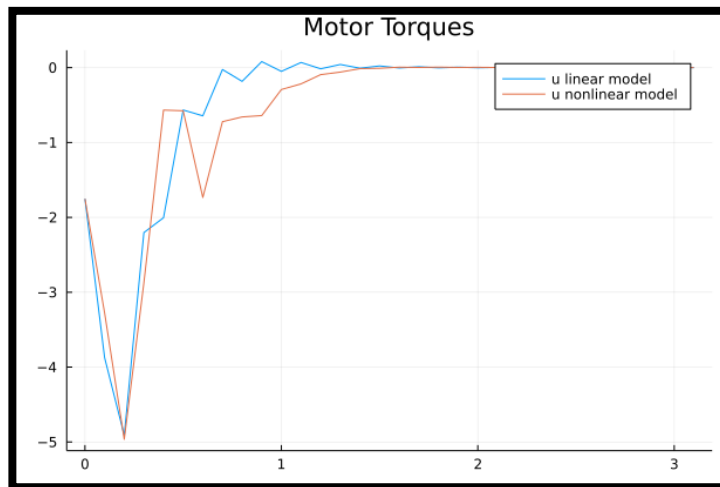
Output



(a)



(b)



(c)

Figure 9.5: Traces for the double inverted pendulum under MPC. The top plot is MPC applied to the linear model, where q_1 and q_2 converging to zero mean that they are going to the upright equilibrium. It is great but not surprising that it works. The middle plot is MPC applied to the nonlinear model, when the control is computed on the basis of a linear approximation. This is what is done on many real robots. It is awesome that this works! The bottom plot compares the torque commands for the motors in the two cases. They are remarkably similar. Larger initial deviations from the upright equilibrium can result in failure.

Chapter 10

Julia Lab 10: The Joy of Doing Calculus with Julia!

Learning Objectives

- How to efficiently compute derivatives and partial derivatives of functions.
- Remove a common source of errors in Engineering: doing Calculus by hand!

Outcomes

- Symbolic computations in Julia
- Numerical derivatives in Julia
- Automatic Differentiation, the current workhorse for Machine Learning

Either download Lab10 from our Canvas site or open up a Jupyter notebook so that you can enter code as we go. It is suggested that you have line numbering toggled on.

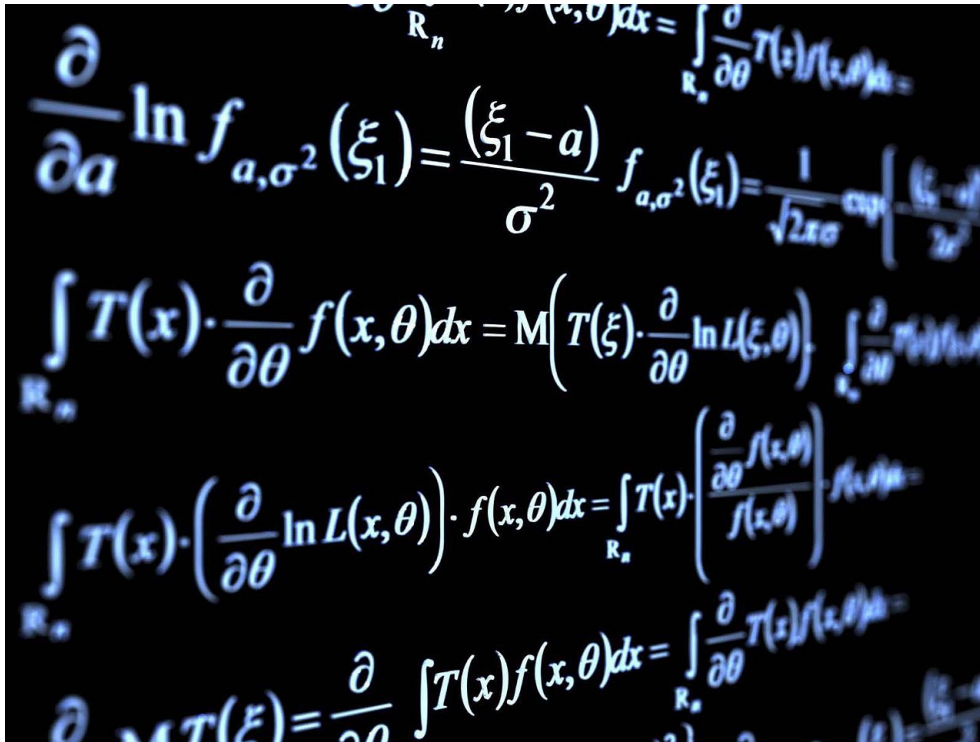


Figure 10.1: <https://towardsdatascience.com/automatic-differentiation-explained-b4ba8e60c2ad> is the source of the image.

The material in this Chapter/Lab is due to Alphonsus Antwi Adu-Bredu, who was the lead GSI for ROB 101 in F-22, while he was also a Ph.D. candidate in Robotics. Here are the key packages we will use.

```

1 # Let's install some Julia packages.
2 using Pkg
3 Pkg.add("Symbolics")
4 Pkg.add("FiniteDiff")
5 Pkg.add("ForwardDiff")
6 Pkg.add("BenchmarkTools");

```

Output

Nothing.

10.1 Symbolic Differentiation

This is the kind of differentiation you learned (or will learn) in your intro Calculus classes. Using this approach, you can take derivatives of expressions using the various rules of differential calculus, such as

- $\frac{d}{dx} c = 0$, where c is a constant
- $\frac{d}{dx} x = 1$
- $\frac{d}{dx} y = 0$, where y is a variable other than x
- $\frac{d}{dx} (u + v) = \frac{d}{dx} u + \frac{d}{dx} v$
- $\frac{d}{dx} (u \cdot v) = u \cdot \frac{d}{dx} v + v \cdot \frac{d}{dx} u$
- $\frac{d}{dx} \frac{u}{v} = \frac{v \cdot \frac{d}{dx} u - u \cdot \frac{d}{dx} v}{v^2}$

- Etc. ...

This form of differentiation is called symbolic, because it allows you to perform differentiation by manipulating symbols, or more precisely, variables, without actually evaluating them as real numbers. This is a big part of Calc I. As you may have guessed, Julia has a sweet package called `Symbolics.jl` which allows you to take symbolic derivatives of input expressions. The next few cells will teach you how to use this package.

Example 1: Find the derivative of the function

$$f(x) = x^3 \cos(x) + x^2 \sin(x)^2 + \log(3x)$$

using Symbolic Differentiation.

Step 1: We load the `Symbolics.jl` package. Just like any other Julia packages used in this course, loading `Symbolics.jl` for the first time often takes a while.

```
1 using Symbolics
```

Step 2: We declare the variables in our function using the `@variables` macro. The only variable in the function above is x .

```
1 @variables x
```

Output

```
[x]
```

Step3: We create the differential operator $\frac{d}{dx}$ and assign it to a variable.

```
1 d/dx = Differential(x)
```

Output

```
(::Differential) (generic function with 2 methods)
```

Step 4: We define $f(x)$ and apply our differential operator $\frac{d}{dx}$ to it.

A cool feature about `Symbolics.jl` is that it renders our expressions in Latex in the cell outputs!

```
1 f = x^3*cos(x) + x^2*sin(x)^2 + log(3x)
2 d/dx(f)
```

Output

$$\frac{d}{dx} (\sin^2(x) x^2 + x^3 \cos(x) + \log(3x))$$

Step 5: We compute the derivative $\frac{df}{dx}$ using the `expand_derivatives` function and store it in variable Δf . The derivative expression is printed out in the cell output.

```
1 dfdx = expand_derivatives(d/dx(f))
```

Output

$$x^{-1} + 2 \sin^2(x) x - x^3 \sin(x) + 3x^2 \cos(x) + 2x^2 \cos(x) \sin(x)$$

Simpler, alternative formulation: We use the `Symbolics.derivative` function to directly compute derivatives of f

```
1 dfdx = Symbolics.derivative(f, x)
```

Output

$$\frac{d}{dx} (\sin^2(x) x^2 + x^3 \cos(x) + \log(3x))$$

Remark: We are not restricted to derivatives of scalar-valued functions. We can also use `Symbolics.jl` to compute gradients, Jacobians, and Hessians of multivariate functions. The following cells show how to get the actual latex code

```
1 # This cell shows how to get the actual latex code
2 using Latexify
3 set_default(fmt = "%.0f", convert_unicode = false)
4 latexify(d/dx(f)) |> print
```

Output

```
\begin{equation}
\mathrm{\frac{d}{d x}}\left( \sin^2\left( x \right) x^2
+ x^3 \cos\left( x \right) + \log\left( 3 x \right) \right)
\end{equation}

$$\frac{d}{dx} (\sin^2(x) x^2 + x^3 \cos(x) + \log(3x))$$

Note that fmt = "%.0f" and fmt = "%.4f" set the formatting for numbers in the expressions.
```

```
1 # This cell shows how to get the actual latex code
2 using Latexify
3 set_default(fmt = "%.4f", convert_unicode = false)
4 latexify(d/dx(f)) |> print
```

Output

```
\begin{equation}
\mathrm{\frac{d}{d x}}\left( \sin^{2.0000}\left( x \right) x^{2.0000} +
x^{3.0000} \cos\left( x \right) + \log\left( 3.0000 x \right) \right)
\end{equation}

$$\frac{d}{dx} (\sin^{2.0000}(x) x^{2.0000} + x^{3.0000} \cos(x) + \log(3.0000x))$$

```

If we later want to evaluate this derivative expression at some numerical value, we can use the `substitute` function, which takes as arguments, the computed derivative expression as well as a “dictionary” that maps variables to their desired values. Here, our desired value for x is 0.5. The value of our derivative at this value of x is 3.0384753223601586

```
1 val = substitute(dfdx, (Dict{x=>0.5}))
```

Output

```
3.038475322360
```

Advantages and Disadvantages of Symbolic Differentiation

- **Advantage:** Symbolic Differentiation allows you to compute exact derivatives up to machine precision.
- **Advantage:** Symbolic Differentiation is also a great tool for analysis. Being able to generate closed-form derivatives of a dynamical model of a robot facilitates the understanding of the dynamics of the robot.
- **Advantage:** Symbolic Differentiation is also significantly useful for generating closed-form derivatives of functions as code. Closed-loop controllers running on real-time systems often have to run at frequencies in the 1kHz range. This constrains the control loop to a time budget of 1 millisecond. Given this time budget, operations in the controller (such as computing derivatives and Jacobians) have to be as fast as possible. Closed-form parameterized functions for derivatives and Jacobians that are computed offline using Symbolic differentiation can then be evaluated online in microseconds, which can be very useful in speeding up your controller and meeting the controller time budget.
- **Disadvantage:** The main disadvantage of symbolic derivatives is that they tend to get gnarly and exponentially long very quickly. This phenomenon is called **Expression swell**.

- **Disadvantage:** Imagine taking the derivative of a function $h(x) = f(x)g(x)$, where $f(x)$ is itself the product $f(x) = u(x)v(x)$. The derivative of $h(x)$ ends up as this gnarly function

$$\frac{d}{dx} h(x) = \left(\frac{d}{dx} u(x)v(x) + u(x) \frac{d}{dx} v(x) \right) g(x) + u(x)v(x) \frac{d}{dx} g(x)$$

leading to inefficient code.

10.2 Numerical Differentiation

Numerical differentiation is another approach for taking derivatives of functions. Unlike Symbolic differentiation, where we derive exact, analytical derivatives of functions, numerical differentiation estimates the derivatives of functions using values of the function at various points.

The simplest and most common method for Numerical Differentiation is **Finite Difference**, as we have used in ROB 101 for building linear approximations of nonlinear functions.

The method of finite differences approximates the derivative $\frac{df(x_0)}{dx}$ of a function f at some (numerical) point x_0 as

$$\frac{df(x_0)}{dx} = \frac{f(x_0 + h) - f(x_0)}{h}.$$

This specific finite difference is specifically known as the **Forward Difference** approximation. There also exists a **Backward Difference** approximation, which is expressed as

$$\frac{df(x_0)}{dx} = \frac{f(x_0) - f(x_0 - h)}{h}$$

and a **Symmetric Difference** approximation expressed as

$$\frac{df(x_0)}{dx} = \frac{f(x_0 + h) - f(x_0 - h)}{2h}.$$

If the derivative of $f(x)$ exists at a point x_0 , for some small h , the forward, backward and symmetric difference approximations are approximately equal. If they end up being significantly different, then the function $f(x)$ is not differentiable at x_0 .

As you may have guessed by now, there exists a really sweet Julia package with implementations of finite differences called `FiniteDiff.jl`, which in fact has empirically the fastest implementation of finite differences in any programming language!

In the following cells, we are going to learn how to use it to take derivatives, gradients, Jacobians, and Hessians of functions. The main difference between derivatives and gradients in relation to the `FiniteDiff.jl` package is that derivatives are used for functions depending on a scalar $x \in \mathbb{R}$ while gradients are used for functions depending on a vector of variables.

Example 2: Find the derivative $\frac{dg(x)}{dx}$ of the function

$$g(x) = x^3 \cos(x) + x^2 \sin(x)^2 + \log(3x)$$

using Finite Differences.

```
1 using FiniteDiff
2
3 g(x) = x^3 ~cos(x) + x^2~sin(x)^2 + log(3x)
```

Output

```
g (generic function with 1 method)
```

To compute the derivative of the function $g(x)$ at $x_0 = 0.5$, we use the function `FiniteDiff.finite_difference_derivative`. The first argument of the function is $g(x)$, the function whose derivative is to be taken, and the second argument is the point at which the derivative is to be taken. Note that we obtain a solution that is approximately equal to that of the Symbolic derivative in the previous section.

```
1 val = FiniteDiff. finite_difference_derivative (g, 0.5)
```

Output

```
3.038475322499568
```

Example 3: Compute the Jacobian of the function

$$h(x) = \begin{bmatrix} x_1^3 \cos(x_2) + x_2^2 \sin(x_1)^2 \\ \log(3x_3) \end{bmatrix}$$

using Finite Differences, where $x \in \mathbb{R}^3$ and x_i is the i -th element of x .

```
1 h(x) = [x[1]^3*cos(x[2]) + x[2]^2*sin(x[1])^2;  
2         log(3x[3])] ]  
3  
4 x0 = [0.1, 0.5, 0.7]  
5 J = FiniteDiff. finite_difference_jacobian (h, x0)
```

Output

```
2x3 Matrix{Float64}:  
0.0759948  0.00948729  0.0  
0.0        0.0          1.42857
```

And the **Hessian** of the super complicated function

$$h_2(x) = x^\top A_2 x + x^\top x x^\top A_4 x$$

can be computed as

```
1 using Random  
2 n=3  
3 A2 = rand(n, n)  
4 A4 = rand(n, n)  
5 h2(x) = x' * A2 * x + x' * x * x' * A4 * x;  
6 x0 = [0.1, 0.5, 0.7]  
7 H = FiniteDiff. finite_difference_hessian (h2, x0)
```

Output

```
3x3 LinearAlgebra.Symmetric{Float64, Matrix{Float64}}:  
3.12914  3.32833  2.95256  
3.32833  5.92296  4.11562  
2.95256  4.11562  5.5207
```

Advantages and Disadvantages of Numerical Differentiation

- **Advantage:** Numerical differentiation is the simplest differentiation method to implement. It only depends on the function values and not on a set of differential calculus rules. In view of this, numerical differentiation can be applied to any differentiable function to get reasonably accurate results.
- **Disadvantage:** Finite Difference suffers from **Truncation error**, resulting in inaccurate derivatives when the step size is too big or too small.
- **Disadvantage:** Finite Difference can also be expensive to compute for functions with large input dimensions.

10.3 Automatic Differentiation

Similar to Symbolic Differentiation, Automatic Differentiation uses a set of rules from differential calculus to compute derivatives. However, its goal is to return a numerical solution instead of a closed-form analytical expression. As such, Automatic Differentiation keeps track of intermediate variables and their derivatives to speed up computation. It decomposes the function into primitive operations using the chain rule, evaluates the operations and their derivatives, and stores them in intermediate variables; see https://julia.quantecon.org/more_julia/optimization_solver_packages.html

To appreciate how automatic differentiation works, let's take a look at an example. We would like to compute the Jacobian of the multivariate function

$$f(x, y) = y \cdot \sin(x) + y^2$$

at a point $x = 0.1, y = 0.3$ using automatic differentiation.

We define the $(\bullet)_x$ operator as the partial derivative with respect to x , that is, $\frac{\partial}{\partial x}$. The notation \dot{a} for the derivative is due to Isaac Newton https://en.wikipedia.org/wiki/Notation_for_differentiation. Next, we decompose $f(x, y)$ into primitive operations and store their values and derivatives with respect to x in intermediate variables.

Intermediate Variables	Derivatives
$a = \sin(x) = 0.0998$	$\dot{a}_x = \cos(x) \cdot \dot{x} = \cos(x) = 0.995$
$b = y \cdot \sin(x) = a \cdot y = 0.02995$	$\dot{b}_x = a \cdot \dot{y} + \dot{a}_x \cdot y = 0 \cdot \sin(x) + y \cdot \cos(x) = 0.2985$
$c = y^2 = 0.09$	$\dot{c}_x = 2y \cdot \dot{y} = 0$
$d = y \cdot \sin(x) + y^2 = b + c = 0.11995$	$\dot{d}_x = \dot{b}_x + \dot{c}_x = y \cdot \cos(x) = 0.2985$

This routine is called a **forward pass** with respect to x . It is worth noting that, unlike symbolic differentiation, the numerical values of each intermediate variable are computed in each row of the table.

Similarly, we define the $(\bullet)_y$ operator as the partial derivative with respect to y , $\frac{\partial}{\partial y}$, and define the intermediate variables and derivatives with respect to y

Intermediate Variables	Derivatives
$a = \sin(x) = 0.0998$	$\dot{a}_y = 0$
$b = y \cdot \sin(x) = a \cdot y = 0.02995$	$\dot{b}_y = a \cdot \dot{y} + \dot{a}_y \cdot y = \sin(x) + 0 = 0.0998$
$c = y^2 = 0.09$	$\dot{c}_y = 2y = 0.6$
$d = y \cdot \sin(x) + y^2 = b + c = 0.11995$	$\dot{d}_y = \dot{b}_y + \dot{c}_y = \sin(x) + 2y = 0.6998$

The Jacobian of $f(x, y)$ can now be defined as

$$Jac = \frac{\partial f(x, y)}{\partial(x, y)} = [\dot{d}_x \quad \dot{d}_y] = [0.2985 \quad 0.6998]$$

The computational routine we have just demonstrated above is a special kind of Automatic Differentiation called **Forward-Mode Automatic Differentiation**.

In general, you would perform a forward pass for each argument to your function f . If the function argument is a vector, you would perform a forward pass for each element in the vector. The computational complexity is of the order $O(n)$. You can imagine that this could end up being computationally inefficient for functions $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ with $n \gg \gg m$. In such cases, another special kind of Automatic Differentiation called **Reverse-mode Automatic Differentiation** will be much more appropriate.

Reverse-mode Automatic Differentiation first performs a forward pass to compute only the intermediate variable values without taking the derivatives. It then performs a reverse (backward) pass to compute the derivatives of the intermediate variables. A function $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$, will require m reverse passes to compute the Jacobian of the function. Because of this, Reverse-Mode automatic differentiation is suitable for functions where $n \gg \gg m$ and inefficient for functions where $m \gg \gg n$.

As you may rightly guess, Julia has packages that have super-fast implementations of forward-mode and reverse-mode automatic differentiation named `ForwardDiff.jl` and `ReverseDiff.jl`. In the following cells, we will demonstrate the usage of `ForwardDiff.jl` for computing derivatives and Jacobians.

Example 4: Find the derivative $\frac{df_2}{dx}$ of the function

$$f_2(x) = x^3 \cos(x) + x^2 \sin(x)^2 + \log(3x)$$

using Forward-mode Automatic Differentiation.

```
1 using ForwardDiff
2 #
3 f_2(x) = x^3*cos(x) + x^2*sin(x)^2 + log(3x)
```

Output

```
f_2 (generic function with 1 method)
```

Next, we will evaluate the derivative of our function at the point $x=0.5$ using the function `ForwardDiff.derivative`. The first argument is the function whose derivative is to be taken, while the second argument is the point at which the derivative is taken.

Notice that we get the exact same result (3.0384753223601586) we had from Symbolic Differentiation. This goes to show that, unlike Finite Differences, and just like Symbolic Differentiation, **Automatic Differentiation computes the exact derivative to machine precision.**

```
1 x = 0.5
2 df_2dx = ForwardDiff.derivative(f_2, x)
```

Output

```
3.0384753223601586
```

Example 5: Compute the Jacobian of the function

$$h_2(x) = \begin{bmatrix} x_1^3 \cos(x_2) + x_2^2 \sin(x_1)^2 \\ \log(3x_3) \end{bmatrix}$$

at the point $x_0 = [0.1, 0.5, 0.7] \in \mathbb{R}^3$ using Forward-mode Automatic Differentiation, where x_i is the i -th component of x .

```
1 h_2(x) = [ x[1]^3*cos(x[2]) + x[2]^2*sin(x[1])^2; log(3x[3]) ]
```

Output

```
h_2 (generic function with 1 method)
```

```
1 x0 = [0.1, 0.5, 0.7]
2 Jac = ForwardDiff.jacobian(h_2, x0)
```

Output

```
2x3 Matrix{Float64}:
 0.0759948  0.00948729  0.0
 0.0        0.0          1.42857
```

Advantages and Disadvantages of Automatic Differentiation

- **Advantage:** Automatic Differentiation is very fast at computing derivatives. Because of this, automatic differentiation is the primary differentiation tool for computing derivatives when training deep neural networks.
- **Advantage:** Unlike with Finite Differences, Automatic Differentiation computes exact derivatives to machine precision.

- **Advantage:** Implementations of Automatic Differentiation in popular Deep Learning frameworks like Pytorch and Tensorflow build **Computational Graphs** when performing the forward pass. These computational graphs allow for easy debugging and analysis.
- **Advantage:** It is possible to differentiate an algorithm.
- **Disadvantage:** Automatic Differentiation is not as easy to implement on your own as Finite Differences. Care usually has to be taken to implement it in a manner that is memory-efficient, particularly for reverse-mode automatic differentiation. Regardless, there are super-fast and clean implementations of automatic differentiation like `ForwardDiff.jl` available here <https://juliadiff.org/ForwardDiff.jl/stable/> and `Zygote.jl` available here <https://fluxml.ai/Zygote.jl/latest/> that exist off-the-shelf and can be applied to any use.

10.4 Which one is fastest? And by How Much?

We use the package `textttBenchmarkTools.jl` to benchmark the relative speeds of the various differentiation methods we've discussed so far. We will be computing the Jacobian of this gnarly vector-valued function

$$h_3(x) = \begin{bmatrix} x_1^3 \cos(x_2) + x_2^2 \sin(x_1)^2 \\ \log(3x_3) \end{bmatrix}$$

at the point $x_0 = [0.5, 0.3, 0.2]$

```
1 using BenchmarkTools
2
3 h3(x) = [ x[1]^3*cos(x[2]) + x[2]^2*sin(x[1])^2; log(3x[3]) ]
4
5 x0 = [0.5, 0.3, 0.2]
```

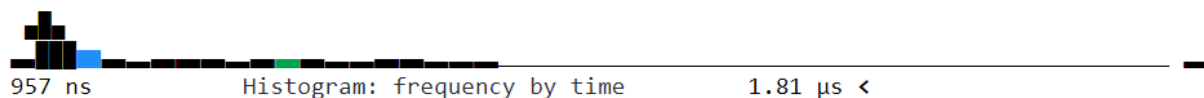
Output

```
3-element Vector{Float64}:
 0.5
 0.3
 0.2
```

```
1 # Benchmarking Automatic Differentiation
2
3 @benchmark ForwardDiff.jacobian(h3, x0)
```

Output

```
BenchmarkTools.Trial: 10000 samples with 20 evaluations.
Range (min ... max): 957.050 ns ... 283.430 μs | GC (min ... max): 0.00% ... 99.28%
Time (median): 1.010 μs | GC (median): 0.00%
Time (mean ± σ): 1.130 μs ± 3.946 μs | GC (mean ± σ): 4.91% ± 1.40%
```



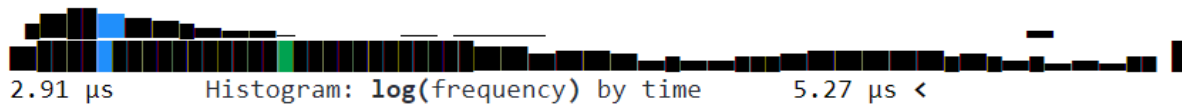
Memory estimate: 640 bytes, allocs estimate: 5.

```
1 # Benchmarking Finite Differences
2
3 @benchmark FiniteDiff.finite_difference_jacobian(h3, x0)
```

Output

BenchmarkTools.Trial: 10000 samples with 9 evaluations.

Range (min ... max):	2.910 μ s ... 970.604 μ s	GC (min ... max):	0.00% ... 99.13%
Time (median):	3.099 μ s	GC (median):	0.00%
Time (mean \pm σ):	3.608 μ s \pm 19.126 μ s	GC (mean \pm σ):	10.56% \pm 1.99%



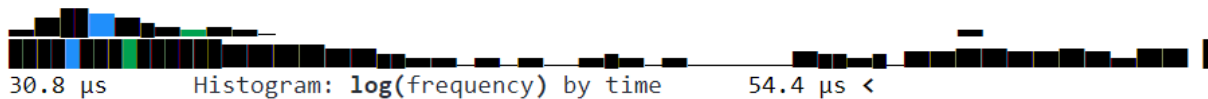
Memory estimate: 2.47 KiB, allocs estimate: 43.

```
1 # Benchmarking Symbolic Differentiation
2
3 @variables x1, x2, x3
4 h3s = [ x1^3*cos(x2) + x2^2*sin(x1)^2; log(3x3) ]
5
6 @benchmark begin
7     dh3 = Symbolics.derivative(h3s, x)
8     substitute.(dh3, (Dict(x1=>x0[1], x2=>x0[2], x3=>x0[3]),))
9 end
```

Output

BenchmarkTools.Trial: 10000 samples with 1 evaluation.

Range (min ... max):	30.831 μ s ... 8.782 ms	GC (min ... max):	0.00% ... 99.36%
Time (median):	32.291 μ s	GC (median):	0.00%
Time (mean \pm σ):	33.894 μ s \pm 87.633 μ s	GC (mean \pm σ):	2.57% \pm 0.99%



Memory estimate: 7.47 KiB, allocs estimate: 143.

Summary

Automatic Differentiation is the fastest, followed by Finite Differences, and Symbolic Differentiation is the slowest! And it's not even close!

- Automatic Differentiation: (median): 1.010 μ s
- Finite Differences: (median): 3.099 μ s
- Symbolic Differentiation: (median): 32.291 μ s

Appendix A

Summary of Key Julia Commands

The following is a curated list of the primary Julia commands that we use in ROB 101. The list is not meant to be exhaustive. You will want to add all of these commands and more to your personal curated list of Julia commands; see Chapter 0.4.

A.1 Creating Vectors and Matrices

Vectors and Matrices in Julia can hold variables of a single `Type`. We are primarily using variables with these 4 types

- (a) `Int64`
- (b) `Float64`
- (c) `Char`
- (d) `String`

How to create vectors:

- (a) Separate elements by commas or semicolons:

```
1 animalsVector = ["lemur", "elephant", "tiger", "panda", "zebra", "cuttlefish"]
```

Output

```
6-element Vector{String}:
 "lemur"
 "elephant"
 "tiger"
 "panda"
 "zebra"
 "cuttlefish"
```

```
1 numbersVector = numbersVector = [1; 2; 3; 4]
```

Output

```
4-element Vector{Int64}:
 1
 2
 3
 4
```

In lecture, we call the above a **column vector**.

- (b) If you use spaces, then you obtain a $1 \times n$ matrix, which in lecture, we call a **row vector**

```
1 animalsVector = ["lemur" "elephant" "tiger" "panda" "zebra" "cuttlefish"]
```

Output

```
1×6 Matrix{String}:  
 "lemur" "elephant" "tiger" "panda" "zebra" "cuttlefish"
```

```
1 numbersVector = [1.0 2 3 4]
```

Output

```
1×4 Matrix{Float64}:  
 1.0  2.0  3.0  4.0
```

- (c) The infamous **adjoint vector** is a special kind of row vector that when multiplied by a column vector of the appropriate size, produces a number and not a 1-element vector.

```
1 numbersVector = [1.0; 2; 3; 4]'
```

Output

```
1×4 adjoint{::Vector{Float64}} with eltype Float64:  
 1.0  2.0  3.0  4.0
```

```
1 x = [1; 2; 3.0]  
2 y = [4; 5; 6]  
3  
4 @show x' # adjoint of x, looks like a row vector  
5  
6 z=x'*y
```

Output

```
x' = [1.0 2.0 3.0]  
  
32.0
```

- (d) `Vector{Float64}(undef, n)` creates an n -element vector with undefined entries.
- (e) `Vector{Float64}(undef, 0)` creates a 0-element vector with undefined entries. This is a truly empty vector.
- (f) If A is a `Vector`, then `B = copy(A)` creates the independent `Vector B` and it is a copy of the `Vector A`.
- (g) If A is a `Vector`, then `B = A` creates the `Vector B` and links it to A . Hence, changes to A also show up in B and vice versa, changes to B also show up in A . Usually, you do not want this to happen and hence you should consider the `copy` command.

How to create matrices:

- (a) Use spaces to separate elements in a same row, and use either commas or semicolons to separate rows.

```
1 A=[1.0 2.0; 3.0 4.0]
```

Output

```
2×2 Matrix{Float64}:  
 1.0  2.0  
 3.0  4.0
```

- (b) `zeros(n,m)`
- (c) `ones(n,m)`
- (d) `using Random`
- (e) `Random.seed!` (4321) or whatever value you want sets the “seed” of the random number generators. This way, we can get repeatable results in HW.
- (f) `rand(n,m)` (uniform) and with an extra “n” `randn(n,m)` (normal, aka Gaussian or Bell Curve)
- (g) There is no explicit command to create an $n \times n$ identity matrix. You do it like so

```

1 # how to make an identity matrix in Julia
2 using LinearAlgebra
3 n=4 # set your size here
4 Id = zeros(n, n) + I
5 # I is a special operator that when added to a square zero matrix
6 # produces an identity matrix of the appropriate size

```

Output

```

4×4 Matrix{Float64}:
 1.0  0.0  0.0  0.0
 0.0  1.0  0.0  0.0
 0.0  0.0  1.0  0.0
 0.0  0.0  0.0  1.0

```

The operator `I` is part of the `LinearAlgebra` package. If you have not already called `using LinearAlgebra`, you will not be able to use the identity operator `I`.

- (h) `Matrix{Float64}(undef, n, m)` creates a matrix of a specified size without specifying its entries. Note that `undef` is a Julia keyword to create a value that is “undefined”.

```

1 Matrix{Float64}(undef, 3, 4)

```

Output

```

3×4 Matrix{Float64}:
 6.90947e-310  6.90947e-310  6.90947e-310  6.90947e-310
 6.90947e-310  6.90947e-310  6.90947e-310  6.9094e-310
 6.90947e-310  6.90947e-310  6.90947e-310  6.90947e-310

```

- (i) `Matrix{Float64}(undef, n, 0)` creates a matrix with n rows and zero columns, while to create a matrix with zero rows and m columns, you use the command `Matrix{Float64}(undef, 0, m)`.

A.2 Indexing or Slicing Vectors and Matrices

Remark A.1 Julia uses ****1-based indexing**** which means that the index starts at 1 and not 0. Be aware that 0-based indexing is used in some other programming languages, such as C++. If you are familiar with MATLAB, it uses 1-based indexing.

```

1 @show row_vec = [1 3 5 7 9]
2 println(" ")
3 # Indexing one or more elements
4 @show x=row_vec[3]
5 @show x=row_vec[2:3]
6 @show x=row_vec[2:end]
7 @show x=row_vec[[2 3 5]]

```

Output

```
row_vec = [1 3 5 7 9] = [1 3 5 7 9]
```

```
x = row_vec[3] = 5
x = row_vec[2:3] = [3, 5]
x = row_vec[2:end] = [3, 5, 7, 9]
x = row_vec[[2 3 5]] = [3 5 9]
```

```
1×3 Matrix{Int64}:
 3  5  9
```

```
1 using Random # Using an external package called Random
2 Random.seed!(1234) # Set the seed so that each of you get the same results.
3 A = rand(4, 6)
4
5 x = A[:,2]
```

Output

```
4-element Vector{Float64}:
 0.7940257103317943
 0.8541465903790502
 0.20058603493384108
 0.2986142783434118
```

```
1 using Random # Using an external package called Random
2 Random.seed!(1234) # Set the seed so that each of you get the same results.
3 A = rand(4, 6)
4
5 x = A[2,:] # Produces a column vector!
```

Output

```
6-element Vector{Float64}:
 0.7667970365022592
 0.8541465903790502
 0.5796722333690416
 0.9567533636029237
 0.6516642063795697
 0.9646697763820897
```

```
1 using Random # Using an external package called Random
2 Random.seed!(1234) # Set the seed so that each of you get the same results.
3 A = rand(4, 6)
4
5 x = A[2:2,:] # to keep it as a row
```

Output

```
1×6 Matrix{Float64}:
 0.766797  0.854147  0.579672  0.956753  0.651664  0.964667
```

```
1 using Random # Using an external package called Random
2 Random.seed!(1234) # Set the seed so that each of you get the same results.
3 A = rand(4, 6)
4
5 x = A[3:end,3:end]
```

Output

```
2×4 Matrix{Float64}:
 0.648882  0.646691  0.0566425  0.945775
 0.0109059 0.112486  0.842714  0.789904
```

```
1 using Random # Using an external package called Random
2 Random.seed!(1234) # Set the seed so that each of you get the same results.
3 A = rand(4, 6)
4 #
5 indRow = [3, 4] # note the commas
6 indCol = [2, 5, 6] # note the commas
7 #
8 x = A[indRow, indCol]
```

Output

```
2×3 Matrix{Float64}:
 0.200586  0.0566425  0.945775
 0.298614  0.842714  0.789904
```

A.3 1-Element Vectors and 1 x 1 Matrices

A $1 \times n$ matrix times an $n \times 1$ vector produces a 1-element vector in Julia. You have to extract the element from the vector if you want to use it for assigning an element some other vector or matrix.

```
1 A = [1.0 2]
2 b = [2.0; 3.0]
3 y=A*b
```

Output

```
1-element Vector{Float64}:
 8.0
```

Note that y is a 1-element vector.

```
1 @show y
2 @show y[1]
```

Output

```
y = [8.0]
y[1] = 8.0
```

```
8.0
```

```
1 A=zeros(2,3)
2 A[1,3] = y
```

Output

```
MethodError: Cannot `convert` an object of type Vector{Float64} to an object of type Float64
```

Closest candidates are:

```
convert(::Type{T}, ::T) where T<:Number at number.jl:6
convert(::Type{T}, ::Number) where T<:Number at number.jl:7
```

```
convert(::Type{T}, ::Base.TwicePrecision) where T<:Number at twiceprecision.jl:250
...
```

Stacktrace:

```
[1] setindex!(::Matrix{Float64}, ::Vector{Float64}, ::Int64, ::Int64)
  @ Base ./array.jl:841
[2] top-level scope
  @ In[34]:2
[3] eval
  @ ./boot.jl:360 [inlined]
[4] include_string(mapexpr::typeof(REPL.softscope), mod::Module, code::String, filename::String)
  @ Base ./loading.jl:1094
```

```
1 A=zeros(2, 3)
2 A[1, 3] = y[1]
3 A
```

Output

```
2×3 Matrix{Float64}:
 0.0  0.0  8.0
 0.0  0.0  0.0
```

The product of a $1 \times n$ adjoint vector and an n -element vector is a scalar and not some kind of array. You do not need to extract the final answer in order to use it somewhere at a later point. In the output, note that there are no brackets around the number 8.0 or other indication that the result is a vector.

```
1 x = [1.0; 2]
2 y = [2.0; 3.0]
3 @show x'
4 @show y
5 z=x' * y
```

Output

```
x' = [1.0 2.0]
y = [2.0, 3.0]

8.0
```

Where as here we multiply a 1×2 matrix times a 2×1 vector in Julia and produce a 1-element vector.

```
1 x = [1.0 2]
2 y = [2.0; 3.0]
3 @show x
4 @show y
5 z=x*y
```

Output

```
x = [1.0 2.0]
y = [2.0, 3.0]
```

```
1-element Vector{Float64}:
 8.0
```

A.4 Size vs Length Commands, Which to use for Vectors and Matrices

```
1 A = [ 1 2 3; 4 5 6; 7 8 9]
```

Output

```
3×3 Matrix{Int64}:  
 1  2  3  
 4  5  6  
 7  8  9
```

```
1 b=[1;2;3]
```

Output

```
3-element Vector{Int64}:  
 1  
 2  
 3
```

```
1 length(b)
```

Output

```
3
```

```
1 length(A) # gives total number of elements in A
```

Output

```
9
```

```
1 size(A) # number of rows and number of columns
```

Output

```
(3, 3)
```

```
1 nRowsA, nColsA = size(A) # number of rows and number of columns  
2 @show nRowsA  
3 nColsA
```

Output

```
nRowsA = 3
```

```
3
```

It is not recommended to apply the `size` command to a `Vector`, but it does return an answer.

```
1 size(b)
```

Output

```
(3,)
```

If you try to assign the number of columns, you'll get an error, however. It really is better not to use the `size` command on vectors.

```
1 nRowsb, nColsb = size(b)
```

Output

```
BoundsError: attempt to access Tuple{Int64} at index [2]
```

```
Stacktrace:
```

```
[1] indexed_iterate(t::Tuple{Int64}, i::Int64, state::Int64)
  @ Base ./tuple.jl:86
[2] top-level scope
  @ In[11]:1
[3] eval
  @ ./boot.jl:360 [inlined]
[4] include_string(mapexpr::typeof(REPL.softscope), mod::Module, code::String, filename::String)
  @ Base ./loading.jl:1094
```

You can also obtain the number of rows or just the number of columns of a matrix.

```
1 A=rand(3,4)
2 display(A)
3 @show nRows = size(A,1); # note the 1
```

Output

```
3×4 Matrix{Float64}:
 0.658815  0.59552  0.61816  0.0368842
 0.515627  0.292462  0.66426  0.643704
 0.260715  0.28858  0.753508  0.
```

```
nRows = size(A, 1) = 3
```

```
1 A=rand(3,4)
2 display(A)
3 @show nCols = size(A,2); # note the 2
```

Output

```
3×4 Matrix{Float64}:
 0.525057  0.082207  0.218177  0.932984
 0.61201  0.199058  0.362036  0.827263
 0.432577  0.576082  0.204728  0.0992992
```

```
nCols = size(A, 2) = 4
```

Length vs Size

- `length(X)` is the total number of elements in the array X . If X is a vector, then it is the usual length. If X is a matrix, then `length` returns the product of the number of rows and the number of columns.
- `nRows`, `nCols = size(A)` is best applied to matrices. It provides the number of rows and the number of columns. You can also use it as follows to return just the number of rows or just the number of columns,
 - `nRows = size(A,1)`
 - `nCols = size(A,2)`
- Applying the `size` command to vectors is just asking for trouble. Please avoid doing this.

A.5 Flow Control: If-then-else and Looping

```
1 # Read me and run me
2 # More conditions can be evaluated using elseif
3 x = 1
4 y = 1
5 if x < y
6     println("x is less than y")
7 elseif x > y
8     println("x is greater than y")
9 else
10    println("x is equal to y")
11 end
```

Output

```
x is equal to y
```

```
1 x=[4.0]
2 if length(x) == 1
3     if string(typeof(x)) [1:6] == "Vector" # Double equals means EQUIVALENT TO
4         @show typeof(x)
5         @show typeof(x[1])
6         @show x=x[1]
7     else
8         @show x
9     end
10 end
11 x
```

Output

```
typeof(x) = Vector{Float64}
typeof(x[1]) = Float64
x = x[1] = 4.0
```

```
4.0
```

```
1 # Run me to see the counter augment by 2 as we go through the loop
2 mySum=0
3 for k = 13:2:26 #k_start:k_increment:k_end note the extra colon
4     mySum = mySum + k # add k to mySum
5     @show (k, mySum)
6 end
7 mySum
```

Output

```
(k, mySum) = (13, 13)
(k, mySum) = (15, 28)
(k, mySum) = (17, 45)
(k, mySum) = (19, 64)
(k, mySum) = (21, 85)
(k, mySum) = (23, 108)
(k, mySum) = (25, 133)
```

```
133
```

```

1 A=[
2 -0.991273  -0.850184  -1.06297    0.609128   0.498376   0.411742
3  -0.0       0.592798  -0.509626  -0.547334  -0.670348  -0.826093
4   0.0       -0.0       0.673963   0.24726   -0.52135   0.680463
5  -0.0       -0.0       0.0        -0.860824  -0.71363   -0.322208
6  -0.0       0.0        -0.0       0.0        0.249046   0.373825
7   0.0       -0.0       -0.0       0.0        0.0        -0.044526
8 ]
9 #
10 b=[0.6849753810315695
11 0.7387064229251636
12 0.8252920694637993
13 0.2707345660171885
14 0.254653180382145
15 0.1555903546558295]; #semicolon suppresses output

```

Output

(nothing)

```

1 N=length (b)
2 x=zeros (N,1) # Makes x an N x 1 matrix of zeros
3 x[N] = b[N]/A[N,N]
4 for i = (N-1):-1:1
5     x[i] = (b[i]-A[i,i+1:end]'*x[i+1:end])/A[i,i]
6 end
7 x

```

Output

```

6x1 Matrix{Float64}:
-21.381592956092216
 9.163388707678338
11.142804824616672
-4.202499623521532
 6.267662793927989
-3.494370809321059

```

A second kind of loop, the while loop.

```

1 N=277777788888899
2 i=0; Ndigits=digits (N) ; K=length (Ndigits)
3 while (K>1)
4     i=i+1
5     prodN=Ndigits[1]
6     for k = 2:K
7         prodN=prodN*Ndigits [k]
8     end
9     @show prodN
10    Ndigits=digits (prodN)
11    K=length (Ndigits)
12 end
13 println ("The multiplicative persistence of $N is  $i")

```

Output

```

prodN = 4996238671872
prodN = 438939648

```

```

prodN = 4478976
prodN = 338688
prodN = 27648
prodN = 2688
prodN = 768
prodN = 336
prodN = 54
prodN = 20
prodN = 0
The multiplicative persistence of 277777788888899 is 11

```

A.6 Functions

```

1 # Simplest kind of function
2 f(x) = 5x + 2

```

Output

```
f (generic function with 1 method)
```

```
1 f(x) = x + 4*x + sin(x)*x
```

Output

```
f (generic function with 1 method)
```

```
1 f(pi/4)
```

Output

```
4.482351184257038
```

```

1 function f(x)
2     y = x + 4*x + sin(x)*x
3     return y
4 end

```

Output

```
f (generic function with 1 method)
```

```
1 f(pi/4)
```

Output

```
4.482351184257038
```

```

1 function peel_one_layer(Temp, k)
2     pivot = Temp[k, k]
3     if abs(pivot) < 1e-6
4         C=NaN; R=NaN; Temp=NaN
5         println("Pivot is too small")
6     else
7         C = Temp[:, k] / pivot
8         R = Temp[k:k, :]

```

```

9     Temp = Temp - C*R
10    end
11    return C, R, Temp
12 end

```

Output

peel_one_layer (generic function with 1 method)

```

1 function myLU(A)
2     Temp = copy(A) # Initialize Temp matrix by
3             #copying the original matrix A
4     nRows, nCols = size(Temp) # Get the size of the input matrix
5     K = minimum([nRows, nCols])
6     # Initialize the lower and upper triangular matrix
7     L = zeros(Float64, (nRows, K)) # Using zeros function by
8             # specifying both type and size
9     U = zeros(Float64, (K, nCols))
10    # Here we do the actual factorization
11    for k = 1:K
12        C, R, Temp = peel_one_layer(Temp, k) # calling a separate function
13        L[:, k] = C
14        U[k:k, :] = R
15    end
16    return L, U
17 end

```

Output

myLU (generic function with 1 method)

```

1 using Random
2 using LinearAlgebra
3 Random.seed!(09182022)
4 A = randn(5, 7)
5 #
6 L, U = myLU(A)

```

Output

```

([1.0 -0.0 ... 0.0 -0.0; 1.3841751352784288 1.0 ... 0.0 -0.0; ... ; 1.8240441684383097
1.7669720431792901 ... 1.0 -0.0; 1.1660675070497575 0.4629237529415664 ...
-0.8475551711339656 1.0], [-0.3236905701556438 0.9372755784538437 ...
-0.009357226530146447 -0.0076384514441543774; 0.0 -2.1005108102249452 ...
1.4460922800584077 0.41893233388727713; ... ; 0.0 0.0 ... 0.19792290864355966
-0.6285612979102336; -5.551115123125783e-17 0.0 ... -1.3304631421106152
1.0052145594104471])

```

A.7 Applying Functions to Arrays via Broadcasting

Julia has a special syntax for applying functions to individual elements of arrays. You need to add a “dot” after the function and before the argument to the function. It’s best understood by doing it.

```

1 x = [0 pi/4 pi/2 3pi/4 pi 5pi/4]

```

Output

```
1×6 Matrix{Float64}:
 0.0  0.785398  1.5708  2.35619  3.14159  3.92699
```

```
1 sin.(x) # note the dot
```

Output

```
1×6 Matrix{Float64}:
 0.0  0.707107  1.0  0.707107  1.22465e-16  -0.707107
```

This also works with functions that you write yourself! It pretty awesome.

```
1 f(x) = 1 + 2x + sin(x^2)
```

Output

```
f (generic function with 1 method)
```

```
1 f.(x) # note the dot
```

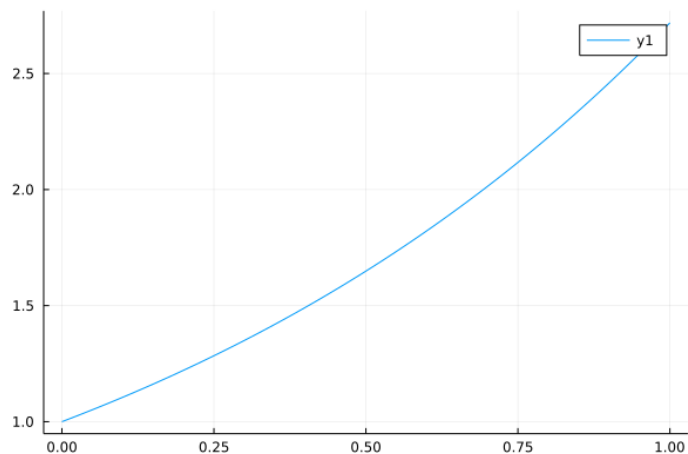
Output

```
1×6 Matrix{Float64}:
 1.0  3.14927  4.76586  5.04438  6.85288  9.13678
```

A.8 Plotting

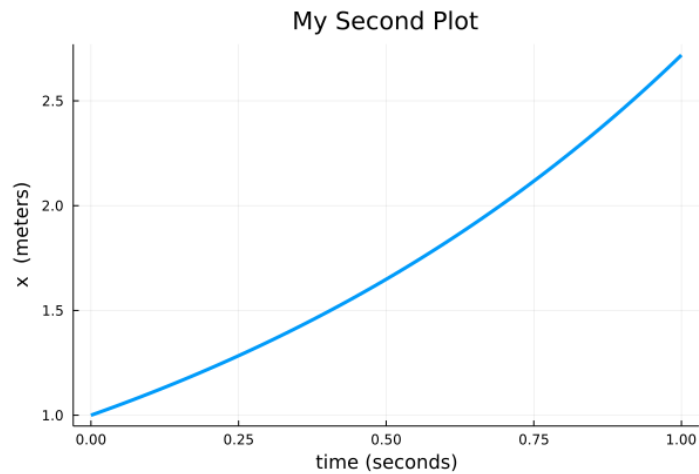
```
1 using Plots
2 gr()
3 time = 0:0.01:1
4 x = exp.(time)
5 plot(time, x)
```

Output



```
1 time = 0:0.01:1
2 x = exp.(time)
3 titre = "My Second Plot"
4 plot(time, x, title=titre, linewidth=3, legend = false)
5 plot!(xlabel = "time (seconds)", ylabel = "x (meters)")
```

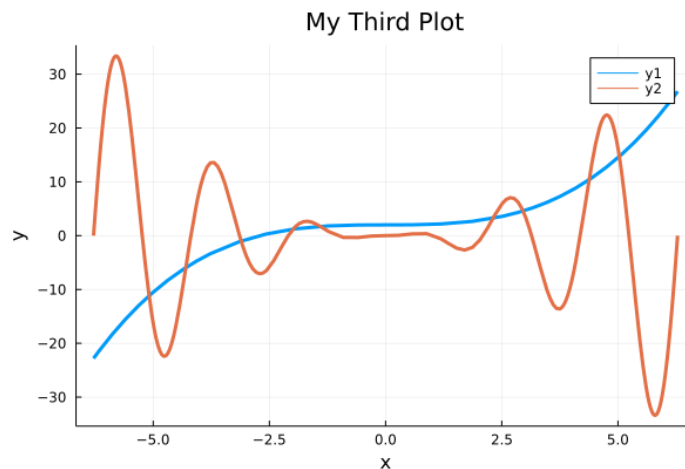
Output



Recall that **plot!** adds to the previous plot; the command is pronounced “plot bang”.

```
1 f(x) = .1x^3 + 2
2 g(x) = (x^2)*sin(3*x)
3 xmin = -2*pi
4 xmax = 2*pi
5 titre = "My Third Plot"
6 plot(f, xmin, xmax, title=titre, linewidth=3)
7 plot!(g, xmin, xmax, title=titre, linewidth=3)
8 plot!(xlabel = "x", ylabel = "y")
```

Output



```
1 time = 0:0.01:1
2 x = exp.(time)
3 plot(time, x)
4 png("AppendixA_SimplePlot") # Produces the png file
```

Output

(nothing comes to the screen) You will find a .png file in your files tab.

Select it and then hit the download button.

A.9 Suggested Resources

- Why Julia?
https://juliadatascience.io/julia_accomplish
- Why Julia in the eyes of MIT?
<https://web.mit.edu/18.06/www/Spring17/Julia-intro.pdf>
- Compares Matlab, Python, and Julia side by side
<https://cheatsheets.quantecon.org/>
- Noteworthy differences with Matlab
<https://docs.julialang.org/en/v1/manual/noteworthy-differences/>
- Rich source of plotting examples
<https://gist.github.com/gizmaa/7214002>
- Compact cheatsheet for plotting in Julia
<https://github.com/sswatson/cheatsheets/blob/master/plotsjl-cheatsheet.pdf>
- Very compact, maybe not the first resource you turn to
<https://juliadocs.github.io/Julia-Cheat-Sheet/>
- Julia for Data Science
<https://juliadatascience.io/>

Appendix B

Building Better Functions Through Structured Returns or Named Tuples

This appendix shows you how to write functions that return answers in the compact format that is favored by Julia’s built-in functions. In ROB 101, it is NOT necessary to use a “structured return”. You can continue building your functions as we have done in the Labs, HWs, and Projects. However, if you want to get a bit fancier, you can learn how to add that “je ne sais pas” to your coding repertoire.

First we illustrate what we mean by “structured return” by using a standard Julia function.

```
1 using LinearAlgebra
2 A = [2 2 3; 4 5 6; 7 8 9]
3 F = lu(A)
```

Output

```
LU{Float64, Matrix{Float64}}
L factor:
3×3 Matrix{Float64}:
 1.0      0.0      0.0
 0.571429  1.0      0.0
 0.285714 -0.666667  1.0
U factor:
3×3 Matrix{Float64}:
 7.0  8.0  9.0
 0.0  0.428571  0.857143
 0.0  0.0  1.0
```

Note that the permutation matrix P was not printed! Is it because $P = I_{3 \times 3}$? Let’s check!

```
1 F.P
```

Output

```
3×3 Matrix{Float64}:
 0.0  0.0  1.0
 0.0  1.0  0.0
 1.0  0.0  0.0
```

Nope! Rows three and one are swapped. It is not because $P = I_{3 \times 3}$.

```
1 F.L
```

Output

```

3x3 Matrix{Float64}:
 1.0      0.0      0.0
 0.571429  1.0      0.0
 0.285714 -0.666667  1.0

```

1 **F.U**

Output

```

3x3 Matrix{Float64}:
 7.0  8.0      9.0
 0.0  0.428571  0.857143
 0.0  0.0      1.0

```

B.1 Building a Structured Return of Your Own

```

1 # First, build a structure
2 struct luStructReturn
3     # List the variables to be used in the return statement
4     # of the function
5     L
6     U
7     P
8 end
9 #
10 # Then use it in the function when you define the return in line 47
11 #
12 using LinearAlgebra
13 #
14 function myLU(M::Array{<:Number, 2}, aTol=1e-12)
15 # Works for rectangular matrices
16     n, m = size(M)
17     k=min(n, m)
18     Areduced = deepcopy(M)
19     L = Array{Float64, 2}(undef, n, 0)
20     U = Array{Float64, 2}(undef, 0, m)
21     P=zeros(n, n) + I
22     for i = 1:k
23         C = Areduced[:, i] # k-th column
24         R = Areduced[i:i, :] # k-th row
25         if maximum(abs.(C)) <= aTol #column of zeros
26             C=0.0*C; C[i]=1.0;
27             U=[U; R];
28             L=[L C];
29             Areduced=Areduced.-C*R;
30         else # put the biggest entry to the top
31             ii=argmax(abs.(C));
32             nrow=ii[1]
33             P[[i, nrow], :] = P[[nrow, i], :];
34             Areduced[[i, nrow], :] = Areduced[[nrow, i], :];
35             if i>1
36                 L[[i, nrow], :] = L[[nrow, i], :];
37             end
38             C = Areduced[:, i] # k-th column
39             R = Areduced[i:i, :] # k-th row
40             pivot = C[i];

```

```

41         C=C/pivot #normalize all entieres by C[i]
42         U=[U;R];
43         L=[L C];
44         Areduced=Areduced-C*R;
45     end
46 end
47 F = luStructReturn(L,U,P)
48 return F
49 end

```

Output

myLU (generic function with 2 methods)

```

1 A = [2 2 3; 4 5 6; 7 8 9]
2 F = myLU(A)

```

Output

```

luStructReturn([1.0 0.0 0.0; 0.5714285714285714 1.0 0.0; 0.2857142857142857
-0.6666666666666666 1.0], [7.0 8.0 9.0; 0.0 0.4285714285714288
0.8571428571428577; 0.0 0.0 1.0], [0.0 0.0 1.0; 0.0 1.0 0.0; 1.0 0.0 0.0])

```

```
1 F.L
```

Output

```

3x3 Matrix{Float64}:
 1.0      0.0      0.0
 0.571429  1.0      0.0
 0.285714 -0.666667  1.0

```

```
1 F.U
```

Output

```

3x3 Matrix{Float64}:
 7.0  8.0  9.0
 0.0  0.428571  0.857143
 0.0  0.0  1.0

```

```
1 F.P
```

Output

```

3x3 Matrix{Float64}:
 0.0  0.0  1.0
 0.0  1.0  0.0
 1.0  0.0  0.0

```

Heads Up!

Suppose you want to modify your structured return to include the row swapping indices that define the permutation matrix, P . You have two choices:

- use a new name for the structure, or
- stop your kernel and clear outputs.

Otherwise, you will receive an error message as follows

```
invalid redefinition of constant luStructReturn
```

Stacktrace:

```
[1] top-level scope
    @ In[11]:2
[2] eval
    @ ./boot.jl:360 [inlined]
[3] include_string(mapexpr::typeof(REPL.softscope), mod::Module,
code::String, filename::String)
    @ Base ./loading.jl:1094
```

In the following we

- stopped our kernel and cleared the outputs;
- redefined the structured return to include the indices, p ; and
- changed a few lines of code; see 8, 22, 34, 49, and 50.

```
1 # First, build the structure
2 struct luStructReturn
3     # List the variables to be used in the return statement
4     # of the function
5     L
6     U
7     P
8     p
9 end
10 #
11 # Then use it in the function when you define the return
12 #
13 using LinearAlgebra
14 #
15 function myLU(M::Array{<:Number, 2}, aTol=1e-12)
16 # Works for rectangular matrices
17     n, m = size(M)
18     k=min(n, m)
19     A=deepcopy(M)
20     L = Array{Float64, 2}(undef, n, 0)
21     U = Array{Float64, 2}(undef, 0, m)
22     p=collect(1:n)
23     for i = 1:k
24         C = A[reduced{:, i}] # k-th column
25         R = A[reduced{i, i, :}] # k-th row
26         if maximum(abs.(C)) <= aTol #column of zeros
27             C=0.0*C; C[i]=1.0;
28             U=[U; R];
29             L=[L C];
```

```

30     Areduced=Areduced.-C*R;
31     else # put the biggest entry to the top
32         ii=argmax( abs.(C) );
33         nrow=ii[1]
34         p[[i,nrow]]=p[[nrow,i]];
35         Areduced[[i,nrow],:]=Areduced[[nrow,i],:];
36         if i>1
37             L[[i,nrow],:]=L[[nrow,i],:];
38         end
39         C = Areduced[:,i] # k-th column
40         R = Areduced[i:i,: ] # k-th row
41         pivot = C[i];
42         C=C/pivot #normalize all entires by C[i]
43         U=[U;R];
44         L=[L C];
45         Areduced=Areduced-C*R;
46     end
47 end
48 myI = zeros(n,n)+I
49 P=myI[p,: ]
50 F = luStructReturn(L,U,P,p)
51 return F
52 end

```

Output

myLU (generic function with 2 methods)

We first repeat the previous example and see that $F.p = [3, 2, 1]$.

```

1 A = [2 2 3; 4 5 6; 7 8 9]
2 F = myLU(A)

```

Output

```

luStructReturn([1.0 0.0 0.0; 0.5714285714285714 1.0 0.0; 0.2857142857142857
-0.6666666666666666 1.0], [7.0 8.0 9.0; 0.0 0.4285714285714288
0.8571428571428577; 0.0 0.0 1.0], [0.0 0.0 1.0; 0.0 1.0 0.0; 1.0 0.0 0.0],
[3, 2, 1])

```

Then we do a huge example where being able to look at the indices, $F.p$, is easier than looking at the permutation matrix itself, $F.P$.

```

1 using Random
2 A=randn(100,70)
3 F=myLU(A)
4 @show norm(F.P*A-F.L*F.U)
5 F.p

```

Output

```
norm(F.P * A - F.L * F.U) = 5.482649156726496e-14
```

```

100-element Vector{Int64}:
 8
 3
52
51

```

37
88
26
18
49
55
10
76
61
.
.
.
11
25
63
54
93
69
95
41
20
98
66
45

B.2 Your First Named Tuple, an Alternative to Structured Returns

Alternative Method to Build a Structured Return via a Named Tuple

- build your function as before, and then
- modify the return statement as follows

```
1 # Now we build the structure  
2 F = (L=L, U=U, p=p, P=myI[p, :])  
3 return F
```

The above is called a “named tuple”.

- To be clear, **you skip the step**

```
1 # First, build the structure  
2 struct luStructReturn  
3 # List the variables to be used in the return statement  
4 # of the function  
5 L  
6 U  
7 P  
8 p  
9 end
```

Check out line #40 below.

```
1 # Start your function as before
```

```

2 using LinearAlgebra
3 #
4 function myLU (M::Array{<:Number, 2}, aTol=1e-12)
5 # Works for rectangular matrices
6     n, m = size(M)
7     k=min(n, m)
8     Areduced = deepcopy(M)
9     L = Array{Float64, 2}(undef, n, 0)
10    U = Array{Float64, 2}(undef, 0, m)
11    p=collect(1:n)
12    for i = 1:k
13        C = Areduced[:,i] # k-th column
14        R = Areduced[i:i, :] # k-th row
15        if maximum(abs.(C)) <= aTol #column of zeros
16            C=0.0*C; C[i]=1.0;
17            U=[U;R];
18            L=[L C];
19            Areduced=Areduced.-C*R;
20        else # put the biggest entry to the top
21            ii=argmax(abs.(C));
22            nrow=ii[1]
23            p[[i, nrow]]=p[[nrow, i]];
24            Areduced[[i, nrow], :]=Areduced[[nrow, i], :];
25            if i>1
26                L[[i, nrow], :]= L[[nrow, i], :];
27            end
28            C = Areduced[:,i] # k-th column
29            R = Areduced[i:i, :] # k-th row
30            pivot = C[i];
31            C=C/pivot #normalize all entieres by C[i]
32            U=[U;R];
33            L=[L C];
34            Areduced=Areduced-C*R;
35        end
36    end
37    myI = zeros(n, n)+I
38    P=myI[p, :]
39    # Now we build the structure
40    F = (L=L, U=U, p=p, P=myI[p, :])
41    return F
42 end

```

Output

myLU (generic function with 2 methods)

```

1 using Random
2 A=randn(20,13)
3 F=myLU(A)
4 @show norm(F.P*A-F.L*F.U)
5 F.p

```

Output

norm(F.P * A - F.L * F.U) = 3.090121509299927e-15

20-element Vector{Int64}:

```
8
4
2
10
7
17
11
19
14
12
18
1
13
9
15
16
6
5
3
20
```

```
1 F
```

Output

```
(L = [1.0 -0.0 ... 0.0 -0.0; 0.2592134812359471 1.0 ... 0.0 -0.0; ... ; -0.704296668486134 -0.0
```

```
1 # You can also call the function as follows
2 (L, U, p, P) = myLU(A)
3 P
```

Output

```
20-element Vector{Int64}:
 7
11
17
13
 8
 4
20
14
19
12
 2
 6
 5
10
15
16
 3
18
 9
 1
```

Exercise B.1 Build your own function called `build_a_named_tuple` which takes in 3 variables, and returns them in a named tuple with names "VarOne", "VarTwo", and "VarThree"

Solution

```
1 function build_a_named_tuple(one, two, three)
2     return (VarOne=one, VarTwo=two, VarThree=three)
3 end
```

Output

build_a_named_tuple (generic function with 1 method)

```
1 #autograder cell
2
3 F = build_a_named_tuple("temp", 2, pi)
4 T1 = (@assert F.VarOne == "temp")
5 T2 = (@assert F.VarTwo == 2)
6 T3 = (@assert F.VarThree == pi)
7 [T1 T2 T3]
```

Output

```
1×3 Matrix{Nothing}:
 nothing nothing nothing
```

You have total freedom in how you name the variables. Here we use X , Y , z , and Z instead of L , U , p , and P . Why? Just because we can. ■

```
1 # Start your function as before
2 using LinearAlgebra
3 #
4 function myLU(M::Array{<:Number, 2}, aTol=1e-12)
5 # Works for rectangular matrices
6     n, m = size(M)
7     k=min(n, m)
8     Areduced = deepcopy(M)
9     L = Array{Float64, 2}(undef, n, 0)
10    U = Array{Float64, 2}(undef, 0, m)
11    p=collect(1:n)
12    for i = 1:k
13        C = Areduced[:,i] # k-th column
14        R = Areduced[i:i, :] # k-th row
15        if maximum(abs.(C)) <= aTol #column of zeros
16            C=0.0*C; C[i]=1.0;
17            U=[U; R];
18            L=[L C];
19            Areduced=Areduced.-C*R;
20        else # put the biggest entry to the top
21            ii=argmax(abs.(C));
22            nrow=ii[1]
23            p[[i, nrow]]=p[[nrow, i]];
24            Areduced[[i, nrow], :]=Areduced[[nrow, i], :];
25            if i>1
26                L[[i, nrow], :]=L[[nrow, i], :];
27            end
28            C = Areduced[:,i] # k-th column
29            R = Areduced[i:i, :] # k-th row
30            pivot = C[i];
```

```

31         C=C/pivot #normalize all entieres by C[i]
32         U=[U;R];
33         L=[L C];
34         Areduced=Areduced-C*R;
35     end
36 end
37 myI = zeros(n,n)+I
38 P=myI[p, :]
39 # Now we build the structure
40 F = (X=L, Y=U, z=p, Z=myI[p, :])
41 return F
42 end

```

Output

myLU (generic function with 2 methods)

```

1 # Note that the variable names are set by X=, Y=, z=, Z=
2 F=myLU(A)

```

Output

```

(X = [1.0 -0.0 ... 0.0 -0.0; 0.2592134812359471 1.0 ... 0.0 -0.0; ... ;
-0.704296668486134 -0.011650553311418845 ... 0.698769089898198
0.7658109144351539; 0.017861627257789486 0.4287972223485453 ...
0.1941140899317409 -0.19326359666771908], Y = [-1.8260925655441076
0.5791501235532478 ... 0.4932420099150497 -0.5599493363829814; 0.0
-2.6564834565572952 ... 0.7925703023450414 0.8688637141207752; ... ; 0.0
-5.795161686936636e-17 ... 2.515380708071959 0.8004064587012474; 0.0
2.445820904052053e-16 ... 0.0 -2.4862928420239956], z = [7, 11, 17, 13, 8, 4,
20, 14, 19, 12, 2, 6, 5, 10, 15, 16, 3, 18, 9, 1], Z = [0.0 0.0 ... 0.0 0.0;
0.0 0.0 ... 0.0 0.0; ... ; 0.0 0.0 ... 0.0 0.0; 1.0 0.0 ... 0.0 0.0])

```

```

1 # Note that the variable names are set by X=, Y=, z=, Z=
2 F.z

```

Output

```

20-element Vector{Int64}:
 7
11
17
13
 8
 4
20
14
19
12
 2
 6
 5
10
15
16
 3
18

```

Should I Use Structured Returns or Named Tuples?

As with most things in programming, there are many ways to accomplish the same goal. You may, if you wish, simply return a bunch of values as we did before

```
return L,U,P,p
```

However, that can get messy, and if you forget the order of the values your function returns, your subsequent computations may be garbage. If you want to make a copy of the data and pass it to another function, such as one that solves a linear system of equations using forward substitution and back substitution, it can be simpler to just pass one variable, the structure. This is where `structs` come in handy. Hence, use them if you think they can help you, or if you just like the extra organization.

B.3 Tools to Solve Linear Equations

If you made it this deeply into the Lab Manual, you deserve to find something awesome. Here, we give you a means to solve all of the cases of $Ax = b$ that we have encountered in ROB 101.

```

1 #
2 function myLU (M::Array{<:Number, 2}, aTol=1e-12)
3 # Works for rectangular matrices
4     n, m = size(M)
5     k=min(n, m)
6     Areduced = deepcopy(M)
7     L = Array{Float64, 2}(undef, n, 0)
8     U = Array{Float64, 2}(undef, 0, m)
9     p=collect(1:n)
10    for i = 1:k
11        C = Areduced[:,i] # k-th column
12        R = Areduced[i:i, :] # k-th row
13        if maximum(abs.(C)) <= aTol #column of zeros
14            C=0.0*C; C[i]=1.0;
15            U=[U; R];
16            L=[L C];
17            Areduced=Areduced.-C*R;
18        else # put the biggest entry to the top
19            ii=argmax(abs.(C));
20            nrow=ii[1]
21            p[[i, nrow]]=p[[nrow, i]];
22            Areduced[[i, nrow], :]=Areduced[[nrow, i], :];
23            if i>1
24                L[[i, nrow], :]=L[[nrow, i], :];
25            end
26            C = Areduced[:,i] # k-th column
27            R = Areduced[i:i, :] # k-th row
28            pivot = C[i];
29            C=C/pivot #normalize all entires by C[i]
30            U=[U; R];
31            L=[L C];
32            Areduced=Areduced-C*R;
33        end
34    end

```

```

35 myI = zeros (n, n) + I
36 P=myI[p, :]
37 # Now we build the structure
38 F = (L=L, U=U, p=p, P=P)
39 return F
40 end
41
42 # LDLT named tuple
43 function myLDLT(A, aTol=1e-12)
44 M = A' * A
45 n, m = size(A)
46 Areduced = M
47 L = Array{Float64, 2}(undef, m, 0)
48 Id = zeros(m, m) + I
49 p=collect(1:m)
50 D=zeros(m, m)
51 for i=1:m
52 ii=argmax(diag(Areduced[i:m, i:m]))
53 mrow=ii[1] + (i-1)
54 if !(i==mrow)
55 p[[i, mrow]] = p[[mrow, i]]
56 Areduced[[i, mrow], :] = Areduced[[mrow, i], :]
57 Areduced[:, [i, mrow]] = Areduced[:, [mrow, i]]
58 end
59 if (i>1)
60 L[[i, mrow], :] = L[[mrow, i], :]
61 end
62 pivot=Areduced[i, i]
63 if !isapprox(pivot, 0, atol=aTol)
64 D[i, i]=pivot
65 C=Areduced[:, i]/pivot
66 L=[L C]
67 Areduced=Areduced - (C*pivot*C')
68 else
69 L=[L Id[:, i:m]]
70 break
71 end
72 end
73 diagD=diag(D)
74 P=Id[p, :]
75 return (L=L, P=P, D=D, diagD=diagD, p=p)
76 end
77
78 # This is a back substitution function. It solves for x in an equation Ux = b,
79 # where U is upper triangular. The function assumes U and b are the correct
80 # sizes. You can add error checking, if you wish.
81 function backwardsub(U, b)
82 # Check if U is non-singular
83 min_diagU = minimum(abs.(diag(U)))
84 if min_diagU < 1E-10
85 return false
86 end
87 n = length(b)
88 x = Vector{Float64}(undef, n)
89 x[n] = b[n]/U[n, n]
90 for i in n-1:-1:1
91 x[i] = (b[i] - (U[i, (i+1):n])' * x[(i+1):n]) ./ U[i, i]

```

```

92     end
93     return x
94 end
95
96 # This is a forward substitution function. It solves for x in an equation Lx = b,
97 # where L is lower triangular. # The function assumes L and b are the correct
98 # sizes. You can add error checking, if you wish.
99 function forwardsub (L, b)
100     # Check if L is non-singular
101     min_diagL = minimum (abs. (diag (L)))
102     if min_diagL < 1E-10
103         return false
104     end
105     n = length (b)
106     x = Vector{Float64} (undef, n);
107     x[1] = b[1]/L[1,1]
108     for i = 2:n
109         x[i] = (b[i] - (L[i, 1:i-1])' * x[1:i-1]) ./ L[i,i]
110     end
111     return x
112 end
113
114 function num_independent_vectors (A, aTol=1e-10)
115     F = myLDLT (A)
116     indicesBigEnough=findall (x->x>aTol, abs. (F.diagD))
117     return length (indicesBigEnough)
118 end
119
120 function grahm_schmidt (U, aTol=1e-8)
121     # Allows columns of U to be dependent
122     nRowsU = size (U, 1) # the 1 returns the number of rows
123     nColsU = size (U, 2) # the 2 returns number of columns.
124     #
125     # We create an empty matrix V that will hold all of the orthonormal vectors
126     V = Array{Float64, 2} (undef, nRowsU, 0)
127     #
128     #start a for loop that runs the number of times that there are columns in U
129     for k in 1:nColsU
130         uk = U[:, k]
131         # next we set up for the general step of the G-S process
132         vk = copy (uk)
133         for i = 1:size (V, 2) # loop over the number of columns of V
134             vi = V[:, i]
135             vk = vk - ( dot (uk, vi) / dot (vi, vi) ) * vi
136         end
137         #
138         if norm (vk) > aTol # Trick? Not really. We only add a normalized vk
139             V = [V vk/norm (vk)] # if vk is not approximately the zero vector. Make
sense?
140         end # And we ignore the vector otherwise
141     end
142     #
143     rankU = size (V, 2)
144     return (Q=V, r=rankU)
145 end
146
147 function myQR (A)

```

```

148 nRowsA, nColsA = size(A)
149 GS = graham_schmidt(A)
150 Q = GS.Q
151 R = Q' * A
152 Flag = ( nColsA == size(Q,2) ) # Is A full column rank?
153 F = (Q=Q, R=R, Flag=Flag)
154 return F
155 end

```

Output

myQR (generic function with 1 method)

The magical super solver for $Ax = b$!

```

1 function mySolve(A, b, myMethod = "LU")
2   nRowsA, nColsA = size(A)
3   dim1 = num_independent_vectors(A)
4   if ( (nRowsA == nColsA) & (dim1 == nColsA) ) # A is square and invertible
5     if myMethod == "LU"
6       F = myLU(A)
7       y = forwardsub(F.L, F.P*b)
8       x = backwardsub(F.U, y)
9     else # Assume QR is ok!
10      F = myQR(A)
11      x = backwardsub(F.R, (F.Q)' * b)
12    end
13    Flag = "A is square invertible"
14  elseif (dim1 == nColsA) # A has linear indep columns, but is not square
15    # Regression style solution
16    M = A' * A
17    b = A' * b
18    F = myQR(M)
19    x = backwardsub(F.R, (F.Q)' * b)
20    Flag = "A has indep columns, not square"
21  elseif (dim1 == nRowsA) # A has linear indep rows, but is not square
22    # x of min norm satisfying the equation
23    F = myQR(A')
24    beta = forwardsub(F.R', b)
25    x = F.Q*beta
26    Flag = "A has indep rows, not square"
27  else # dependent rows and columns
28    # Ax=b <=> QR x = b => R x = Q' * b <=> x = R' * z and R * R' z = Q' * b
29    G = myQR(A); R = G.R; Q = G.Q
30    F = mySolve(R * R', Q' * b) # function calls itself!
31    z = F.x
32    x = (R') * z
33    if (num_independent_vectors([A b]) == dim1)
34      Flag = "b in column span of A, neither columns nor rows of A indep"
35    else
36      Flag = "b not in column span of A, neither columns nor rows of A indep"
37    end
38  end
39  return (x=x, Flag = Flag)
40 end

```

Output

mySolve (generic function with 2 methods)

```
1 A = randn(5,5)
2 b = randn(5,1)
3 F = mySolve(A, b)
```

Output

```
(x = [-0.25142805928052786, 0.0026859440179519173, 0.9036926694667949,
-0.22222812261095745, 0.7299120656135045],
Flag = "A is square invertible")
```

```
1 norm(A*F.x-b)
```

Output

```
2.7230185981536845e-16
```

```
1 A = randn(5,5)
2 b = randn(5,1)
3 F = mySolve(A, b)
```

Output

```
(x = [-0.7170921127391888, 1.7999232745022158, 0.016231490871714712,
-0.6053674054406761, -0.12624083607954703],
Flag = "A has indep columns, not square")
```

```
1 norm(A*F.x-b)
```

Output

```
8.524389570411963e-13
```

```
1 A = randn(6,5)
2 b = randn(6,1)
3 F = mySolve(A, b)
```

Output

```
norm(A'*A*F.x-A'*b)
```

```
1 A = randn(3,5)
2 b = randn(3,1)
3 F = mySolve(A, b)
```

Output

```
(x = [0.10287772187194529, 0.10648005509200273, -0.31426106675440746,
0.2114509321783164, 0.43636854425759064],
Flag = "A has indep rows, not square")
```

```
1 norm(A*F.x-b)
```

Output

Our final cases are the “trickiest”, $Ax = b$ when both the rows and columns of A are linearly dependent. If $b \in \text{col span}\{A\}$, there then does exist a solution. Because the columns of A are dependent, that solution would not be unique. Can we still find a solution of minimum norm? And what if $b \notin \text{col span}\{A\}$?

Solving $Ax = b$ when neither Rows nor Columns of A are Linearly Independent

Consider $Ax = b$ where A is $n \times m$. We assume, that the rows and columns of A are both linearly dependent and hence our normal means of finding a solution do not apply. What do we do?

We can still use our version of the Gram-Schmidt Algorithm that works for linearly dependent vectors to perform a QR Factorization, namely, $A = Q \cdot R$, where Q is an orthonormal matrix, meaning $Q^T \cdot Q = I$. Then Q is $n \times k$, where $k < \min(n, m)$, and R is then $k \times m$. Hence, we have

$$\begin{aligned} \|Ax - b\|^2 &= \|Q \cdot Rx - b\|^2 \\ &= (x^T R^T \cdot Q^T - b^T)(Q \cdot Rx - b) \\ &= x^T R^T \cdot Q^T \cdot Q \cdot Rx - 2x^T R^T \cdot Q^T b + b^T b \\ &= x^T R^T \cdot I_k \cdot Rx - 2x^T R^T \cdot Q^T b + b^T b \\ &= b^T b - \bar{b}^T \bar{b} + (x^T R^T - \bar{b})(Rx - \bar{b}) \\ &= (b^T b - \bar{b}^T \bar{b}) + \|Rx - \bar{b}\|^2 \end{aligned}$$

where $\bar{b} = Q^T b$, the orthogonal projection of b onto the column span of A . Because k is the rank of A , it follows from Chapter 10.5 of our textbook that the rows of R are linearly independent. Because the rows of R are linearly independent, $Rx = (Q^T)b$ is an underdetermined equation and we can apply known results for its solution, namely

$$x^* = \arg \min_{Rx=Q^T b} \|x\|^2 \iff (R \cdot R^T z = Q^T b \text{ and } x^* = R^T z),$$

where $R \cdot R^T$ is square and invertible. We solve the right-hand side of the above equation using the first case covered by our function `mySolve(A, b)`. It's quite cool that we can complete a function by making a call to itself!

We note that once we choose x such that $Rx - \bar{b} = Rx - Q^T b = 0$, then

$$\|Ax - b\|^2 = 0 \iff (b^T b - \bar{b}^T \bar{b}) = 0.$$

In other words, b is in the column span of A if, and only if, $(b^T b - \bar{b}^T \bar{b}) = 0$.

Next we test all of this out.

We first test the case that $b \in \text{col span}\{A\}$.

```
1 A = randn(5, 4)
2 b = A[:, 3] + A[:, 1] + pi * A[:, 2] # linear combo of columns of A
3 A = [A[:, 1:3] A[:, 2]] # make columns of A dependent
4 F = mySolve(A, b)
```

Output

```
(x = [1.0, 1.5707963267948961, 0.9999999999999997, 1.5707963267948961],
Flag = "b in column span of A, neither columns nor rows of A indep")
```

```
1 norm(A * F . x - b)
```

Output

5.756054031998179e-15

Next we check when $b \notin \text{col span}\{A\}$.

```
1 A = randn(5,4)
2 b = rand(5,1)
3 A=[A[:,1:3] A[:,2]]
4 F = mySolve(A, b)
```

Output

```
(x = [0.041439798464912705, -0.13327553074344564, -0.37488456666129705,
-0.13327553074344564], Flag = "b not in column span of A, neither columns nor
rows of A indep")
```

```
1 norm(A*F.x-b)
```

Output

```
0.9980756059006674
```

```
1 G=myQR(A)
2 bb=G.Q'*b
3 (b'*b - bb'*bb)^.5
```

Output

```
1×1 Matrix{Float64}:
 0.9980756059006671
```

We note that, as predicted by our theory, $\text{norm}(A \cdot F \cdot x - b) = (b' \cdot b - bb' \cdot bb)^{0.5}$. We have computed the x of smallest norm that minimizes the norm squared error of $Ax - b$ when A has neither independent columns nor independent rows. That's pretty incredible, actually!

Appendix C

From MATLAB to Julia

Command Comparison Side by Side:

The following document takes you through a nice comparison of MATLAB and Julia, with Python thrown in as a bonus.

<https://cheatsheets.quantecon.org/>

If you know Matlab, there are numerous places where Julia is just enough different to cause significant frustration. If you read the above document carefully and refer to it regularly, you will be able to pick up Julia quite quickly and with minimal frustration.

Here are some key things to note.

- When using Julia in a jupyter notebook, only the result of the last computation in the cell is printed to the workspace. You do NOT need to place semicolons after each calculation.
- Julia uses spaces to separate elements in a row vector or a row of a matrix and NOT commas as in MATLAB. Julia uses commas OR semicolons to separate rows in a vector or matrix, and NOT only semicolons as in Matlab; see Appendix A.1.
- Julia uses square brackets when accessing elements of a vector or matrix, whereas Matlab uses parentheses. Julia $v = A[1, 4]$, whereas Matlab $v = A(1, 4)$; see Appendix A.1.
- Julia is a bit strange when extracting a row from a matrix. Julia $v = A[2:2, :]$, whereas Matlab $v = A(2, :)$. Note the $2:2$; this is only required when you want the extracted row to remain a row. The Julia command $v = A[2, :]$ will extract the second row of the matrix A , but it will be presented to you as a column vector; see Appendix A.2.
- Column extraction is the same in Julia and Matlab, barring the change from parentheses to square brackets.
- Julia is heavily Typed and hence a $1 \times n$ (row) vector times an $n \times 1$ (column) vector is a 1-element vector and not a scalar as in Matlab. You have to extract the element from the 1-element vector if you want to use it as a number; see Appendix A.3.
- `for` loops, `while` loops, and `if` statements can be done the same way in Julia and Matlab, though Julia has a few extra means of specifying the range of a counter when doing `for` loops; see Appendix A.5.
- The syntax for functions in Julia is different from the syntax in Matlab, but not so much that it makes it hard to learn or anything. The main difference is how one specifies the output arguments of a function. Julia uses a `return` statement while Matlab places them on the first line of the function's specification; see Appendix A.6.
- Julia does not have an equivalent of Matlab's `eye(n, m)` command; see Appendix A.1.
- The error messages in Matlab are quite useful, in general, and rather easy to understand. The error messages in Julia are currently a mess. Julia 2.0 is supposed to fix the error messages. We are currently at Julia 1.6.1. Oh well!
- Plotting is relatively easier in Matlab than in Julia.

- Julia gives you the speed of compiled code. Julia can be as fast as C++. Matlab? Not so much.
- In MATLAB, every package that you have purchased from the MATHWORKS is immediately available to you. Julia tries to avoid being a memory hog. You have to add packages when you need them. In ROB 101, a bunch of packages have already been added, but you have to learn the `using` command to activate them. See this manual for examples.

Appendix D

From C++ to Julia

C++ vs Julia: What are the differences?

What is C++?

- has imperative, object-oriented and generic programming features.
- allows low level memory manipulation.
- compiles directly to a machine's native code: allowing it to be one of the fastest languages in the world.

What is Julia?

- a high-level, high-performance dynamic programming language for technical computing.
- has syntax that is familiar to users of other technical computing environments.
- has an extensive mathematical function library.

This document shows some key comparisons between C++ and Julia.

https://docs.google.com/spreadsheets/d/1bU8V81PG2kHo1k7M_ylykijzLvFGvw1CEPaZHqTTk8/edit?usp=sharing

Julia is a very high level programming language, unlike C++. Julia provides a lot of mathematical functions that C++ does not have. Hence, learning Julia with a C++ foundation is very helpful. There are, however, some critical differences between their syntax that can make it hard to go from one language to the other when you are first learning them. In the above spreadsheet, we compared how to accomplish a few key tasks in C++ and Julia. Please share with us in Piazza other commands in C++ that you wish to execute in Julia; we'll try to add them to our spreadsheet.

Appendix E

Creating Your Own Local Julia Installation

Warning!

We provide this information as a service. We do not have enough staff to provide IT support for your personal Julia installations. Now, some of your fellow classmates are probably able and happy to help you. Post on Piazza for help, but do not expect ROB 101 Staff to respond unless they have a free moment.

There is basic information on the F-21 ROB 102 webpage: https://robotics102.github.io/tutorials/setup.html#install_julia?

Professor Grizzle strongly recommends Anaconda as the user interface https://julia.quantecon.org/getting_started_julia/getting_started.html. It allows you to work with the same jupyter notebooks we use in ROB 101. Of course, you will not be able to directly download Labs, HWs, or Projects from the ROB 101 Canvas Site, nor will you be able to submit assignments through your personal installation. Please keep these points in mind.