Debugging

Reynald Oliveria

May 2, 2022

0.1 Preface 2
0.2 Where do bugs come from? 2
  0.2.1 Syntax Errors and Runtime Errors 2
    0.2.1.1 Syntax Errors 2
    0.2.1.2 Runtime Errors 4
  0.2.2 Logical Errors 4
    0.2.2.1 Failures in reasoning 4
    0.2.2.2 Incorrect assumptions 6
    0.2.2.3 Misunderstood functionality 7
0.3 General Debugging Techniques 8
  0.3.1 Some common bugs 8
    0.3.1.1 Index out-of-bounds 8
    0.3.1.2 Never-terminating loop 9
  0.3.2 Displaying values mid-run 10
  0.3.3 Code testing 14
0.4 Debugging in MATLAB 16
  0.4.1 Errors from Linear Algebra 16
  0.4.2 MATLAB Arrays 17
    0.4.2.1 Arrays as function parameters 17
    0.4.2.2 Cell arrays 18
  0.4.3 Machine Precision 19
    0.4.3.1 Error propagation 21
    0.4.3.2 Catastrophic Cancellation 22
    0.4.3.3 Condition number 22
  0.4.4 Debugging Tools in MATLAB 22

0.1 Preface

Unlike other chapters, this chapter's main focus is programming rather than some application of mathematical concepts. If you are reading this, you probably are taking a MATH or AMSC course that requires you to program. These courses typically do not have a programming course as a prerequisite. As it is a math course, time will probably not be spent much on programming. But, good programming and bad programming can be the difference in getting the right and wrong answer. So, it's worth taking a look at how we can reduce bad programming.

This chapter will go over general debugging techniques, as well as the tools available to help you debug. Section 2 will go over the different types and sources of bugs. Section 3 will be about general debugging techniques applicable to most programming languages. These sections can probably be skimmed if you have already taken another programming course. Most math courses that require programming use MATLAB, and so, examples will be written in MATLAB. Further, there will be a special focus in sections 2 and 3 on bugs that are more prevalent in MATLAB than in other languages.

Because MATLAB is specially designed for numerical computation, and numerical computation draws many techniques from concepts learned in Linear Algebra and Calculus II, this chapter will assume knowledge of those subjects. Sections 4 and 5 will take a closer look at bugs that are specific to MATLAB, as well as tools builtin to the MATLAB environment that can help with debugging.

Note that this chapter is **not** an introduction to MATLAB. It will be specifically be about identifying bugs, and getting your programs to work. As such, some familiarity with MATLAB is assumed in this chapter. Specifically, the MATLAB covered in the MATLAB Onramp Lecture series [2] will be assumed as familiar. A more comprehensive introduction to MATLAB is also available with the MATLAB Fundamentals course [3].

0.2 Where do bugs come from?

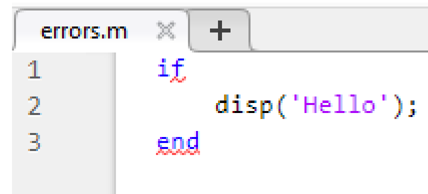
0.2.1 Syntax Errors and Runtime Errors

Strictly speaking, we can classify programming errors detected by a machine into two: **compile-time errors** and **runtime errors**. Compile-time errors occur during the translation of the code you write to a code that the machine understands. The one class of compile-time errors that are relevant to us are **syntax errors**.

0.2.1.1 Syntax Errors

Syntax errors are errors that are akin to grammatical errors in English. It occurs when a programmer types a piece of code in such a way that a pattern is not

recognizable by the machine. For example, take the sentence "The dog small ran away." As humans, we can probably figure out that "small" refers to the dog. But a machine would first check the grammar of the sentence, realize that it does not expect "small" to occur where it does, and just throw out an error. It will not even try to understand it. Let's take a look at this example in the MATLAB environment.

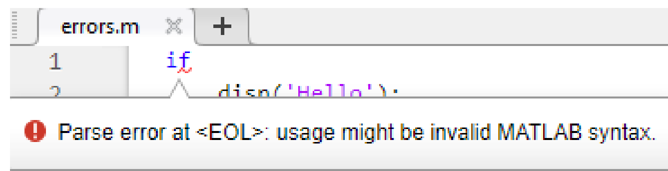


We see here that the `if`-statement is malformed. The is no condition to this `if`-statement. This is like saying "If, then say 'Hello'." This is wrong grammar: If what? And precisely, when we try to run this code, we get:

```
>> errors
Error: File: errors.m Line: 1 Column: 3
Invalid expression. Check for missing or extra characters.
```

From the message, it finds an error in line 1 and column 3. This means that on the first line, there is an error starting on the third character. And indeed, we see that it is precisely on the first line at the third character where we are missing a condition.

However, if we look at the code from above, we see that the `f` in the word `if` is underlined in red. And precisely after that character, MATLAB finds an error when we run the code. So let's hover over this red underline, and see what happens.



The floating textbox tells us that there is a "Parse error" at "<EOL>." A "Parse error" refers to how the machine will not parse this piece of code correctly which is a syntax error. "<EOL>" means End of Line. So, the textbox is telling us to look at the end of line for a syntax error. This is precisely what MATLAB also tells us when we try to run the code.

The machinery that makes this underline, and adds this floating textbox, is known as the MATLAB Code Analyzer [6]. More documentation can be found in this website. There is not much content to syntax errors to learn except for identifying them. Learning to fix syntax errors is learning the syntax of the language, and I refer you again to the Onramp lecture series to do that. And identifying them is made easy as MATLAB or the Code Analyzer tells you precisely where syntax errors occur.

0.2.1.2 Runtime Errors

Runtime errors are errors that occur when the code is running. These errors are akin to sentences that do not make sense even though they are grammatically correct. For example, "More people have been to Berlin than I have." Let's look at this example.

```
>> A = zeros(2);
>> A(2,4)
Index in position 2 exceeds array bounds. Index must not exceed 2.
```

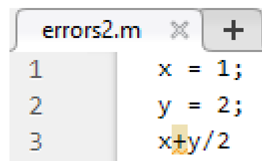
Here we are trying to access the element in the second row and fourth column of a matrix with only two rows and two columns. This error is definitely contrived, but we will discuss other ways that runtime errors can occur. We are less interested in the idea of a runtime errors, rather, we are interested in how deeper errors manifest as runtime errors.

0.2.2 Logical Errors

Logical errors are bugs that causes a program to function in a way that is not intended or not function at all. They are errors that born from a failure in the logic of the programmer rather than an ignorance of the grammar of the language itself. We can broadly classify logical errors into three types: **failures in reasoning**, **incorrect assumptions**, and **misunderstood functionality**.

0.2.2.1 Failures in reasoning

A way in which a program might not function correctly is if a programmer reasons incorrectly as they code. Perhaps a mathematical concept is forgotten, like here.



In the program above, the programmer attempts to take the average of two numbers stored in the variables `x` and `y`. But, because multiplication and division are applied before addition, it turns out that the program does not calculate the average. Instead, it calculates the sum of `x` and half of `y`.

```
>> errors2

ans =

     2
```

We can see this when we try to find the average of 1 and 2. We see that the program outputs that the average is 2, but the correct average is 1.5

Let's take a look at a slightly more complicated example. Let's say that we are interviewing witnesses of a crime. And we are trying to figure out precisely how many people were in the crime scene when the crime happened. So, we interview two witnesses as asked them to name all the people that they saw. We know at most 100 people were in the crime scene.

We give the lists to a programmer. The list of people given to us by the first witness is stored in `list1`, and the list by the second is stored in `list2`. The programmer then writes this piece of code to find the total number of people present in the crime scene.

```
totalNumberOfPeople.m
1  function total = totalNumberOfPeople(list1, list2)
2  total = length(list1) + length(list2);
3  end
```

It seems sensible enough; the answer should just be the number of people in the first list added to the number of people in the second. But then, when we try to run it:

```
>> totalNumberOfPeople(list1, list2)

ans =

   180
```

We get 180? But, we knew that there are at most 100 people in the crime scene. So, did the witnesses lie to us? The programmer takes a closer look at the lists to see what is going on. And this is what the programmer sees:

`list1`	`list2`
Liam	Liam
Noah	Noah
Kai	Oliver
Oliver	Elijah
Elijah	Williams
⋮	⋮

Aha! We see that of course that some people would be listed by both witnesses. By simply adding the lists together, the programmer is causing the program to double count the people both witnesses listed.

A key thing to note in these examples is that the machine ran the program just fine. We were only able to detect the error by inspecting it ourselves. And the way we did it, is by putting inputs and seeing if they make sense. In other words, we tested the code by trying out inputs for which we know something about the outputs. We will discuss this useful debugging technique of **code testing** in a later section.

0.2.2.2 Incorrect assumptions

Programs are often written for some real-life application. Sometimes, the computations that need to be done are needed by a non-programmer client. And so, they might hire a programmer to write a piece of code for them to compute the things they need computed. However, the subject (domain) the client wants to study using the computations might not be familiar to the programmer.

For example, suppose an economist hires a programmer to write a program for them. The economist wants to study if there is a difference in the performance of the stock market performance on days when it rains or snows versus when it does not. The economist instructs the programmer to see if there is a substantial difference in the performance of stock market on rainy/snowy days versus days where there is no precipitation.

The programmer crafts two lists. The `stock_per` list is a list of the daily returns of a stock over a one-year period. The `did_it_rain` list is a 365-long list with 1 as the n-th entry if it rained or snowed on day n of that year, and 0 otherwise. The programmer then tries to calculate the average performance for non-rainy days and rainy days with this piece of code.

```
sunny_day_avg = 0;
sunny_day_count = 0;
rainy_day_avg = 0;
rainy_day_count = 0;
for i = 1:365
    if did_it_rain(i) == 0
        sunny_day_avg = sunny_day_avg + stock_per(i);
        sunny_day_count = sunny_day_count + 1;
    else
        rainy_day_avg = rainy_day_avg + stock_per(i);
        rainy_day_count = rainy_day_count + 1;
    end
end

sunny_day_avg = sunny_day_avg/sunny_day_count
rainy_day_avg = rainy_day_avg/rainy_day_count
```

When we run this piece of code we get a runtime error, saying that we are trying to access an element beyond the length of the `stock_per` list.

```
>> stock_rain_correlation
Index exceeds the number of array elements. Index must not exceed 252.

Error in stock_rain_correlation (line 10)
        sunny_day_avg = sunny_day_avg + stock_per(i);
```

This error occurs when we try to read the 253rd element. Why then does the `stock_per` list only have 252 days? Well, the stock market does not trade on weekends and some holidays. And thus, there will not be daily returns data for all 365 days in a year. In this case, we see that because of the programmer's lack of understanding about the stock market, the programmer incorrectly assumed that every day is a trading day.

0.2.2.3 Misunderstood functionality

Instead of making a false assumption because of a lack of understanding about an outside subject, a programmer might not fully understand a feature of a programming language. For example, a programmer is tasked with figuring out what is the highest average daily patrons of the busiest among three restaurants. Because the programmer heard that MATLAB is good at dealing with lists of numbers, she chose to use it.

In the matrix `A`, each row represents a restaurant. The entries of each row represents the number of patrons the restaurant had. For example, the element at

the 2nd row and 4th column of `A` is the number of patrons the second restaurant had on the fourth day of the study. To accomplish her task, the programmer finds the maximum among the average daily patrons of the three restaurants with this code:

```
A = [2 21 12 22 2 21 2;
     26 23 20 1 3 10 14;
     29 23 6 9 25 29 12];

max(mean(A))
```

When we run this code we get:

```
>> busy_restaurant

ans =

   22.3333
```

From the output, it does not seem suspicious. But, the `mean` function, takes averages column-wise. The programmer wrongly understood the `mean` function to be taking averages row-wise. So the output actually is the average number of patrons across the three restaurants on the busiest day. This is different from what we wanted: the average number of daily patrons in the busiest restaurant.

We should be worried about this example as we see that the output does not give us anything to worry. Further, MATLAB is not throwing an error. How are we supposed to find this error? We will get back to this example in the section on general debugging techniques. A great way to understand the precise functionality of MATLAB feature is via their official documentation. To access the documentation of a function, you can use `doc <function-name>` in the command window [8].

0.3 General Debugging Techniques

0.3.1 Some common bugs

In this section, we will explore bugs that are both prevalent in MATLAB and in other programming languages. However, we will focus on the bugs that are more relevant to MATLAB and the MATLAB environment.

0.3.1.1 Index out-of-bounds

An index out-of-bounds error occurs when a program is told to access the n-th element of a list which has less than n elements. We've seen this error earlier

as our example of a runtime error. And we see this error again as an example of false assumptions leading to errors. As we have seen, this error manifests as a runtime error with MATLAB alerting us that it happened.

0.3.1.2 Never-terminating loop

As the name suggests, this is a bug that causes a loop to theoretically never end. Sometimes, this bug does not manifest as a runtime error. For example, suppose a programmer is tasked to project how long a loan will be paid off if someone pays only $25 a month. The programmer writes the code below.

```
remaining_balance = 1000;
interest_rate = 0.01;
payment = 25;
months = 0;

while remaining_balance ~= 0
    remaining_balance = remaining_balance * (1 + interest_rate);
    remaining_balance = remaining_balance - payment;
    months = months + 1;
end
```

The issue with this code is that the condition for the `while` loop to end is for the remaining balance to be 0. However, because at month 51, the balance is less than $25, the calculated remaining balance will never be 0. Instead, it will be negative. And since the balance does not equal 0, the program keeps going and the remaining balance keeps becoming more and more negative.

The way a programmer might realize that they have made a mistake is by realizing that a computation should not take as long as the program has been running. Then the programmer can find this mistake by having the program display the balance at each iteration of the loop. In this way, they will see that the remaining balance never actually becomes 0. The programmer then corrects their code to be:

```
remaining_balance = 0;
interest_rate = 0.01;
payment = 25;
months = 0;

while remaining_balance > 0
    remaining_balance = remaining_balance * (1 + interest_rate);
    remaining_balance = remaining_balance - payment;
    months = months + 1;
end
```

Sometimes, if a loop involves storing things into memory on each iteration, a never-terminating loop might cause a memory error. In that, by having a loop that never ends just write values into memory, the memory available to the program might run out. For example, if the programmer is tasked to also store the projected history of balances and they write:

```
remaining_balance = 1000;
interest_rate = 0.01;
payment = 25;
months = 0;
balance_history = [];

while remaining_balance ~= 0
    remaining_balance = remaining_balance * (1 + interest_rate);
    remaining_balance = remaining_balance - payment;
    months = months + 1;
    balance_history(months) = remaining_balance;
end
```

We see that by running the program long enough, the program eventually throws an error saying the MATLAB has used up all the memory available to it.

```
>> how_many_months
Requested 42838447x1 (0.3GB) array exceeds maximum array size preference (0.3GB). This might cause MATLAB to become
unresponsive.

Error in how_many_months (line 11)
    balance_history(months) = remaining_balance;
```

0.3.2 Displaying values mid-run

MATLAB has a very useful feature. It will display the output of a computation or variable assignment if there is no semicolon at the end of the line. In this way, the programmer earlier with their never-terminating loop may realize this error by modifying their code to be:

```
remaining_balance = 1000;
interest_rate = 0.01;
payment = 25;
months = 0;

while remaining_balance ~= 0
    remaining_balance = remaining_balance * (1 + interest_rate);
    remaining_balance = remaining_balance - payment
    months = months + 1;
end
```

The equal sign that is highlighted in the above code is warning us that the output is not going to be suppressed. But, the output being displayed is exactly what we want to debug. Then we see the code displays:

```
>> how_many_months

remaining_balance =

   985


remaining_balance =

  969.8500


remaining_balance =

  954.5485

        ⋮

remaining_balance =

   33.0523


remaining_balance =

    8.3828


remaining_balance =

  -16.5334
```

From the negative value of the remaining balance, the programmer may find their mistake in the condition on their while loop. This rudimentary technique can be a quite powerful tool for debugging. Suppose instead that the programmer sought to find the difference in how long a loan is paid off with $25 dollar loan payments versus $40 dollar loan payment. The programmer then writes this program with two never-terminating loops:

```
remaining_balance = 1000;
interest_rate = 0.01;
payment = 25;
months = 0;

while remaining_balance ~= 0
    remaining_balance = remaining_balance * (1 + interest_rate);
    remaining_balance = remaining_balance - payment;
    months = months + 1;
end

month_count_25 = months;

remaining_balance = 1000;
interest_rate = 0.01;
payment = 40;
months = 0;


while remaining_balance ~= 0
    remaining_balance = remaining_balance * (1 + interest_rate);
    remaining_balance = remaining_balance - payment;
    months = months + 1;
end

month_count_40 = months;
```

The programmer then realizes that the program is taking longer than they should. The programmer suspects a never-terminating loop, but there are two loops. And so, to check, the programmer sets up "checkpoints" in a program to see if the program ever reaches certain points of the code, like so:

```
remaining_balance = 1000;
interest_rate = 0.01;
payment = 25;
months = 0;

while remaining_balance ~= 0
    remaining_balance = remaining_balance * (1 + interest_rate);
    remaining_balance = remaining_balance - payment;
    months = months + 1;
end

disp("checkpoint")

month_count_25 = months;

remaining_balance = 1000;
interest_rate = 0.01;
payment = 40;
months = 0;


while remaining_balance ~= 0
    remaining_balance = remaining_balance * (1 + interest_rate);
    remaining_balance = remaining_balance - payment;
    months = months + 1;
end

disp("checkpoint")

month_count_40 = months;
```

The output after three minutes of running is then:

```
>> months_difference
```

No checkpoint is displayed. From here, the programmer determines that the first loop seems to run forever, and they can go ahead in debug that. The idea of using code checkpoints can be useful even if we are not dealing with never-terminating loops. In practice, we could use these checkpoints after every nontrivial calculation, that way, we are able to pinpoint inaccuracies in the calculations.

0.3.3 Code testing

As we saw, some errors do not manifest until we analyze a known output given an input. For these errors, we must test the code to even know that an error exists. And with the same idea as having "checkpoints" in our code, we should also be testing as we go. When we finish writing a piece of code to compute a non-trivial calculation, we ought to test it before we continue. Typically, the code we will write depends on the code that we have written. And so, by *testing our code as we write our code*, it makes it easier to find bugs. Otherwise, there is simply more code to look through to find where bugs occur.

A pair of input and known output is a **test case**. We use test cases to debug code and ensure accuracy of the calculations done by our code. We plug in our input and compare the code output to our known output. Usually, we must make our own test cases. And so, an ideal test case would be where the input is so that calculations can be done reasonably easily by hand. For example, recall the buggy program that attempts to calculate the average number of daily patrons in the busiest restaurant:

```
A = [2 21 12 22 2 21 2;
     26 23 20 1 3 10 14;
     29 23 6 9 25 29 12];

max(mean(A))
```

Recall that the rows of the matrix `A` represent a restaurant, and the columns represent days. And so, to see the number of patrons restaurant 3 had on day 5, one would look at the element on the third row and fifth column. The error here is that the programmer thought that the `mean` function took averages row-wise, but instead, it takes averages column-wise.

Since we are trying to calculate the average number of daily patrons in the busiest restaurant, we ought to construct an input where:

1. We know which restaurant is the busiest restaurant
2. We know the average number of daily patrons of that restaurant

And so, for a test case, the matrix A can be constructed like so:

$$A = \begin{bmatrix} 0 & 0 & 0 & \cdots & 0 \\ 1 & 1 & 1 & \cdots & 1 \\ 2 & 2 & 2 & \cdots & 2 \end{bmatrix}$$

In this test case, it is clear that restaurant three is the busiest restaurant. Further, it clearly averages 2 patrons everyday. When we plug this matrix A as our test case, we see that the output is 1. This test case should cause the programmer some suspicion. Perhaps, they rewrite their code to be:

```
A = [0 0 0 0 0 0 0;
     1 1 1 1 1 1 1;
     2 2 2 2 2 2 2];

means = mean(A)
max(means)
```

Insodoing, the programmer introduced "checkpoints" in their programs. The output is then:

```
>> busy_restaurant

means =

     1     1     1     1     1     1     1


ans =

     1
```

It is now clear that the average is taken column-wise instead of row-wise. We see this from the fact that `means` has 7 columns instead of having 3 rows. After investigating the documentation of the `mean` function [10], the programmer figures that the code can be made to function correctly with this correction:

```
A = [0 0 0 0 0 0 0;
     1 1 1 1 1 1 1;
     2 2 2 2 2 2 2];

max(mean(A,2))
```

Another set of test-cases that ought to be investigated are **edge cases**. An edge case is an extreme input, but still within the realm of possibilities. For example, take the "corrected code" from when the programmer was trying to find the balance history of a loan when the borrower pays $25 monthly payments:

```
remaining_balance = 24;
interest_rate = 0.01;
payment = 25;
months = 0;
balance_history = [];

while remaining_balance > 0
    remaining_balance = remaining_balance * (1 + interest_rate);
    remaining_balance = remaining_balance - payment;
    months = months + 1;
    balance_history(months) = remaining_balance;
end

balance_history
```

An edge case would be if the loan amount was less than what is paid monthly. Then, the program outputs.

```
>> find_balance_history

balance_history =

   -0.7600
```

I will leave it as an exercise for the reader to figure out how to correct for this edge case.

0.4 Debugging in MATLAB

0.4.1 Errors from Linear Algebra

Because MATLAB uses matrices as the default datatype for 2-dimensional arrays, we must be wary about the rules surrounding doing calculations with matrices. In fact, every array with numerical content is treated as a matrix. The key thing to remember is the rule for matrix multiplication. Recall that an $m \times n$ matrix, that is, a matrix with m rows and n columns, can only be multiplied to another matrix with dimensions $n \times k$. And so, uniquely, we get dimension errors in MATLAB like so:

```
>> A * B
Error using  *
Incorrect dimensions for matrix multiplication. Check that the number of columns in the first matrix matches the
number of rows in the second matrix. To perform elementwise multiplication, use '.*'.
```

Further, note that some functions that you may encounter can be based on matrix multiplication. For example, the `quadprog` function solves a quadratic programming problem. From your linear algebra course, you would know that a quadratic programming problem can be of the form of finding vector $\mathbf{x}$ for matrices $\mathbf{H}$ and vector $\mathbf{f}$ to minimize:

$$\frac{1}{2}\mathbf{x}^{\mathsf{T}}\mathbf{H}\mathbf{x} + \mathbf{f}^{\mathsf{T}}\mathbf{x}$$

We see that matrix multiplication is involved, and so we must be wary about the dimensions of the matrices we input into the `quadprog` function. Further, as we saw the `*` operator does matrix-multiplication instead of element-wise multiplication. This can be quite confusing as `+` and `-` are applied element-wise. Other operators which act with respect to the matrix calculus instead of element-wise are the exponentiation operator: `^`, the left-inverse operator: `\`, and the right-inverse operator: `/`. To do these operators element-wise instead, we add a dot in front of them. For example:

```
>> A = [1 2; 1 2];
>> B = [1 0; 0 1];
>> A * B

ans =

     1     2
     1     2

>> A .* B

ans =

     1     0
     0     2
```

We see that `*` does matrix multiplication, whereas `.*` does element-wise multiplication.

0.4.2 MATLAB Arrays

0.4.2.1 Arrays as function parameters

As discussed above, we have to be wary if we want to apply operations element-wise or treating them as vectors/matrices. And so, we also have to be wary about this when writing functions. We need to think about whether arrays

passed in as a parameter should be treated as a objects of linear algebra, or as a list of numbers.

Take this first example:

```
function ret = doSomething(b)
    if size(b) ~= [3 1]
        error('argument is the wrong size.')
    end
    A = [0 -1 0;
         1 0 0;
         0 0 1];
    ret = A * b;
end
```

To not get an error, `b` would have to be a 3×1 array. And what this would do, is treat `b` as a vector in 3-space, and rotate it about the z-axis $90°$.

Compare it to this next example:

```
function ret = doSomething(b)
    if size(b) ~= [3 1]
        error('argument is the wrong size.')
    end
    A = [0 -1 0;
         1 0 0;
         0 0 1];
    ret = A .* b;
end
```

Here, `b` also has to be a 3×1 array. And what this does is it creates a matrix `A` where $\vec{x} \mapsto A\vec{x}$ for $\vec{x}$ in 3-space is rotated about the z-axis $90°$, then stretched as described by the `b`-vector. Note that these two functions are just different by one character: "`.`"

We can always force a function to treat parameters element-wise using `arrayfun`. A more detailed documentation of `arrayfun` can be found here [4].

0.4.2.2 Cell arrays

Because MATLAB treats arrays as matrices, there are special rules about the contents of matrices as well. For example, one cannot have characters and numbers in the same array. Further, a matrix cannot hold arrays of different sizes. However, there are times when we might want to hold different datatypes in the same structure. A way to do this is to use **cells**.

Cells basically encapsulate another variable, and so, MATLAB will treat them as the same datatype. This allows the storage of different datatypes in the same "list." For example, suppose `A`, `B`, and `C` store online ratings for restaurants A, B, and C respectively. Naturally, they will have a different number of people who have rated them. If we wanted to make a ratings array, this would lead to a dimension error:

```
>> rating_lists = [A;B;C]
Error using vertcat
Dimensions of arrays being concatenated are not consistent.
```

But, we can instead make a rating cell array like so:

```
>> rating_lists = {A;B;C}

rating_lists =

  3×1 cell array

    {[      2  3  5  5  1]}
    {[5  5  3  5  1  3  5  4]}
    {[  5  4  1  5  5  4  4]}
```

Note the use of curly braces. We also use curly braces to extract our original arrays. To get the `C` array, we do:

```
>> rating_lists{3}

ans =

     5     4     1     5     5     4     4
```

There is also a function called `cellfun` which applies a function over the cells of a cell array. This function is similar to `arrayfun`. Documenation of `cellfun` can be found here [5].

0.4.3 Machine Precision

MATLAB, because it is meant for the implementation of numerical methods, is a bit more "conscious" about the precision in which its calculations are done. In the courses that you will be taking, you might need to also be wary about the precision. In essence, problems that arise concerning precision comes from

how, because of how numbers in computers are stored, we cannot represent all numbers with exact accuracy [12].

For example, even in our decimal system, we cannot represent $\frac{1}{3}$ in a finite amount of space. We have to truncate at some point, e.g, $0.3333\ldots$. And as such, we cannot get exact calculations in our decimal system. Note that $3 \times \frac{1}{3} = 1$. But, $3 \times 0.3333 = 0.9999$, which is not exactly 1. Such errors also happen in MATLAB.

For example, here, as in linear algebra we diagonalize a matrix B, and we get P and D so that $B = PDP^{-1}$.

```
>> B

B =

    11    16    12
    16    38    24
    12    24    24

>> [P,D] = eig(B)

P =

    0.9347   -0.0000    0.3554
   -0.2843    0.6000    0.7478
   -0.2132   -0.8000    0.5608


D =

    3.3960         0         0
         0    6.0000         0
         0         0   63.6040
```

If we compare PDP^{-1} and B, they seem to be equal:

```
>> P*D*inv(P)

ans =

   11.0000   16.0000   12.0000
   16.0000   38.0000   24.0000
   12.0000   24.0000   24.0000
```

However, if MATLAB does the comparison, we see that it does not think that PDP^{-1} and B are exactly the same.

```
>> P*D*inv(P)

ans =

   11.0000   16.0000   12.0000
   16.0000   38.0000   24.0000
   12.0000   24.0000   24.0000
```

This inexact evaluation comes from small imprecisions in the way that P and D are calculated from B. We'll discuss two things to be wary about that can cause imprecision to be larger than what you might expect.

0.4.3.1 Error propagation

Error propagation occurs when initial errors become larger through calculations. One particular culprit, is multiplication. Suppose that we are trying to represent the value of x in MATLAB. Because of how computers store numbers, it stores an approximation instead: $\tilde{x}$. We can write $\tilde{x}$ as $\tilde{x} = x + \varepsilon$, where $|\varepsilon|$ is the **absolute error** of this approximation. The **relative error** of the approximation is the absolute error divided by the true value, in other words: $|\varepsilon|/x$.

Suppose, we want to find the square of x. MATLAB would then approximate $x^2 \approx \tilde{x}^2$. Note then that $\tilde{x}^2 = (x+\varepsilon)^2 = x^2 + 2\varepsilon x + \varepsilon^2$. And so, the absolute error of the approximation $x^2 \approx \tilde{x}^2$ is $2\varepsilon x + \varepsilon^2$. And the relative error, assuming ϵ is small, is:

$$\frac{2\varepsilon x + \varepsilon^2}{x^2} \approx 2\frac{\varepsilon}{x}$$

So we see that the relative error roughly doubled by squaring. And if we continue to square, we will once again double the error. Intuitively, errors enlarge geometrically with respect to the number of multiplications and divisions.

0.4.3.2 Catastrophic Cancellation

Another source of imprecision in calculations is **catastrophic cancellation**. Catastrophic cancellation occurs when subtracting good approximations of two close numbers resulting in a large relative error. I leave a more formal analysis like we did for error propagation to an interested reader. But to get a sense of catastrophic cancellation, let us explore an example.

Suppose Alice and Bob are arguing who is taller, they are very close in height. Alice measures 65.24 inches in height, and Bob measures 65.36 in height. Because they do not have a very accurate measuring device, Bob measures himself to be 65.4 inches tall, and Alice measures herself to be 65.2 inches tall. Note then that Alice's measurement are off by a relative error of at most 0.062%. But, from Alice's measurement, Alice concludes that there is a 0.2 inch difference in their heights. The true difference is 0.12 inches. This gives us a relative error of $(0.2 - 0.12)/0.12 = 66.\bar{6}\%$. The relative error of the approximate difference has increased by more than a thousand fold.

0.4.3.3 Condition number

For matrices, a good way to measure its propensity to produce these imprecisions is its **condition number** [1]. This can be calculated using the MATLAB function `cond`. Roughly, the number of digits the condition number of matrix has, is the number of digits you might expect to lose in precision when doing operations on the condition number. MATLAB even gives warnings on operations that are especially susceptible to the condition number, for example, taking the inverse of a matrix:

The suggestion to use the backslash or forward-slash commands instead of the inverse command is a general good practice in MATLAB. As much as possible, we should not be attempting compute the inverse of a matrix.

0.4.4 Debugging Tools in MATLAB

A great way to deal with these precision errors is to decide precisely at what precision calculations are done using **variable precision arithmetic**. Within the symbolic toolbox, there is the `vpa`-function. The `vpa`-function performs calculations with a specified precision. For example, the code `vpa(<expr>, 64)` calculates the expression `<expr>` with 64 digits of precision.

Because `vpa` is part of the symbolic toolbox, we can use symbolic expressions like `pi` as builtin representations of mathematical constants. If we are to use `vpa` without specifying digits of precision, it will default to 32 digits. To change this default by specifying `digits(d)`. The default precision of `vpa` to `d` digits. More information can be found here [11].

Another tool that can be used is MATLAB's debugger. MATLAB's debugger

allows you to set "breakpoints" where the program will pause when it reaches that line. In conjunction with the workspace, this is powerful tool that can be used to find errors. Going through a program line by line is quite inefficient, typically the use of the debugger is reserved after the use of techniques presented in an earlier section. A guide of how to use the MATLAB's debugger can be found on this website [7].

MATLAB's debugger is the ultimate debugging tool. I use the word "ultimate" in the sense of "as powerful as one can expect", and also in the sense of "should be used last." Again, MATLAB's debugger lets you go through code line by line, insodoing, you will be thinking about how the code should act line by line. If we are to jump straight into using MATLAB's debugger, we might waste a lot of time trying to pinpoint where a bug occurs. A good practice is to first use the techniques discussed earlier (displaying output mid-run, setting checkpoints, etc.) to first narrow down which part of the code we should be debugging. Further, MATLAB's debugger will not help us identify for which input our code may be buggy. For that, we still need to do code testing.

References

[1] Ascher, U. M., and Greif, C. 5.8 Estimating Errors and the Condition Number, 2011.

[2] Bach, R. MATLAB onramp. https://matlabacademy.mathworks.com/details/matlab-onramp/gettingstarted.

[3] Byrne, E. MATLAB fundamentals. https://matlabacademy.mathworks.com/details/matlab-fundamentals/mlbe.

[4] MathWorks. arrayfun. https://www.mathworks.com/help/matlab/ref/arrayfun.html.

[5] MathWorks. cellfun. https://www.mathworks.com/help/matlab/ref/arrayfun.html.

[6] MathWorks. Check code for errors and warnings using the code analyzer. https://www.mathworks.com/help/matlab/matlab_prog/check-code-for-errors-and-warnings.html.

[7] MathWorks. Debug matlab code files. https://www.mathworks.com/help/matlab/matlab_prog/debugging-process-and-features.html.

[8] MathWorks. doc. https://www.mathworks.com/help/matlab/ref/doc.html.

[9] MathWorks. Find functions to use. https://www.mathworks.com/help/matlab/matlab_env/find-functions-using-the-function-browser.html.

[10] MathWorks. mean. https://www.mathworks.com/help/matlab/ref/mean.html.

[11] MathWorks. Variable-precision arithmetic. https://www.mathworks.com/help/symbolic/vpa.html.

[12] Muller, Jean-Michel, e. a. *Handbook of Floating-Point Arithmetic*. 2010.