Linear Algebra with SageMath

Learn math with open-source software

Linear Algebra with SageMath

Learn math with open-source software

Allaoua Boughrira, Hellen Colman, Michael Kattner, Samuel Lubliner
City Colleges of Chicago

October 28, 2025

[1]github.com/SageMathOER-CCC/linear-algebra-with-sage
[2]creativecommons.org/licenses/by-sa/4.0

Preface

PLACEHOLDER

Hellen Colman
Chicago, January 2025

Acknowledgements

We would like to acknowledge the following peer-reviewers:

- [Name], [Affiliation]

We would like to acknowledge the following proof-readers:

- [Name], [Affiliation]

From the Student Authors

PLACEHOLDER

Allaoua Boughrira and Samuel Lubliner

Authors and Contributors

Allaoua Boughrira
Mathematics
Wright College
a.boughrira@gmail.com

Hellen Colman
Math Department
Wright College
hcolman@ccc.edu

Michael Kattner
Mathematics
Wright College
MDKattner@gmail.com

Samuel Lubliner
Computer Science
Wright College
sage.oer@gmail.com

Contents

Preface v

Acknowledgements vii

From the Student Authors ix

Authors and Contributors xi

1 Getting Started 1

1.1 Intro to Sage 1
1.2 Data Types 3
1.3 Flow Control Structures 5
1.4 Defining Functions 8
1.5 Object-Oriented Programming 10
1.6 Display Values 11
1.7 Debugging 12
1.8 Documentation. 14
1.9 Miscellaneous Features 15
1.10 Run Sage in the browser 16

2 Vectors and Matrices, The Basics 17

2.1 Vectors 17
2.2 Matrices 19

3 System of Equations 23

3.1 Solving Equations. 23
3.2 Solving Systems of Equations 24
3.3 Graphing of Systems 26

4 System of Equations with Matrices 29

4.1 Gauss-Jordan Elimination. 29
4.2 Augmented Matrix 30
4.3 RREF 31
4.4 Row Reduction Results. 32

4.5 Differences in Solutions . . . 33
4.6 Pivots . . . 34

5 Vectors 35

5.1 Basic Arithmetic Operations . . . 35
5.2 Dot and Cross Products . . . 35

6 Matrices 37

6.1 Special Matrices . . . 37
6.2 Operations With Matrices . . . 38
6.3 Transpose, Conjugate and Trace . . . 39

7 Determinants 41

7.1 Determinant and Inverse . . . 41

8 Inverses 43

8.1 Inverses of Matrices . . . 43

9 Adjoint Matrix 47

9.1 Adjoint Matrix . . . 47
9.2 Alternative Computation of the Adjoint, Cofactors, and Minors . 48
9.3 Historical Note: Adjoint vs. Adjugate . . . 49

10 LU Decomposition 51

10.1 LU Decomposition . . . 51

11 Equations of Lines and Planes 53

12 Orthogonality 55

12.1 Normalized Vector . . . 55

13 Linear Transformation From R Pow N to R Pow M 57

14 Eigenvectors and Eigenvalues 59

14.1 Eigenvectors . . . 59

15 Diagonalization 61

16 General Vector Spaces 63

17 Linear Independence 65

18 General Linear Transformations 67

Back Matter

References 69

Index 73

Chapter 1

Getting Started

Welcome to our introduction to SageMath (also referred to as Sage). This chapter is designed for learners of all backgrounds —whether you are new to programming or aiming to expand your mathematical toolkit. There are various ways to run Sage, including SageMathCell, CoCalc, and a local installation. The simplest way to start is by using the SageMathCell embedded in this book. We will also cover how to use CoCalc, a cloud-based platform for running Sage in a collaborative environment.

Sage is a free, open-source mathematics software system that integrates various open-source math packages[1]. This chapter introduces Sage's syntax, data types, variables, and debugging techniques. Our goal is to equip you with the foundational knowledge to explore mathematical problems and programming concepts in an intuitive and practical manner.

Join us as we explore the capabilities of Sage!

1.1 Intro to Sage

You can execute and modify Sage code directly within the SageMathCells embedded on this webpage. Cells on the same page share a common memory space. To ensure accurate results, run the cells in the sequence in which they appear. Running them out of order may cause unexpected outcomes due to dependencies between the cells.

```
# This is an empty cell that you can use to type code
# and run Sage commands. These lines here are an example
# of comments for the reader and are ignored by Sage.
```

Note that these Sage cells allow the user to experiment freely with any of the Sage- supported commands. The content of the cells can be altered at runtime (on the live web version of the book) and executed in real-time on a remote Sage server. Users can modify the content of cells and execute any other Sage commands to explore various mathematical concepts interactively.

1.1.1 Sage as a Calculator

Before we get started with linear algebra, let's explore how Sage can be used as a calculator. Here are the basic arithmetic operators:

[1] doc.sagemath.org/html/en/reference/spkg/

- `+` Addition
- `-` Subtraction
- `*` Multiplication
- `**` or `^` Exponentiation
- `/` Division
- `//` Integer division
- `%` Modulo - remainder of division

There are two ways to run the code within the cells:

- Click the Evaluate (Sage) button under the cell.
- Use the keyboard shortcut Shift + Enter while the cursor is active in the cell.

Try the following examples:

```
# Lines that start with a pound sign are comments
# and ignored by Sage
1 + 1
```

```
100 - 1
```

```
3 * 4
```

Sage supports two exponentiation operators:

```
# Sage uses two exponentiation operators
# ** is valid in Sage and Python
2**3
```

```
# Sage uses two exponentiation operators
# ^ is valid in Sage
2 ^ 3
```

Division in Sage:

```
5 / 3  # Returns a rational number
```

```
5 / 3.0  # Returns a floating-point approximation
```

Integer division and modulo:

```
5 // 3  # Returns the quotient
```

```
5 % 3   # Returns the remainder
```

1.1.2 Variables and Names

Variables store values in the computer's memory. This includes value of an expression to a variable. Use the assignment operator `=` to assign a value to a variable. The variable name is on the left side, and the value is on the right. Unlike the expressions above, an assignment does not display anything. To view a variable's value, type its name and run the cell.

```
a = 1
b = 2
sum = a + b
sum
```

When choosing variable names, follow these rules for valid identifiers:

- Identifiers cannot start with a digit.
- Identifiers are case-sensitive.
-
    - Letters (`a-z`, `A-Z`)
    - Digits (`0-9`)
    - Underscore (`_`)
- Do not use spaces, hyphens, punctuation, or special characters while naming your identifiers.
- Do not use reserved keywords as variable names.

Python has a set of reserved keywords that cannot be used as variable names:
`False`, `None`, `True`, `and`, `as`, `assert`, `async`, `await`, `break`, `class`, `continue`, `def`, `del`, `elif`, `else`, `except`, `finally`, `for`, `from`, `global`, `if`, `import`, `in`, `is`, `lambda`, `nonlocal`, `not`, `or`, `pass`, `raise`, `return`, `try`, `while`, `with`, `yield`.

To check if a word is a reserved keyword, use the `keyword` module in Python.

```
import keyword
keyword.iskeyword('if')
```

The output is `True` because `if` is a keyword. Try checking other names.

1.2 Data Types

In computer science, **Data types** define the properties of data, and consequently the behavior of operations on these data types. Since Sage is built on Python, it naturally inherits all Python's built-in data types. Sage does also introduces new additional and custom classes and specific data types better-suited and optimized for mathematical computations.

Let's check the type of a simple integer in Sage.

```
n = 2
print(n)
type(n)
```

The `type()` function confirms that `2` is an instance of Sage's **Integer** class.

Sage supports **symbolic** computation, where it retain and preserves the actual value of a math expressions rather than evaluating them for approximated values.

```
sym = sqrt(2) / log(3)
show(sym)
type(sym)
```

String: a `str` is a sequence of characters. Strings can be enclosed in single or double quotes.

```
greeting = "Hello,␣World!"
print(greeting)
print(type(greeting))
```

Boolean: a (`bool`) data type represents one of only two possible values: `True` or `False`.

```
b = 5 in Primes()  # Check if 5 is a prime number
print(f"{b}␣is␣{type(b)}")
```

List: A mutable sequence or collection of items enclosed in square brackets `[]`. (an object is mutable if you can change its value after creating it).

```
l = [1, 3, 3, 2]
print(l)
print(type(l))
```

Lists are indexed starting at `0`. The first element is at index zero, and can be accessed as follows:

```
l[0]
```

The `len()` function returns the number of elements in a list.

```
len(l)
```

Tuple: is an immutable sequence of items enclosed in parentheses `()`. (an object is immutable if you cannot change its value after creating it).

```
t = (1, 3, 3, 2)
print(t)
type(t)
```

set (with a lowercase `s`): is a Python built-in type, which represents an unordered collection of unique items, enclosed in curly braces `{}`. The printout of the following example shows there are no duplicates.

```
s = {1, 3, 3, 2}
print(s)
type(s)
```

In Sage, `Set` (with an uppercase `S`) extends the native Python's `set` with additional mathematical functionality and operations.

```
S = Set([1, 3, 3, 2])
type(S)
```

We start by defining a `list` using the square brackets `[]`. Then, Sage `Set()` function removes any duplicates and provides the mathematical set operations. Even though Sage supports Python sets, we will use Sage `Set` for the added features. Be sure to define `Set()` with an uppercase `S`.

```
S = Set([5, 5, 1, 3, 5, 3, 2, 2, 3])
print(S)
```

A **Dictionary** is a collection of key-value pairs, enclosed in curly braces `{}`.

```
d = {
    "title": "Linear␣Algebra␣with␣SageMath",
    "institution": "City␣Colleges␣of␣Chicago",
    "topics_covered": [
        "Set␣Theory",
        "Combinations␣and␣Permutations",
        "Logic",
        "Quantifiers",
        "Relations",
        "Functions",
        "Recursion",
        "Graphs",
        "Trees",
        "Lattices",
        "Boolean␣Algebras",
        "Finite␣State␣Machines"
    ],
    "format": ["Web", "PDF"]
}
type(d)
```

Use the `pprint` module to improve the dictionary readability.

```
import pprint
pprint.pprint(d)
```

1.3 Flow Control Structures

When writing programs, we want to control the flow of execution. Flow control structures allow your code to make decisions or repeat actions based on conditions. These structures are part of Python and work the same way in Sage. There are three primary types of flow control structures:

- **Assignment** statements store values in *variables*. These let us reuse results and build more complex expressions step by step. An assignment is performed using the `=` operator as discussed earlier (see Subsection 1.1.2). Note that Sage also supports compound assignment operators like `+=`, `-=`, `*=`, `/=`, and `%=` which combine assignment with basic arithmetic operations (addition, subtraction, multiplication, division and modulus).

- **Branching** uses *conditional statements* like `if`, `elif`, and `else` to execute different blocks of code based on logical tests.

- **Loops** such as `for` and `while` let us iterate over some data structures and also repeat blocks of code multiple times. This is useful when processing sequences, performing computations, or automating repetitive tasks.

These core concepts apply to almost every programming language and are fully supported in Sage through its Python foundation.

Notes. Sage uses the same control structures as Python, so most Python syntax for logic and repetition will work seamlessly in Sage.

1.3.1 Conditional Statements

The `if` statement lets your program execute a block of code only when a condition is true. You can add `else` and `elif` clauses to cover additional conditions.

```
x = 7
if x % 2 == 0:
    print("x␣is␣even")
elif x % 3 == 0:
    print("x␣is␣divisible␣by␣3")
else:
    print("x␣is␣odd␣and␣not␣divisible␣by␣3")
```

Use indentation to define blocks of code that belong to the `if`, `elif`, or `else` clauses. Just like in Python, the indentation is significant and is used to define code blocks.

1.3.1.1 Error Handling

Similar to `if` statements, `try` and `except` blocks allow for the conditional execution of code in the event of an error in the code of the `try` block. Then at the end of execution, regardless of if an error occurs the code in the `finally` block will execute. Take for example division by zero:

```
try:
    1/0
except:
    print("Division␣by␣zero␣is␣undefined")
finally:
    print("The␣code␣is␣done")
```

In the case of valid division here is how execution occurs:

```
try:
    1/1
except:
    print("Division␣by␣zero␣is␣undefined")
finally:
    print("The␣code␣is␣done")
```

1.3.2 Iteration

Iteration is a programming technique that allows us to efficiently repeat instructions with minimal syntax. The `for` loop assigns a value from a sequence to the loop variable and executes the loop body once for each value.

Here is a basic example of a `for` loop:

```
# Print the numbers from 0 to 19
# Notice that the loop is zero-indexed and excludes 20
for i in range(20):
    print(i)
```

By default, `range(n)` starts at 0. To specify a different starting value, provide two arguments:

```
# Here, the starting value (10) is included
for i in range(10, 20):
    print(i)
```

You can also define a step value to control the increment:

```
# Prints numbers from 30 to 90, stepping by 9
for i in range(30, 90, 9):
    print(i)
```

1.3.2.1 List Comprehension

List comprehension is a concise way to create lists. Unlike Python's `range()`, Sage's list comprehension syntax includes the ending value in a range.

```
# Create a list of the cubes of the numbers from 9 to 20
# The for loop is written inside square brackets
[n**3 for n in [9..20]]
```

You can also filter elements using a condition. Below, we create a list containing only the cubes of even numbers:

```
[n**3 for n in [9..20] if n % 2 == 0]
```

1.3.3 Other Flow Control Structures

In addition to `if` statements, Sage supports other common Python control structures:

- The `while` loops repeats a block of code while a condition remains true.
- The `break` statement terminates and exit a loop early.
- The `continue` statement skips the rest of the loop body and jump to the next iteration.
- The `pass` statement serves as a placeholder for future code to be added later, or to tell Sage do nothing (useful for example when we want to catch an exception so that the program does not crash, yet choose no to do anything with the exception object).

We will see examples on how to use these statements later on in the book. Here is a quick example of a `while` loop that prints out the numbers from 0 to 4:

```
count = 0
while count < 5:
    print(count)
    count += 1
```

1.4 Defining Functions

Sage comes with many built-in functions. While Math terminology is not always standard, be sure to refer to the documentation to understand the exact functionality of these built-in functions and know how to use them. You can also define custom functions to suit your specific needs. You are welcome to use the custom functions we define in this book. However, since these custom functions are not part of the Sage source code, you will need to copy and paste the functions into your Sage environment. In this section, we will explore how to define custom functions and use them.

To define a custom function in Sage, use the `def` keyword followed by the function name and the function's arguments. The body of the function is indented, and it should contain a `return` statement that outputs a value. Note that the function definition will only be stored in memory after executing the cell. You won't see any output when defining the function, but once it is defined, you can use it in other cells. If the cell is successfully executed, you will see a green box underneath it. If the box is not green, run the cell again to define the function.

A simple example of defining a function is one that returns the n^{th} (0-indexed) row of Pascal's Triangle. Pascal's Triangle is a triangular array of numbers where each number is the sum of the two numbers directly above it.

Here's a function definition that computes a specific row of Pascal's Triangle. You need execute the cell to store the function in memory. You can only call the `pascal_row()` function once the definition has been executed. If you attempt to use the function without defining it first, you will receive a `NameError`.

```
def pascal_row(n):
    return [binomial(n, i) for i in range(n + 1)]
```

After defining the function above, let's try calling it. :

```
pascal_row(5)
```

Sage functions can sometimes produce unexpected results if given improper input. For instance, passing a string or a decimal value into the function will raise a `TypeError`:

```
pascal_row("5")
```

However, if you pass a negative integer, the function will silently return an empty list. This lack of error handling can lead to unnoticed errors or unexpected behaviors that are difficult to debug, so it is essential to incorporate input validation. Let's add a `ValueError` to handle negative input properly:

```
def pascal_row(n):
    if n < 0:
        raise ValueError("`n`␣must␣be␣a␣non-negative␣
            integer")
    return [binomial(n, i) for i in range(n + 1)]
```

With the updated function definition above, try calling the function again with a negative integer. You will now receive an informative error message rather than an empty list:

```
pascal_row(-5)
```

Functions can also include a `docstring` in the function definition to describe its purpose, inputs, outputs, and any examples of usage. The `docstring` is a string that appears as the first statement in the function body. This documentation can be accessed using the `help()` function or the `?` operator.

```
def pascal_row(n):
    """
␣␣␣␣Return␣row␣`n`␣of␣Pascal's␣triangle.

␣␣␣␣INPUT:
␣␣␣␣-␣``n``␣--␣non-negative␣integer;␣the␣row␣number␣of␣
    Pascal's␣triangle␣to␣return.
␣␣␣␣␣␣The␣row␣index␣starts␣from␣0,␣which␣corresponds␣to␣the␣
    top␣row.

␣␣␣␣OUTPUT:␣list;␣row␣`n`␣of␣Pascal's␣triangle␣as␣a␣list␣of␣
    integers.

␣␣␣␣EXAMPLES:
␣␣␣␣This␣example␣illustrates␣how␣to␣get␣various␣rows␣of␣
    Pascal's␣triangle␣(0-indexed)␣:

␣␣␣␣␣␣␣␣sage:␣pascal_row(0)␣␣#␣the␣top␣row
␣␣␣␣␣␣␣␣[1]
␣␣␣␣␣␣␣␣sage:␣pascal_row(4)
␣␣␣␣␣␣␣␣[1,␣4,␣6,␣4,␣1]

␣␣␣␣It␣is␣an␣error␣to␣provide␣a␣negative␣value␣for␣`n`:
␣␣␣␣␣␣␣␣sage:␣pascal_row(-1)
␣␣␣␣␣␣␣␣Traceback␣(most␣recent␣call␣last):
␣␣␣␣␣␣␣␣...
␣␣␣␣␣␣␣␣ValueError:␣`n`␣must␣be␣a␣non-negative␣integer

␣␣␣␣..␣NOTE::
␣␣␣␣␣␣␣␣This␣function␣uses␣the␣`binomial`␣function␣to␣
    compute␣each
␣␣␣␣␣␣␣␣element␣of␣the␣row.
␣␣␣␣"""
    if n < 0:
        raise ValueError("`n`␣must␣be␣a␣non-negative␣
            integer")

    return [binomial(n, i) for i in range(n + 1)]
```

After redefining the function, you can view the `docstring` by calling the `help()` function on the function name:

```
help(pascal_row)
```

Alternatively, you can access the source code using the `??` operator:

```
pascal_row??
```

To learn more on code style conventions and writing documentation strings, refer to the General Conventions article in the Sage Developer's Guide.

1.5 Object-Oriented Programming

Object-Oriented Programming (OOP) is a programming paradigm that models the world as a collection of interacting **objects**. An object is an **instance** of a **class**, which can represent almost anything.

Classes act as blueprints that define the structure and behavior of objects. A class specifies the **attributes** and **methods** of an object. - An **attribute** is a variable that stores information about the object. - A **method** is a function that interacts with or modifies the object. Although you can create custom classes, many useful classes are already available in Sage and Python, such as those for integers, lists, strings, and graphs.

1.5.1 Objects in Sage

In Python and Sage, almost everything is an object. When assigning a value to a variable, the variable references an object. The `type()` function allows us to check an object's class.

```
vowels = ['a', 'e', 'i', 'o', 'u']
type(vowels)
```

```
type('a')
```

The output confirms that `'a'` is an instance of the `str` (string) class, and `vowels` is an instance of the `list` class. We just created a `list` object named `vowels` by assigning a series of characters within the square brackets to a variable. The object `vowels` represents a `list` of `string` elements, and we can interact with it using various methods.

1.5.2 Dot Notation and Methods

Dot notation is used to access an object's attributes and methods. For example, the `list` class has an `append()` method that allows us to add elements to a list.

```
vowels.append('y')
vowels
```

Here, `'y'` is passed as a **parameter** to the `append()` method, adding it to the end of the list. The `list` class provides many other methods for interacting with lists.

1.5.3 Sage's Set Class

While `list` is a built-in Python class, Sage provides specialized classes for mathematical objects. One such class is `Set`, which we will explore later on in the next chapter.

```
v = Set(vowels)
type(v)
```

The `Set` class in Sage provides attributes and methods specifically designed for working with sets. While OOP might seem abstract at first, it will become clearer as we explore more and dive deeper into Sage features. Sage's built-in classes offer a structured way to represent data and perform powerful mathematical operations. In the next chapters, we will see how Sage utilizes OOP principles and its built-in classes to perform mathematical operations.

1.6 Display Values

Sage provides multiple ways to display values on the screen. The simplest way is to type the value into a cell, and Sage will display it. Sage also offers functions to format and display output in different styles.

Sage automatically displays the value of the last line in a cell unless a specific function is used for output. Here are some key functions for displaying values:

- `print()` displays the value of the expression inside the parentheses as plain text.
- `pretty_print()` displays rich text as typeset LaTeX output.
- `show()` is an alias for `pretty_print()` and provides additional functionality for graphics.
- `latex()` returns the raw LaTeX code for the given expression, which then can be used in LaTeX documents.
- `%display latex` enables automatic rendering of all output in LaTeX format.
- While Python string formatting is available and can be used, it may not reliably render rich text or LaTeX expressions due to compatibility issues.

Let's explore these display methods in action.

Typing a string directly into a Sage cell displays it with quotes.

```
"Hello, World!"
```

Using `print()` removes the quotes.

```
print("Hello, World!")
```

The `show()` function formats mathematical expressions for better readability.

```
show(sqrt(2) / log(3))
```

To display multiple values in a single cell, use `show()` for each one.

```
a = x^2
b = pi
show(a)
show(b)
```

The `latex()` function returns the raw LaTeX code for an expression.

```
latex(sqrt(2) / log(3))
```

In Jupyter notebooks or SageMathCell, you can set the display mode to LaTeX using `%display latex`.

```
%display latex
# Notice we don't need the show() function
sqrt(2) / log(3)
```

Once set, all expressions onward will continue to be rendered in LaTeX format until the display mode is changed.

```
ZZ
```

To return to the default output format, use `%display plain`.

```
%display plain
sqrt(2) / log(3)
```

```
ZZ
```

1.7 Debugging

Error messages are an inevitable part of programming. When you encounter one, read it carefully for clues about the cause. Some messages are clear and descriptive, while others may seem cryptic. With practice, you will develop valuable skills debugging your code and resolving errors

Note that not all errors result in error messages. **Logical errors** occur when the syntax is correct, but the program does not produce the expected result. Usually, these are a bit harder to trace.

Remember, mistakes are learning opportunities —everyone makes them! Here are some useful debugging tips:

- Read the error message carefully —it often provides useful hints.
- Consult the documentation to understand the correct syntax and usage.
- Google-search the error message —it's likely that others have encountered the same issue.
- Check SageMath forums for previous discussions.
- Take a break and return with a fresh perspective.
- Ask the Sage community if you are still stuck after trying all the above steps.

Let's dive in and make some mistakes together!

A **SyntaxError** usually occurs when the code is not written according to the language rules.

```
# Run this cell and see what happens
1message = "Hello,␣World!"
print(1message)
```

Why didn't this print `Hello, World!` to the console? The error message indicates a `SyntaxError: invalid decimal literal`. The issue here is the invalid variable name. Valid *identifiers* must:

- Start with a letter or an underscore (never with a number).
- Avoid any special characters other than the underscores.

Let's correct the variable name:

```
message = "Hello,␣World!"
print(message)
```

A **NameError** occurs when a variable or function is referenced before being defined.

```
print(Hi)
```

Sage assumes `Hi` is a variable, but we have not defined it yet. There are two ways to fix this:

- Use quotes to indicate that `Hi` is a string.

```
print("Hi")
```

- Alternatively, if we intended `Hi` to be a variable, then we must define it before first use.

```
Hi = "Hello,␣World!"
print(Hi)
```

Reading the documentation is essential to understanding the proper use of methods. If we incorrectly use a method, we will likely get a `NameError` (as seen above), an `AttributeError`, a `TypeError`, or `ValueError`, depending on the mistake.

Here is another example of a `NameError`:

```
l = [6, 1, 5, 2, 4, 3]
sort(l)
```

The `sort()` method must be called on the list object using dot notation.

```
l = [4, 1, 2, 3]
l.sort()
print(l)
```

An **AttributeError** occurs when an invalid method is called on an object.

```
l = [1, 2, 3]
l.len()
```

The `len()` function must be used separately rather than as a method of a list.

```
len(l)
```

A **TypeError** occurs when an operation or function is applied to an *incorrect* data type.

```
l = [1, 2, 3]
l.append(4, 5)
```

The `append()` method only takes one argument. To add multiple elements, use `extend()`.

```
l.extend([4, 5])
print(l)
```

A **ValueError** occurs when an operation receives an argument of the correct type but with an invalid value.

```
factorial(-5)
```

Although the resulting error message is lengthy, the last line informs us that Factorials are only defined for non-negative integers.

```
factorial(5)
```

A **Logical error** does not produce an error message but leads to incorrect results.

Here, assuming your task is to print the numbers from 1 to 10, and you mistakenly write the following code:

```
for i in range(10):
    print(i)
```

This instead will print the numbers 0 to 9 (because the start is inclusive but not the stop). If we want numbers 1 to 10, we need to adjust the range.

```
for i in range(1, 11):
    print(i)
```

To learn more, check out the CoCalc article[1] about the top mathematical syntax errors in Sage.

1.8 Documentation

Sage offers a wide range of features. To explore what Sage can do, check out the Quick Reference Card[1] and the Reference Manual[2] for detailed information.

The tutorial[3] offers a useful overview for getting familiar with Sage and its functionalities.

You can find Sage documentation[4] at the official website. At this stage, reading the documentation is optional, but we will guide you through getting started with Sage in this book.

To quickly reference Sage documentation, use the ? operator in Sage. This can be a useful way to get immediate help with functions or commands. You can also view the source code of functions using the ?? operator.

```
Set?
```

```
Set??
```

```
factor?
```

```
factor??
```

[1]github.com/sagemathinc/cocalc/wiki/MathematicalSyntaxErrors
[1]wiki.sagemath.org/quickref
[2]doc.sagemath.org/html/en/reference/
[3]doc.sagemath.org/html/en/tutorial/
[4]doc.sagemath.org/html/en/index.html

1.9 Miscellaneous Features

Sage is feature rich, and the following is a brief introduction to some of its miscellaneous features. Keep in mind that the primary goal of this book is to introduce Sage software and demonstrate how it can be used to experiment with linear algebra concepts within Sage environment.

Sage is used here interactively, and mainly covering the basics. Having an understanding of any of the commands presented in this section would be crucial for working on a production-grade project with complex mathematical models (e.g. handling large datasets). In such cases, it would be more appropriate to use these commands within a standalone Sage environment. These commands are presented here just for the sake of completeness.

1.9.1 Reading and Writing Files in Sage

Sage provides various ways to handle input and output (I/O) operations.

This subsection explores writing data to files and importing data from files.

Sage allows reading from and writing to files using standard Python file-handling functions.

Writing to a file:

```
with open("output.txt", "w") as file:
    file.write("Hello,␣Sage!")
```

Reading from a file:

```
with open("output.txt", "r") as file:
    content = file.read()
    print(content)  # Output: Hello, Sage!
```

1.9.2 Executing Shell Commands in Sage

Sage allows executing shell commands directly using the `!` operator (prefix the shell command to be executed).

Listing the content of the current directory showing the file that we just created:

```
!ls -la
```

1.9.3 Importing and Exporting Data (CSV, JSON, TXT)

Sage supports structured data formats such as CSV and JSON.

Generating a CSV file using shell command:

```
!printf
    "Name,Age,Country\nAlice,25,USA\nBob,30,UK\nCharlie,28,Canada\n"
    > data.csv
```

Reading a CSV file in Sage:

```
import csv

with open("data.csv", "r") as file:
    reader = csv.reader(file)
    for row in reader:
        print(row)
```

1.9.4 Using External Libraries in Sage

Sage allows using external Python libraries to do advance calculation or access and communicate over a network (urllib.request library) .

Using NumPy for numerical computations:

```
import numpy as np
array = np.array([1, 2, 3])
print(array)  # Output: [1 2 3]
```

1.10 Run Sage in the browser

The easiest way to get started is by running Sage online. However, if you do not have reliable internet access, you can also install the software locally on your own computer. Begin your journey with Sage by following these steps:

1. Navigate to Sage website[1].
2. Click on Sage on CoCalc[2].
3. Create a CoCalc account[3].
4. Go to Your Projects[4] on CoCalc and create a new project.
5. Start your new project and create a new worksheet. Choose the SageMath Worksheet option.
6. Enter Sage code into the worksheet. Try to evaluate a simple expression and use the worksheet like a calculator. Execute the code by clicking `Run` or using the shortcut `Shift + Enter`. We will learn more ways to run code in the next section.
7. Save your worksheet as a PDF for your records.
8. To learn more about Sage worksheets, refer to the documentation[5].
9. Alternatively, you can run Sage code in a Jupyter Notebook[6] for additional features.
10. If you are feeling adventurous, you can install Sage[7] and run it locally on your own computer. Keep in mind that a local install will be the most involved way to run Sage code. When using Sage locally, commands to display graphics will create and then open a temporary file, which can be saved permanently through the software used to view it.

[1]https://www.sagemath.org/
[2]https://cocalc.com/features/sage
[3]https://cocalc.com/auth/sign-up
[4]https://cocalc.com/projects
[5]https://doc.cocalc.com/sagews.html
[6]doc.cocalc.com/jupyter-start.html
[7]doc.sagemath.org/html/en/installation/index.html

Chapter 2

Vectors and Matrices, The Basics

Vectors and matrices are fundamental concepts in linear algebra. This chapter introduces how to define and construct them in Sage.

2.1 Vectors

In this section, we will see how to define vectors, and perform basic operations on them.

Sage provides built-in support for vectors. In Sage, vectors are represented as n-tuples, $(v_1, v_2, ..., v_n) \in R^n$ where n is the number of *components* in the vector. Vectors can be defined using `vector` command, and passing the values of the vectors components.

```
v=vector([1, 2, 3])
v
```

The number of components n of a vector $v = (v_1, \ldots, v_n)$ is obtained in Sage by using the command `degree`.

```
v.degree()
```

To retrieve the components of a vector as a list, the method `list` can be used

```
v.list()
```

Note that the return type of that command is the Python built-in `List` type; an ordered list of numbers where the order matters. As such, any and all native list methods can be used on the returned value.

Recall that lists in Sage are 0-indexed, meaning that the first element of the list is at index 0. To access a specific component of a vector, we can use vector indexing method. Here how to access the first component of the vector v.

```
v[0]
```

The magnitude of a vector, $||v|| = \sqrt{v_1^2 + v_2^2 + ... + v_n^2}$ is obtained in Sage by using the vector method `norm`.

```
v.norm()
```

A vector in R^{n+1} can be constructed from a vector in R^n by appending the values for the additional components.

```
vector(v.list() + [4])
```

Vectors in Sage can be created in different *fields* based on the datatype of the components of the vector. While working in a specific field, we need to explicitly pass the field when instantiating a vector using the `vector` command.

```
# creating 3D vectors in Integers field
u = vector(ZZ, v)
u
```

Note that ZZ is Sage notation for the Integers field. Similarly, QQ is for Rational numbers, RR for Reals, and CC for Complex numbers. The method `base_ring` returns the *base ring* of the vector.

```
u.base_ring()
```

If the field is omitted, Sage will infer the field from the datatype of the vector components.

```
w = vector([1/2, 2/3, 3/5])
w.base_ring()
```

Sage also supports vectors over the complex field CC. While such vectors are not commonly encountered in elementary linear algebra, they play an essential role in many engineering applications. For instance, in signal processing, orthogonal signals are frequently expressed as complex vectors, enabling the use of a single transformation matrix to act on the entire set, rather than applying the transformation to each signal individually.

To create a vector in the complex field, we can either explicitly specify the field as CC, or implicitly by using complex numbers as components of the vector like below.

```
z=vector([1.2 + i * 2.3, 3.5 - i * 5.7])
z.base_ring()
```

In Sage, the complex conjugate of a vector is found by calling the `conjugate()` method on the vector itself.

```
z.conjugate()
```

In low dimensions ($n \leq 3$), the geometrical representation of a vector can be visualized as an arrow starting from the origin to the point $P = (v_1, v_2, ..., v_n)$. Although Sage accepts vectors of any dimension, the visual representation is only possible and meaningful in 2D and 3D, and high dimensional vectors are to be taken as abstract objects.

To display a vector in Sage, we can use the internal method `plot` of *vector* class.

```
u.plot(color='red', thickness=2)
```

Note that we can also obtain the same visual representation using Sage default `plot` method.

```
plot(u, color='red', thickness=2)
```

2.2 Matrices

In this section, we will see how to define matrices, and perform basic operations on them. A matrix is an $m \times n$ rectangular array of numbers:

$$\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix}$$

where m is the number of rows and n is the number of columns. Each *entry* $a_{ij}, i = 1 \dots m, j = 1 \dots n$, corresponds to the value at the intersection of the i-th row and j-th column.

Just like vectors, Sage does come with built-in support for matrices. There are many ways to define a matrix in Sage using the command `matrix`, passing for instance each of the rows of the matrix as a *list of numbers*.

```
M = matrix([
    [11, 13, 17, 19],
    [23, 29, 31, 37],
    [41, 43, 47, 53],
])

M
```

Or as a *list of vectors*.

```
v1=vector([11, 13, 17, 19])
v2=vector([23, 29, 31, 37])
v3=vector([41, 43, 47, 53])

M = matrix([v1, v2, v3])
M
```

Alternatively, we can create a matrix by passing the entries as a list, and the dimensions as arguments to the `matrix` command. Here is an example of a 2×3 matrix.

```
M = matrix(2, 3, [1, 1, 2, 3, 5, 8])
M
```

Note that in Sage, you can create a list consisting of repeated copies of the same element using the repetition operator `*`. For example, `[1] * 6` produces a list with six ones, which is convenient when filling a matrix with identical entries like in the following example.

```
matrix(2, 3, [1] * 6)
```

We do not need to specify both dimensions of a matrix. Sage can infer the missing dimension from the number of entries in the list. Here is an example of a 3×4 matrix containing the first 100 integers, and where only the number of rows is specified.

```
matrix(5, list(range(100)))
```

The entries of the matrix can also be defined programmatically using a list comprehension.

```
M = matrix(2, 5, [j + i * 5 for i in range(2) for j in
    range(5)])
M
```

Sage also allows to construct a larger matrix by combining smaller *submatrices* using the `block_matrix()` function (or `matrix.block()`). Submatrices can be arranged in rows and columns to form a single, larger matrix. Each row is entered as a list. In the following example, we create a 5×5 matrix from four submatrices of different dimensions.

```
A = matrix(2, 3, [1]*6)  # A 2x3 matrix filled with 1's
B = matrix(2, 2, [2]*4)  # A 2x2 matrix filled with 2's
C = matrix(3, 3, [3]*9)  # A 3x3 matrix filled with 3's
D = matrix(3, 2, [4]*6)  # A 3x2 matrix filled with 4's

M = block_matrix([
    [A, B],
    [C, D]
])
M
```

Matrices of different dimensions can be used with the command `block_matrix` as long as the dimensions of the submatrices allow for their concatenation.

The individual entries of a matrix can be accessed using the row and column indices. Since indices in Sage start at 0, to retrieve the entry a_{ij} in a matrix M, we type `M[i-1, j-1]`. Here is, for instance, how we can retrieve the entry at the second row and third column of the matrix M.

```
a_23 = M[1, 2]
a_23
```

Sage provides dedicated methods to retrieve all rows of the matrix, or all its columns. It also support the retrieval of the diagonal elements of a matrix, or extracting a specific row or a specific column of a given matrix.

The `rows` method returns all rows of the matrix (as a list of tuples).

```
M.rows()
```

In the same way, the `columns` method returns all the columns of the matrix (as a list of tuples).

```
M.columns()
```

The `M.row(i-1)` returns the i-th row of the matrix.

```
M.row(1)     # 2nd row
```

The `M.column(j-1)` returns the j-th column of the matrix.

```
M.column(3)  # 4th column
```

The `M.diagonal()` returns the main diagonal of the matrix.

```
M.diagonal()
```

Similarly, Sage offers more methods to help us iterate over the rows or the columns of a matrix. For instance, the method `nrows()` returns the number of rows of a matrix.

```
M.nrows()
```

Sage method `ncols()` returns the number of rows of the matrix.

```
M.ncols()
```

Alternatively, the method `dimensions` returns the dimension of the matrix as a pair `(nrows, ncols)`.

```
M.dimensions()
```

To extract a *submatrix* from a matrix given the range of rows and columns to include, Sage command `matrix_from_rows_and_columns()` takes two lists: the first for row indices, and the second for column indices as in the following example.

```
A = matrix([[1, 2, 3],
            [4, 5, 6],
            [7, 8, 9]])

A.matrix_from_rows_and_columns([0, 1], [1, 2])
```

The resulting submatrix from the previous command contains the entries at the intersection of the indexed rows (i.e. first and second rows) and the indexed columns (i.e. the second and third columns).

Note that the order of the indices in the lists matters and will affect the order of the rows and columns in the resulting submatrix. In the following example, the order of the columns in the resulting submatrix is different from the previous example.

```
A.matrix_from_rows_and_columns([0, 1], [2, 1])
```

An alternative, and perhaps more straightforward way to achieve the same result is to use Python-like slicing and explicitly listing the range of interest as in the following example.

```
A[0:2, 1:3]
```

The syntax `A[i:j, k:l]` extracts the submatrix from rows i to $j-1$ and columns k to $l-1$ (lower bound inclusive but not the upper bound of the range).

Note that a matrix with a single list of values creates a *vector-like* object, but it is NOT a vector. Here is a vector in Sage.

```
v=vector([1, 2, 3])
v
```

Compare that to this single-row matrix. Observe the square brackets in matrices in lieu of the parenthesis.

```
m=matrix([1, 2, 3])
m
```

Type assertion is a useful tool to check if two objects are *identical* and are of the same type. For instance, comparing the types of v and m yields `False` because they are NOT of the same type.

```
v==m
```

Just like vectors, matrices in Sage can also be created in different *Fields*, inferred from the datatype passed to the `matrix` command, or explicitly when instantiating the matrix.

```
m_int = matrix(ZZ, 3, list(range(15)))
m_int
```

The method `base_ring` returns the *base ring* of the matrix.

```
m_int.base_ring()
```

Once again, if the field is omitted, Sage will simply infers the field from the datatype of the matrix entries. Sage matrices supports the same fields as vectors (ZZ, QQ, RR, CC).

Chapter 3

System of Equations

Linear systems of equations are often given geometric meaning as intersections between lines, planes, or higher dimensional equivalents. This chapter will introduce how to use sage to find solutions of equations in their familiar form.

3.1 Solving Equations

The `solve` function algebraically solves an equation or system of equations. We will begin by focusing on solving a single equation. By default, Sage assumes the expression is equal to 0.

Let's solve for x in the equation $8 + x = 0$.

```
solve(8 + x, x)
```

Let's solve for x in the equation $8 + x = 5$.

```
solve(8 + x == 5, x)
```

We can store the equation in a new variable and perform operations on it. Recall the single equal sign (=) is the **assignment operator**, while the double equal sign (==) is the **equality operator**.

```
# Define the equation and store it in a variable
E = 8 + x == 5

# Solve the equation for x
solve(E, x)
```

Observe that the solution of an equation is given between square brackets, indicating that the data type is a `list`. We can access the solutions in the `list` with square brackets. Notice that after accessing the solution, the brackets are no longer present.

```
# Define the equation and store it in a variable
E = 8 + x == 5

# Store the solution in a variable
S = solve(E, x)
```

```
# Access the zero-th element of the list
S[0]
```

In this case the `list` has only one element. However, other equations may have multiple solutions and therefore multiple elements in the `list`.

We can also access each side of the equation. Here is the right-hand side of the equation.

```
E.rhs()
```

Here is the left-hand side of the equation.

```
E.lhs()
```

To use a variable other than `x`, we need to create a **symbolic** variable object with the `var()` function. Otherwise, Sage will not recognize the variable as a symbolic variable.

```
# Create a symbolic variable
var('z')

# Solve for z
solve(8 + z == 5, z)
```

3.2 Solving Systems of Equations

Sage allows for us to generalize the previous methods to solve systems of equations. In order to do this we must first define multiple symbolic variables, then we use the `solve` function as we did before.

There are various ways to create multiple symbolic variables at once. The following are all valid.

```
# Single string
var('x␣y')

# List of strings
var(['x','y'])

# Multiple strings
var('x', 'y')
```

Notice how these symbolic variables are all equivalent regardless of how they are created.

```
var('x␣y') == var(['x','y']) == var('x', 'y')
```

To solve a system of equations algebraically, we first define the variables and the equations in the system.

```
# Create symbolic variables
var('x␣y')

# Define the equations and store them in variables
eq1 = 2*x + 3*y == 4
eq2 = 5*x - 6*y == 7

show(eq1)
show(eq2)
```

In this case, the first argument to `solve` is a list of equations and the following arguments are the variables being solved for.

```
S = solve([eq1, eq2], x, y)
S
```

Observe that Sage returns a nested list structure. The outer list contains all possible solutions to the system (in this case, there is only one solution). Each solution is represented as an inner list of equations showing the value of each variable.

```
# Access the first (and only) solution
S[0]
```

```
# Access the x-value equation from the solution
S[0][0]
```

```
# Access the y-value equation from the solution
S[0][1]
```

We also can solve a system of three equations with three variables. This system of equations has exactly one solution and will hereby be referred to as **Case I**.

```
var('x␣y␣z')
eq1 = y - z == 0
eq2 = x + 2*y == 4
eq3 = x + z == 4
solve([eq1, eq2, eq3], x, y, z)
```

The following is an example of a system with infinitely many solutions. The general solution is expressed as a parameterized function. This system will hereby be referred to as **Case II**.

```
var('x␣y␣z')
eq1 = x + 2*y == 4
eq2 = y - z == 0
eq3 = x + 2*z == 4
solve([eq1, eq2, eq3], x, y, z)
```

The following is an example of an inconsistent system. Since the system has no solutions, Sage returns an empty list. This system will hereby be referred to as **Case III**.

```
var('x␣y␣z')
eq1 = x + 2*y == 4
eq2 = x + 2*y == 1
eq3 = x + 2*z == 4
solve([eq1, eq2, eq3], x, y, z)
```

3.3 Graphing of Systems

Since each equation defines implicitly a function, we can use `implicit_plot` to graph each equation in 2D-space.

```
var('x␣y')
eq1 = 2*x + 3*y == 4
eq2 = 5*x - 6*y == 7
p1 = implicit_plot(eq1, (x, -2, 5), (y, -4, 4),
    color='green')
p2 = implicit_plot(eq2, (x, -2, 5), (y, -4, 4), color='red')
p1+p2
```

We can also plot a point whose coordinates are the ones given in the solution.

```
S = solve([eq1, eq2], x, y)
A = S[0][0].rhs()
B = S[0][1].rhs()
P = point((A, B))
P
```

Now the full plot with the equations, the solution, and a legend.

```
PP = point((A, B), size=50, legend_label=f'Solution:␣({A},␣
    {B})')

show(p1 + p2 + PP, axes=True)
```

We can plot an equations in 3D-space using `implicit_plot3d`. Here is an example with **Case I**:

```
var('x␣y␣z')
eq1 = y - z == 0
eq2 = x + 2*y == 4
eq3 = x + z == 4
a = implicit_plot3d(eq1, (x,0,5), (y,0,5), (z,0,5),
    color='blue')
b = implicit_plot3d(eq2, (x,0,5), (y,0,5), (z,0,5),
    color='red')
c = implicit_plot3d(eq3, (x,0,5), (y,0,5), (z,0,5),
    color='green')
```

```
a + b + c
```

Here is the same method with **Case II**:

```
eq1 = x + 2*y == 4
eq2 = y - z == 0
eq3 = x + 2*z == 4
a = implicit_plot3d(eq1, (x,0,5), (y,0,5), (z,0,5),
    color='blue')
b = implicit_plot3d(eq2, (x,0,5), (y,0,5), (z,0,5),
    color='red')
c = implicit_plot3d(eq3, (x,0,5), (y,0,5), (z,0,5),
    color='green')
a + b + c
```

Here is another example with **Case III**:

```
eq1 = x + 2*y == 4
eq2 = x + 2*y == 1
eq3 = x + 2*z == 4
a = implicit_plot3d(eq1, (x,0,5), (y,0,5), (z,0,5),
    color='blue')
b = implicit_plot3d(eq2, (x,0,5), (y,0,5), (z,0,5),
    color='red')
c = implicit_plot3d(eq3, (x,0,5), (y,0,5), (z,0,5),
    color='green')
a + b + c
```

Chapter 4

System of Equations with Matrices

Systems of linear equations can be interpreted as matrices in order to make solving them more efficient as the number of variables grows. This chapter shows how to use Sage to construct and solve these matrices.

4.1 Gauss-Jordan Elimination

We can also solve a system of linear equations using matrices. First, construct the augmented matrix by extracting the coefficients of the variables and placing the constants from the right-hand side of the equations as the last column. Then, perform elementary row operations to reduce this augmented matrix to **reduced row echelon form** (RREF). Finally, convert the matrix back into a system of equations to explicitly display the solutions. A matrix is in **reduced row echelon form** if:

- The first nonzero number in the row is a leading 1.
- In any two consecutive rows that do not consist entirely of zeros, the leading 1 in the lower row occurs farther to the right than the leading 1 in the higher row.
- Rows that consist entirely of zeros are at the bottom of the matrix.
- Each column that contains a leading 1 has zeros everywhere else in that column.

The following are examples of methods that preform an elementary row operation on a matrix. Keep in mind that all of the following methods use zero-based indexing.

The `add_multiple_of_row(i, j, s)` method preforms an elementary row operation on a matrix by adding s times row j to row i.

The `swap_rows(r1, r2)` method also preforms an elementary row operation on a matrix by swapping rows `r1` and `r2`.

The `rescale_row(i,s)` method preforms an elementary row operation by multiplying row i by s in place.

In the following sections, we will use Sage to solve **Case I** with an augmented matrix. Here is **Case I** restated:

$$\begin{aligned} y - z &= 0 \\ x + 2y &= 4 \\ x + z &= 4 \end{aligned}$$

4.2 Augmented Matrix

First, we create the **augmented matrix** for the system of equations. Each row in the augmented matrix lists the coefficients of the variables in an equation. While we do not directly manipulate the variables themselves, they indicate the position of each coefficient. The last column of the augmented matrix contains the constants from the right-hand side of each equation.

$$\left[\begin{array}{ccc|c} 0 & 1 & -1 & 0 \\ 1 & 2 & 0 & 4 \\ 1 & 0 & 1 & 4 \end{array}\right]$$

Let's define the coefficient matrix, consisting of the coefficients extracted from the system of equations where each column corresponds to a variable:

```
A = matrix([
    [0, 1, -1],
    [1, 2, 0],
    [1, 0, 1]
])
A
```

Let's define the constants vector:

```
b = vector([0, 4, 4])  # Right-hand side of the equations
b
```

Next, create the augmented matrix by passing the constants vector to the `augment()` method of the coefficient matrix.

```
Ab = A.augment(b)
Ab
```

Alternatively, we can create the augmented matrix directly by passing a list of lists to the matrix function.

```
Ab = matrix([
        [0, 1, -1, 0], # y - z  = 0
        [1, 2, 0, 4],  # x + 2y = 4
        [1, 0, 1, 4]   # x + 2z = 4
    ])
Ab
```

4.3 RREF

First, we will move the leading 1 from the second row to the first row.

```
Ab.swap_rows(0,1)
Ab
```

The next step is to multiply the first row by -1 and add it to the third row.

```
Ab.add_multiple_of_row(2, 0, -1)
Ab
```

Next, multiple the second row by 2 and add it to the third row.

```
Ab.add_multiple_of_row(2, 1, 2)
Ab
```

Multiply the third row by -1 and add it to the second row.

```
Ab.add_multiple_of_row(1, 2, -1)
Ab
```

Multiply the second row by -2 and add it to the first row.

```
Ab.add_multiple_of_row(0, 1, -2)
Ab
```

Multiply the third row in place by -1.

```
Ab.rescale_row(2,-1)
Ab
```

Notice that this matrix satisfies the definition of reduced row echelon form. Now that we have the matrix in reduced echelon form, we can convert the matrix back into a system of equations:

$$\begin{aligned} x &= 4 \\ y &= 0 \\ z &= 0 \end{aligned}$$

Alternatively, Sage has a built in method to do all of this for us. The `rref()` method returns the reduced row echelon Form of a matrix.

```
Ab = matrix([ # Re-use our old augmented matrix
        [0, 1, -1, 0],
        [1, 2, 0, 4],
        [1, 0, 1, 4]
    ])
Ab.rref()
```

Here is an example of `rref()` on **Case II**, with infinitely many solutions.

```
M = matrix([
        [1, 2, 0, 4],
        [0, 1, -1, 0],
        [1, 0, 2, 4]
    ])
M.rref()
```

Here is `rref()` on **Case III**, with no solutions:

```
T = matrix([
        [1, 2, 0, 4],
        [1, 2, 0, 1],
        [1, 0, 2, 4]
    ])
T.rref()
```

4.4 Row Reduction Results

As an introduction to solving equations in Sage, we worked with equations symbolically. Now, we will learn how to solve systems of equations with vectors and matrices.

Let's solve **Case II** again:

$$\begin{gathered} y - z = 0 \\ x + 2y = 4 \\ x + z = 4 \end{gathered}$$

In Sage, we can create an coefficient matrix using the `matrix` function.

```
M = matrix([
        [1, 2, 0],
        [0, 1, -1],
        [1, 0, 2]
    ])
M
```

Next, provide a list of constants to the `vector` function.

```
u = vector([4, 0, 4])
u
```

Finally, call the `solve_right()` method on the matrix and pass the solution vector as its argument.

```
M.solve_right(u)
```

The output `(4, 0, 0)` expresses one solution to the system:

$$\begin{gathered} x = 4 \\ y = 0 \\ z = 0 \end{gathered}$$

4.5 Differences in Solutions

To find all solutions we first need to augment matrix M with vector b. This is done with the `augment()` method.

```
# subdivide=True adds a dividing line to the new matrix
Mu = M.augment(u, subdivide=True)
Mu
```

To solve this new matrix, call the `rref()` method to output the matrix in reduced row echelon form.

```
Mu.rref()
```

The row of zeros at the bottom indicates that the system has infinitely many solutions.

These methods also differ in how they handle a system with no solutions. For example we will pass **Case III** to the `solve_right()` method, and an error will be thrown.

```
T = matrix([
    [1, 2, 0],
    [1, 2, 0],
    [1, 0, 2]
])
v = vector([4, 1, 4])
try:
    T.solve_right(v)
except Exception as e:
    print(e)
```

Where as the `rref()` method will return a matrix which represents an inconsistent system.

```
T.augment(v, subdivide=True).rref()
```

The two methods differ very little when there is only one solution, like with **Case I** bellow. The `solve_right()` method is efficient when the system has a unique solution.

```
A = matrix([
    [0, 1, -1],
    [1, 2, 0],
    [1, 0, 1]
])
b = vector([0, 4, 4])
A.solve_right(b)
```

```
Ab = A.augment(b)
Ab.rref()
```

4.6 Pivots

The concept of pivots give us the ability to differentiate how many ”dimensions” a system of equations’ solutions has. Pivots refer to the following features of a matrix’s **reduced row echelon form**:

- Each leading 1 denotes a pivot
- The **pivot position** refers to the row and column a pivot inhabits
- The **pivot column** refers to the column in which a pivot inhabits
- Rows that consist entirely of zeros do not contain pivots.

Sage has a built in `pivot()` method which returns the, zero-indexed, pivot columns of each row and excludes rows without pivots as a `list`.

Here is an example with **Case I**:

```
A = matrix([ # Re-use Case I matrix
        [0, 1, -1, 0],
        [1, 2, 0, 4],
        [1, 0, 1, 4]
    ])
A.pivots()
```

Here is an example with **Case II**:

```
M = matrix([
        [1, 2, 0, 4],
        [0, 1, -1, 0],
        [1, 0, 2, 4]
    ])
M.pivots()
```

Here is another example of a system with infinitely many solutions but this time with one pivot.

```
C = matrix([
        [3, 3, 1, 1],
        [9, 9, 3, 3],
        [6, 6, 2, 2]
    ])
print(C.rref())
print(C.pivots())
```

This is what was referred to earlier by the ”dimensions”, more formally referred to as degrees of freedom, the solutions inhabit.

Chapter 5

Vectors

Vectors can be thought of as the building blocks of linear algebra. In this chapter, we will explore more vector operations in Sage.

5.1 Basic Arithmetic Operations

In this section, we will introduce how to perform the arithmetic operations on vectors in Sage.

Vectors can be added and subtracted using the + and - operators. The result of adding or subtracting two vectors is a new vector with the same number of components.

```
v = vector([1, 2, 3])
w = vector([4, 5, 6])
v + w
```

```
v - w
```

Vectors can be multiplied by real numbers (scalar multiplication) using the * operator. Here is an example of scalar multiplication of a vector by 2.

```
2 * v
```

5.2 Dot and Cross Products

The *dot product* of two vectors $v = (v_1, v_2, ..., v_n)$ and $w = (w_1, w_2, ..., w_n)$ is defined as the sum of the products of their corresponding components $v \cdot w = w \cdot v = \sum_{i=1}^{n} v_i w_i$.

```
v = vector([1, 2, 3])
w = vector([4, 5, 6])
v.dot_product(w)
```

Observe that the dot product of two vectors produces a scalar (a single number), not a vector. Also, note that the *dot* product in Sage is implemented in two ways:

```
print(v * w)
print(v.dot_product(w))
```

We can also compute the dot product manually by its definition.

```
sum([v[i]*w[i] for i in range(v.degree())])
```

The cross product of two vectors $v = (v_1, v_2, v_3)$ and $w = (w_1, w_2, w_3)$ is defined as the vector whose components are given by $v \times w = (v_2w_3 - v_3w_2, v_3w_1 - v_1w_3, v_1w_2 - v_2w_1)$. In Sage, the cross product is calculated using the `cross_product` method.

```
v = vector([1, 2, 3])
w = vector([4, 5, 6])
v.cross_product(w)
```

Observe that the cross product of two vectors is a vector.

Chapter 6

Matrices

Matrices are often interpreted as a natural expansion of vectors into higher dimensions. But, the concept of a matrix extends beyond the simple grid of numbers; in fact, matrices serve as mathematical tool for structuring data, encoding relationships between multiple quantities, representing linear transformations, or solving systems of equations.

In this chapter, we will see most common matrix operations often used in linear algebra.

6.1 Special Matrices

Some matrices occur frequently enough to be given special names:

- The *zero matrix* contains only zeros and acts as the additive identity. The command `zero_matrix(m, n)` creates an m by n zero matrix.

  ```
  # 2x5 zero matrix
  zero_matrix(2,5)
  ```

  For a square matrix, the command can be shortened to `zero_matrix(n)` like in the example below.

  ```
  # 3x3 zero matrix
  zero_matrix(3)
  ```

- A *diagonal matrix* is a square matrix that has zero entries everywhere outside the *main diagonal.* Note that a diagonal matrix may have some or all its diagonal entries equal to 0. Sage offers the command `diagonal_matrix([a_1, a_2, ..., a_n])` to create a diagonal matrix with diagonal entries $a_1, a_2, \ldots, a_n$ and 0 elsewhere.

  Here is an example of a 3×3 diagonal matrix with entries 2, 4, and 6 on the main diagonal.

  ```
  diagonal_matrix([2, 4, 6])
  ```

  Diagonal matrices are simple to multiply and often appear in eigenvalue problems which we will see in upcoming chapters. To check if a matrix is diagonal, the method `is_diagonal()` can be used.

```
d=diagonal_matrix([1, 3, 5])
d.is_diagonal()
```

- An *identity matrix* is a diagonal matrix with all its diagonal elements equals to 1. The command `identity_matrix(n)` returns an identity matrix of dimension $n \times n$. Here is an example of a 5×5 identity matrix.

```
identity_matrix(5)
```

- Although the *ones* matrix is not a predefined matrix in Sage, it is common in other computational frameworks. Essentially, it is a matrix where every entry is 1. It can be created by leverage the built-in list duplication operator `*` applied on a list of ones.

```
# 2x4 matrix with all entries equal to 1
matrix([[1]*4]*2)
```

- Two other common types of square matrices are the *Upper* and *Lower* triangular matrices. A square matrix is an upper triangular matrix if all entries below the diagonal are zero. The following is an example of a 3×3 upper triangular matrix.

```
Matrix([[1, 2, 3], [0, 4, 5], [0, 0, 6]])
```

  Note that Sage does not have these predefined commands to create such matrices, or check if a square matrix is of either type (upper or lower triangular). They can however be obtained by leveraging the `LU()` method that we will see later on.

6.2 Operations With Matrices

Matrices can be added, scaled by a scalar (scalar multiplication), or multiplied with other matrices (matrix multiplication). Addition and scalar multiplication are performed element-wise, analogous to the corresponding operations on vectors.

```
# Defining two 3x3 matrices
A = matrix([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
B = matrix([[9, 8, 7], [6, 5, 4], [3, 2, 1]])

# Matrix addition.
A + B
```

```
# Matrix subtraction
A - B
```

```
# Scalar multiplication
3 * A
```

Matrix multiplication is an operation between two matrices where the rows of the first matrix are multiplied (*dot product*) by the columns of the second matrix to produce the matrix product. For the multiplication to be defined, the number of columns in the first matrix must equal the number of rows in the second matrix. The resulting matrix has the same number of rows as the first matrix and the same number of columns as the second matrix.

```
# Matrix-matrix multiplication
C = matrix([[1, 4, 7, 2, 9],
            [0, 3, 5, 8, 6],
            [9, 1, 0, 4, 2]])
A * C
```

In the case of multiplying a matrix by a vector, the vector is treated as a matrix with a single column. The output however is of type `Vector` and is *not* of type `Matrix`.

```
# Matrix multiplication with vector [3, 3, 3]
v = vector([3]*3)
A * v
```

Note that matrix addition is commutative and associative, meaning that the order of addition does not affect the result. However, matrix multiplication is not commutative; in general, $AB \neq BA$. Matrix multiplication is associative, so $(AB)C = A(BC)$, and it distributes over addition: $A(B + C) = AB + AC$. Also, the identity matrix serves as the multiplicative identity for matrices, whereas the zero matrix serves as the additive identity.

6.3 Transpose, Conjugate and Trace

The transpose of a matrix is obtained by flipping it over its diagonal, swapping rows with columns. The transpose is denoted as A^T and can be computed in Sage using the `transpose` method:

```
A = matrix([[2, 3, 5], [7, 11, 13], [17, 19, 23]])
A.transpose()
```

A matrix A is *symmetric* if it is equal to its transpose, i.e., $A = A^T$. Sage offers `is_symmetric()` method to check if a matrix is symmetric.

```
A.is_symmetric()
```

For matrices with complex entries, the complex conjugate is found by taking the conjugate of each individual entry. This operation is often used in conjunction with the transpose to form the conjugate transpose (also called the Hermitian transpose)

```
A = matrix(CC, [[1 + 2*I, 3 - I], [-2*I, 4]])
A.conjugate()
```

Note that matrices with Complex values, and computing conjugates are very common in signal processing and quantum mechanics.

Just like vectors, matrices also has a norm quantity defined. But unlike vectors, there are several types of norms for matrices. The method `norm` in Sage returns the Frobenius norm which is defined as the square root of the sum of the squares of all the entries in the matrix.

```
# The norm of a matrix
A = matrix([[1, 2], [3, 4]])
```

```
A.norm()
```

The *rank* of a matrix is the dimension of the vector space spanned by its rows (row rank) or columns (column rank). It measures the number of linearly independent rows or columns and is used to determine whether a linear system has unique, infinite, or no solutions. In Sage the rank of a matrix is computed with the `.rank()` method:

```
# rank=2: rows/columns are linearly dependent
B = matrix([[1,2,3],[2,4,6],[1,0,1]])
B.rank()
```

A matrix is called *full rank* when its rank equals the smaller of the number of rows and columns.

```
# a full rank matrix with rank=3
C = matrix([[1,0,0],[0,1,0],[0,0,1]])
C.rank()
```

The *trace* of a square matrix $A = [a_{ij}]$ of order $n \times n$, denoted by $\mathrm{tr}(A)$, is defined as the sum of the entries along its main diagonal: $\mathrm{tr}(A) = \sum_{i=1}^{n} a_{ii}$.

The trace has many useful properties, such as linearity and invariance under similarity transformations:

- $\mathrm{tr}(A + B) = \mathrm{tr}(A) + \mathrm{tr}(B)$
- $\mathrm{tr}(kA) = k\,\mathrm{tr}(A)$ for any scalar k
- $\mathrm{tr}(AB) = \mathrm{tr}(BA)$ for any square matrices A and B of the same size

The built-in Sage method `.trace()` is used to compute the trace of a square matrix as shown in the example below.

```
A.trace()
```

Chapter 7

Determinants

Matrices also support non-arithmetic, unary operations; operation that are applied to the matrix itself and its entrees. Two such operations are the *determinant* of a *square* matrix and its *inverse*.

In this chapter, we will learn about a fundamental concept in linear algebra that is determinants, which provides important insights into the properties of matrices, and the linear transformation or the system of linear equations they represent.

7.1 Determinant and Inverse

The determinant of a square matrix A is a number $det(A)$ that must satisfies the following key properties:

- Adding a row to another row on A does not change $det(A)$.
- Scaling a row of A by a scalar k multiplies $det(A)$ by k.
- Swapping two rows of a matrix A multiplies its determinant by -1.
- The determinant of the identity matrix is 1.

Based on the above definition, the following properties can be established:

- The determinant of the zero matrix is 0.
- If two rows of the matrix are identical, its determinant is 0.
- If a matrix has a row of all zeros, its determinant is 0.
- The determinants of a matrix and its transpose are equal: $\det(A^T) = \det(A)$.
- The determinant of a diagonal, or triangular matrix (upper or lower) is the product of its diagonal entries.
- The determinant of the product of two matrices equals the product of their determinants: $\det(AB) = \det(A) \cdot \det(B)$.
- Performing a *row replacement* on A does not change $det(A)$. In other words, replacing any one row by itself, plus a multiple of another row: $\forall k \neq 0, R_i \leftarrow R_i + k \cdot R_j$ (with $i \neq j$).

Note that all the above properties also hold true and extends for column operations as well.

The determinant of a matrix A is a scalar value $det(A)$ (also denoted $|A|$) that tells important information about the linear transformation represented by that matrix, such as whether tha matrix is invertible or how it scales space during transformations. In geometric terms for 2D and 3D space, the determinant of a square matrix measures the *signed area* of the parallelogram (in 2D) or the *signed volume* of parallelepiped (in 3D) formed by the row (or column) vectors of the matrix.

7.1.1 Determinant of Matrix

In Sage, the determinant of a matrix can be computed using the `det` command:

```
A = matrix([[2, 3, 5], [7, 11, 13], [17, 19, 23]])
A.det()
```

A matrix is *singular* if it does not have an inverse, which occurs when its determinant is zero. In Sage, to check whether a matrix is singular or not the method `is_singular()` can be used. Note that just as with the determinant, this method is only applicable to square matrices.

```
M = matrix([[1, 2], [2, 4]])
M.is_singular()
```

The matrix M is singular because its determinant is zero, hence does not have an inverse. Note that Sage offers several other methods to help check various properties of matrices related to determinants and inverses.

Note that Sage defines a method `is_square()` that returns `True` if the matrix is square.

```
M.is_square()
```

7.1.2 Inverse of Matrix

The inverse of a matrix A is denoted as A^{-1} and is defined as the matrix that, when multiplied with A, yields the identity matrix. The inverse exists only if the determinant of the matrix is non-zero. In Sage, the inverse of a matrix can be computed using the `inverse` method:

```
A.inverse()
```

The method `is_invertible()` returns `True` if the matrix has an inverse.

```
M.is_invertible()
```

The inverse of a matrix has several important properties:

- The inverse of the identity matrix is itself: $I^{-1} = I$.
- The inverse of a product of matrices is the product of their inverses in reverse order: $(AB)^{-1} = B^{-1}A^{-1}$.
- The inverse of a transpose is the transpose of the inverse: $(A^T)^{-1} = (A^{-1})^T$.
- The inverse of an inverse returns the original matrix: $(A^{-1})^{-1} = A$.

Chapter 8

Inverses

PLACEHOLDER

8.1 Inverses of Matrices

The inverse of a matrix A is denoted as A^{-1} and is defined as the matrix that, when multiplied with A, yields the identity matrix. The inverse exists only if the determinant of the matrix is non-zero.

The inverse of a matrix has several important properties:

- The inverse of the identity matrix is itself: $I^{-1} = I$.
- The inverse of a product of matrices is the product of their inverses in reverse order: $(AB)^{-1} = B^{-1}A^{-1}$.
- The inverse of a transpose is the transpose of the inverse: $(A^T)^{-1} = (A^{-1})^T$.
- The inverse of an inverse returns the original matrix: $(A^{-1})^{-1} = A$.

In Sage, the inverse of a matrix can be computed using the `inverse` method:

```
A = matrix([
    [4, 0, 0],
    [1, 1, 7],
    [0, 9, 3]
])
A.inverse()
```

Exponent notation also produces the same effect.

```
A^-1
```

8.1.1 Minors, Cofactors and the Adjugate Matrix

The *adjugate* of a matrix A is the transpose of its cofactor matrix C: $adj(A) = C^T$. In Sage, the `adjugate` method returns the adjugate matrix.

```
A.adjugate()
```

Currently, Sage does not have an implementation to directly extract the *minors* of a matrix. To compute a minor M_{ij} of a matrix A, Sage offers `delete_rows()` and `delete_columns()` methods which can be used to delete a specific row i and column j and obtain a submatrix whose determinant yields a minor M_{ij}, which then can be used to compute the *cofactor* $c_{ij} = (-1)^{i+j} M_{ij}$.

```
# By Removing first row and second column
minor_1_2 = det(A.delete_rows([0]).delete_columns([1]))
print("m_12:", minor_1_2)

# Cofactor c_12
cofactor_1_2=(-1)^(1+2) * minor_1_2
print("c_12:", cofactor_1_2)
```

Generally, given a square matrix of order $n \times n$, the cofactors are obtained by varying the row and column indices i and j from 1 to n. These computed cofactors constitute the entries of the cofactor matrix $C = \left[c_{ij}\right]$.

```
n = A.nrows()

# Cofactor matrix
C = matrix([
    [(-1)^(i + j) * det(A.delete_rows([i -
        1]).delete_columns([j - 1])) for j in range(1, n +
        1)]
    for i in range(1, n + 1)
])
C
```

Note that the transpose of the cofactor matrix is the adjugate matrix

```
A.adjugate() == C.transpose()
```

The Adjugate matrix can then be used to compute the inverse of a non-singular matrix using the formula: $A^{-1} = \frac{1}{\det(A)} \cdot \operatorname{adj}(A)$.

```
A.inverse() == A.adjugate() / A.det()
```

Note that the although methods `minor()` and `cofactor()` are currently not implemented in Sage, leveraging the command `adj` can help compute them, like shown below.

```
cof = A.adjugate().transpose()

# Cofactor c_12
cofactor_1_2 = cof[0, 1]
print("c_12:", cofactor_1_2)

# Minor M_12
minor_1_2 = cofactor_1_2 / ((-1)^(1 + 2))
print("m_12:", minor_1_2)
```

8.1.2 Row Reduction

Row reduction can also be used to produce the inverse of a matrix. First we must augment the original matrix with the identity matrix.

```
AI = A.augment(identity_matrix(3))
AI
```

Then by invoking `rref` the right sub-matrix is now the inverse.

```
rrefAI = AI.rref()
rrefAI
```

To select for the inverse matrix we can use the `submatrix` method to return the inverse alone.

```
A.inverse() == rrefAI.submatrix(0,3)
```

8.1.3 Compare Methods

These methods can have diffrent Let's see what happens with a matrix that is not invertible.

```
M = matrix([
    [7, 7, 1],
    [2, 7, 1],
    [5, 7, 1]
])
M.det()
```

In the case of a matrix that is not invertible, the `inverse` method will throw an error.

```
try:
    M.inverse()
except Exception as e:
    print(e)
```

Using the `adjugate` method and dividing by the determinate will also yield an error.

```
try:
    M.inverse() == M.adjugate() / M.det()
except Exception as e:
    print(e)
```

While the `rref` method will return an inconsistent matrix.

```
M.augment(identity_matrix(3)).rref()
```

Chapter 9

Adjoint Matrix

In this chapter, we will learn about a ...

9.1 Adjoint Matrix

In some fields, working over complex vector spaces is necessary and common. In these contexts, the *adjoint matrix* of a matrix A refers to its *conjugate transpose*. It is obtained by first taking the complex conjugate of each entry of A, and then transposing the resulting matrix.

The adjoint of a matrix A is denoted by A^* and defined as: $A^* = \overline{A}^T$ where $\overline{A}$ represents the complex conjugate of A.

It is important to note that this meaning of *adjoint* differs from the *adjugate* matrix, which will be defined later as the transpose of the cofactor matrix. The two concepts are conceptually distinct:

- *Adjoint:* involves complex conjugation and transposition.
- *Adjugate:* involves cofactors and determinants (used in matrix inversion).

Sage has built-in method `conjugate_transpose` to compute the adjoint matrix as follows:

```
# Define a 2x2 complex matrix
C = matrix(CDF, [[1 + 2 * I, 3], [4, 5 - I]])
C.conjugate_transpose()
```

9.1.1 Minors, Cofactors and the Adjugate Matrix

The *adjugate* of a matrix A is the transpose of its cofactor matrix C: $adj(A) = C^T$. In Sage, the `adjugate` method returns the adjugate matrix.

```
M.adjugate()
```

Currently, Sage does not have an implementation to directly extract the *minors* of a matrix. To compute a minor M_{ij} of a matrix A, Sage offers `delete_rows()` and `delete_columns()` methods which can be used to delete a specific row i and column j and obtain a submatrix whose determinant yields a minor M_{ij}, which then can be used to compute the *cofactor* $c_{ij} = (-1)^{i+j} M_{ij}$.

```
A = matrix([[1, 2, 3], [5, 7, 11], [13, 17, 19]])

# By Removing first row and second column
minor_1_2 = det(A.delete_rows([0]).delete_columns([1]))
print("m_12:", minor_1_2)

# Cofactor c_12
cofactor_1_2=(-1)^(1+2) * minor_1_2
print("c_12:", cofactor_1_2)
```

Generally, given a square matrix of order $n \times n$, the cofactors are obtained by varying the row and column indices i and j from 1 to n. These computed cofactors constitute the entries of the cofactor matrix $C = \left[c_{ij}\right]$.

```
n = A.nrows()

# Cofactor matrix
C = matrix([
    [(-1)^(i + j) * det(A.delete_rows([i -
        1]).delete_columns([j - 1])) for j in range(1, n +
        1)]
    for i in range(1, n + 1)
])
C
```

Note that the transpose of the cofactor matrix is the adjugate matrix

```
A.adjugate() == C.transpose()
```

The Adjugate matrix can then be used to compute the inverse of a non-singular matrix using the formula: $A^{-1} = \frac{1}{\det(A)} \cdot \operatorname{adj}(A)$.

```
A.inverse() == A.adjugate() / A.det()
```

Note that the although methods `minor()` and `cofactor()` are currently not implemented in Sage, leveraging the command `adj` can help compute them, like shown below.

```
cof = A.adjugate().transpose()

# Cofactor c_12
cofactor_1_2 = cof[0, 1]
print("c_12:", cofactor_1_2)

# Minor M_12
minor_1_2 = cofactor_1_2 / ((-1)^(1 + 2))
print("m_12:", minor_1_2)
```

9.2 Alternative Computation of the Adjoint, Cofactors, and Minors

9.3 Historical Note: Adjoint vs. Adjugate

Chapter 10

LU Decomposition

In this chapter, we will learn about a LU decomposition.

10.1 LU Decomposition

The LU decomposition of a matrix is a factorization of the matrix into a product of a lower triangular matrix L and an upper triangular matrix U. This decomposition is useful for solving systems of linear equations and computing the determinant of the matrix.

In Sage, the LU decomposition can be computed using the `LU()` method (note that Sage implementation of this method returns three matrices: the lower triangular matrix L, the upper triangular matrix U, and an extra permutation matrix P):

```
A = matrix([[41, 37, 47], [61, 31, 59], [71, 73, 79]])
P, L, U = A.LU()
P # Permutation (or the pivot) matrix
```

The product of the matrices P, L, and U yields the original matrix A. The matrix P is called the *Pivot* (or the permutation) matrix, which always has a determinant of 1 (has exactly one 1 in every row and column). The matrix L is the Lower triangular matrix L. its determinant is equal to the product of the diagonal entries.

```
print(L, end="\n\n")
print(L.determinant())
```

Similarly, the matrix U is the Upper triangular matrix U. its determinant is also equal to the product of the diagonal entries.

```
# The upper triangular matrix U
U
```

Note that the LU decomposition is not unique, and there are many different ways to perform it. For instance, we can choose to decomposition with a pivot that has a non-zero determinant (also equal to 1 in this case):

```
P, L, U = A.LU(pivot='nonzero')
print(P)
print('-'*7+'\n',P.det())
```

With *partial* pivoting, every entry of L will have absolute value of 1 or less.

```
P, L, U = A.LU(pivot='partial')
L
```

Additionally, Sage offers different ways to format the display of the result of the LU decomposition.

```
P, L, U = A.LU(format='plu')
print(P, end="\n\n")
print(L, end="\n\n")
print(U, end="\n\n")
```

And for a slightly compact format:

```
P, M = A.LU(format='compact')
print(P, end="\n\n")  # only display the diagonal entrees
print(M, end="\n\n")  # combines the upper and lower
    triangular matrices
```

Chapter 11

Equations of Lines and Planes

PLACEHOLDER.

Chapter 12

Orthogonality

PLACEHOLDER.

12.1 Normalized Vector

A normalized vector is a vector that has been scaled to have a norm of 1 (hence the name *unit vector*). This is often done to simplify calculations or to ensure that the vector represents only a direction without any magnitude. Normalizing a vector involves dividing each component of the vector by its norm (or magnitude).

```
# Normalizing a 3D vector
v = vector([3, 4, 0])
w = v.normalized()

print("vector(v):", v, "␣with␣norm:", v.norm())
print("Normalized␣vector(w):", w, "␣with␣norm:", w.norm())
```

The normalized vector is a unit vector that points in the same direction as the original vector. Normalizing a vector means scaling it so that it has length 1 while preserving its direction. This is useful in applications such as directional vectors, unit vectors, and simplifying computations involving angles or projections.

Chapter 13

Linear Transformation From R Pow N to R Pow M

PLACEHOLDER.

Chapter 14

Eigenvectors and Eigenvalues

PLACEHOLDER.

14.1 Eigenvectors

An eigenvector, of a matrix, is a non-zero vector where multiplication by said matrix does not change its direction. This sage block will generate a gif to demonstrate; you can define a new matrix or leave the values in.

All eigenvectors exist along the green lines of the gif.

```
A = matrix([
        [6, 0],
        [4, 2]
        ])
def EigenVis2D(T):
    if T.dimensions() != (2,2): raise Exception('Input␣must␣
        be␣a␣2x2␣matrix') #Tests for an appropriate matrix

    t = var('t') # Symbolic variable for parametric_plot
    V = matrix([[n1, n2] for n1 in range(-5,6) for n2 in
        range(-5,6)]).transpose() #Creates a matrix of the
        vectors, as columns, undergoing transformation,
        these will be represented as dots
    I = identity_matrix(2)
    if not T.eigenvectors_right()[0][0] in RealField():
        print("Complex␣eigenvalues,␣unable␣to␣plot␣
            eigenvectors␣and␣eigenspaces.")
        I_plot = list_plot(V.columns(), figsize=10)
    else:
        E_vectors = [vector for vector_list in
            [eigen_space[1] for eigen_space in
            T.eigenvectors_right()] for vector in
            vector_list ] # collects eigenvectors into a
            list E
        I_plot = list_plot(V.columns(), figsize=10) +
            sum(parametric_plot(t*(e/e.norm()), (t, -7, 7),
            color='green') for e in E_vectors) + sum(plot(e,
            color='green',) for e in E_vectors)

    # Display the numeric info about the transformation
    print("Eigen␣spaces:␣(Eigen␣Value,␣[Eigen␣Vectors])")
    %display latex
    show(var('T'), "␣=␣", T)
    for e_space in T.eigenvectors_right(): show(e_space[0:2])
    %display plain

    T -= I
    Frames = [ # All frames of the animation are collected
        into a list
        I_plot + list_plot(((T*(n/30) + I)*V).columns(),
            color='red', ymin=-5, ymax=5, xmin=-5, xmax=5,
            figsize=6) for n in range(31)
    ]

    Frames = Frames + [Frames[-1]] * 16 # Padding extra end
        frames to make the animation stick to view the end
        of the transformation

    GIF = animate(Frames)
    GIF.show(delay=10)
EigenVis2D(A)
```

14.1.1 Eigenvectors of Matrices

We will use the matrix A to demonstrate.

```
A = matrix([
    [6, 0],
    [4, 2]
    ])
A
```

We can find its eigenvectors with the `eigenvectors_right` method.

```
A.eigenvectors_right()
```

We will focus on the first tuple in the list that was returned.

Chapter 15

Diagonalization

PLACEHOLDER.

Chapter 16

General Vector Spaces

PLACEHOLDER.

Chapter 17

Linear Independence

PLACEHOLDER.

Chapter 18

General Linear Transformations

PLACEHOLDER.

References

Most of the content in this book is based on the Linear Algebra lectures taught by Professor Hellen Colman at Wilbur Wright College. We focused our efforts on creating original work, and we drew inspiration from the following sources:

Kuttler, K. A first course in linear algebra. Mathematics LibreTexts. https://math.libretexts.org/Bookshelves/Linear_Algebra/A_First_Course_in_Linear_Algebra_(Kuttler)?readerView, 17 Sept 2022. SageMath, the Sage Mathematics Software System (Version 10.2), The Sage Developers, 2024, https://www.sagemath.org. Beezer, Robert A., et al. The PreTeXt Guide. Pretextbook.org, 2024, https://pretextbook.org/doc/guide/html/guide-toc.html. Zimmermann, Paul. Computational Mathematics with SageMath. Society For Industrial And Applied Mathematics, 2019.

Colophon

PDF Download : This book was authored in PreTeXt and is available for free download in PDF format by clicking here[1].

[1] github.com/SageMathOER-CCC/linear-algebra-with-sage/blob/release/output/print/manuscript.pdf

Index

arithmetic operators, 1

data types
- boolean, 4
- dictionary, 5
- integer, 3
- list, 4
- set (Python), 4
- string, 4
- symbolic, 3
- tuple, 4

debugging
- attribute error, 13
- logical error, 14
- name error, 13
- syntax error, 12
- type error, 13
- value error, 14

error message, 12

functions (programming), 8

identifiers, 3

iteration
- for loop, 6
- list comprehension, 7

Matrices, 19, 48, 49
- Adjoint, 47
- Adjugate
  - Cofactor, 43, 47
- column, 20
- columns, 20
- Complex Conjugate, 39
- Determinant, 41
- diagonal, 21
- dimensions, 21
- Field, 22
- indexing, 20
- Inverse, 42
- Multiplication, 38
- ncols, 21
- Norm, 39
- nrows, 21
- Operations
  - Arithmetic, 38
- Properties, 39
- Rank, 40
- Ring, 22
- row, 20
- rows, 20
- Symmetric, 39
- Trace, 40
- Transpose, 39
- Upper Triangular Matrix
  - Lower Triangular Matrix, 38, 51
- Zero Matrix
  - Diagonal Matrix, 37

run code
- CoCalc, 16
- Jupyter Notebook, 16
- local, 16
- SageMath worksheets, 16

Vectors, 17
- Arithmetic Operations, 35
- Complex, 18
- Complex Conjugate, 18
- Cross Product, 36
- Field, 18
- Operations
  - Dot/Scalar Product, 35
- Ring, 18
- Scalar Multiplication, 35
- Unit Vector
  - Normalization, 55
- Visual Representation, 18