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

February 28, 2026

Authors: Allaoua Boughrira, Hellen Colman, Michael Kattner, Samuel Lubliner
Significant Contributors: Alex Koci, Sagha Axelsson
Cover Design: Hellen Colman, made with SageMath — licensed under the CC-BY-NC-SA 4.0 License

Edition: February 2026

Website: GitHub Repository

Preface

This book is the second in a series of textbooks designed to provide instruction on using SageMath to study various subjects in undergraduate mathematics. In this volume, we focus on using Sage in a first-year course on Linear Algebra.

The goal is not to present a comprehensive treatment of Linear Algebra, but rather to offer a clear and accessible introduction to Sage's robust capabilities for working with vectors, matrices, systems of equations, and linear transformations.

This book was written by undergraduate students at Wright College who were enrolled in my Math 299 class, Writing in the Sciences.

For many years, I have been teaching Linear Algebra using the open source mathematical software SageMath. Despite the fabulous capabilities of this software, students were often frustrated by the lack of specific documentation geared towards beginning undergrad students in Linear Algebra.

This book was born out of this frustration and the desire to make our own contribution to the Open Education movement, from which we have benefitted greatly. In the context of Open Pedagogy, my students and I ventured into a challenging learning experience based on the principles of freedom and responsibility. Each week, students wrote a chapter of this book. They found the topics and found their voice. We critically analyzed their writing, and they edited and edited again and again. They wrote code, tested it and polished it. In the process, we all learned so much about Sage, and we found some bugs in the software that are now in the process of being fixed thanks to its very active community of developers.

The result is the book we dreamed of having when we first attempted Linear Algebra with Sage. Our book is intended to provide concise and complete instructions on how to use different Sage functions to solve problems in Linear Algebra. Our goal is to streamline the learning process, helping students focus more on mathematics and reducing the friction of learning how to code. Our textbook is interactive and designed for all math students, regardless of programming experience. Rooted in the open education philosophy, our textbook is, and always will be, free for all.

I am very proud of the work of my students and hope that this book will serve as inspiration for other students to take ownership of a commons-based education. Towards a future where higher education is equally accessible to all.

Hellen Colman
Chicago, February 2026

Acknowledgements

We would like to acknowledge the following peer-reviewers:

- Justin Lowry, Wright College
- Vartuyi Manoyan, Wright College

We would like to acknowledge the following proof-readers:

- Marcy Rae Henry, Wright College

The making of this book is supported in part by the Wright College Open Educational Resource Expansion grant from the Secretary of State/Illinois State Library. Many thanks to Tineka Scalzo, the Wright College librarian, for her unwavering advocacy and invaluable support.

We extend our sincere gratitude to David Potash, President of Wright College (2013-2024), for his steadfast moral and material support since the project's inception. His encouragement, demonstrated through attending student presentations and securing grant funding on their behalf, along with his scholarly vision and generosity, has been instrumental in bringing this project to fruition.

From the Student Authors

This book reflects a shared commitment to collaboration and the ideals of Open Education. Its purpose is to broaden access to higher learning by offering clear and welcoming materials that help students engage with open-source tools in mathematics.

We are especially grateful to Hellen Colman, Professor of Mathematics at Wright College, our co-author, and our guiding compass and lighthouse, helping us navigate forward with purpose and direction. Her depth of mathematical knowledge helped maintain the precision and significance of the content. Drawing inspiration from her Linear Algebra online lectures and at the City Colleges of Chicago-Wilbur Wright College, this work has benefited greatly from her mentorship and encouragement. Her confidence in us played a key role in influencing our thinking and methods, from the Linear Algebra coursework through the development of this OER.

We extend our appreciation to our peer reviewers and proofreaders, whose careful and thorough work helped maintain the precision and readability of this textbook. Their efforts were vital to the successful completion of this project.

Developing this text was a true team endeavor, shaped by the dedication and motivation of many contributors. The standard of the final result speaks to our combined efforts and shared passion. Most importantly, we extend our sincere appreciation to all who contributed to making this endeavor become a reality. Their commitment and spirit of cooperation have left a meaningful and enduring mark on this work and on Open Education as a whole.

We are also thankful for the many skilled developers and mathematicians across open-source communities. The PreTeXt community was central to our writing workflow, and the SageMath community contributed invaluable disciplinary insight. We sincerely acknowledge everyone involved in building Sage, as well as the creators of PreTeXt.

Allaoua Boughrira, Michael Kattner, and Samuel Lubliner

Authors and Contributors

ALLAOUA BOUGHRIRA
Mathematics
Wright College
a.boughrira@gmail.com

HELLEN COLMAN
Math Department
Wilbur Wright College
hcolman@ccc.edu

MICHAEL KATTNER
Mathematics
Wilbur Wright College
MDKattner@gmail.com

SAMUEL LUBLINER
Computer Science
Wilbur Wright College
sage.oer@gmail.com

ALEX KOCI (CONTRIB.)
Computer Engineering
Wilbur Wright College
kocialex04@gmail.com

SAGHA AXELSSON (CONTRIB.)
Bioengineering
Harold Washington College
axelssonsagha@gmail.com

Contents

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. 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:

- + 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 also introduces new 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 Exception Handling

Similar to `if` statements, `try` and `except` blocks allow for the conditional execution of code in the event of an exception in the code of the `try` block. Then at the end of execution, regardless of if an exception 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. Because math terminology is not always standardized, it is important to refer to the documentation to understand the exact functionality of these built-in functions and 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 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 and the Reference Manual for detailed information.

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

You can find Sage documentation 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??
```

1.9 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:

```python
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.
2. Click on Sage on CoCalc.
3. Create a CoCalc account.
4. Go to Your Projects 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.
9. Alternatively, you can run Sage code in a Jupyter Notebook for additional features.
10. If you are feeling adventurous, you can install Sage 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.

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 is 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 *base rings* based on the datatype of the components of the vector. While working in a specific ring, we need to explicitly pass the ring 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 Integer numbers. Similarly, QQ is for Rational numbers, RR for Real numbers, and CC for Complex numbers. The method `base_ring` returns the *base ring* of the vector.

```
u.base_ring()
```

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

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

Sage also supports complex vectors. 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 whose components are complex numbers, we can either explicitly specify the ring as CC, or implicitly by using complex numbers as components of the vector as seen 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 the Sage default `plot` method.

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

Sage also provides alternative methods to represent vectors visually. For instance, the `arrow` method can be used to create an arrow representation of a vector in 3D space like in the following example.

```
show(arrow((0, 0, 0), (2, 3, 4), color='blue', width=2))
```

Similarly, the `arrow2d` method can be used to create an arrow representation of a vector in 2D space like in the following example.

```
show(
    arrow2d((0, 0), (0.5, 1), color='red', width=2),
    xmin=-0.2, xmax=1, ymin=-0.2, ymax=1.2,
)
```

Note that both methods take points coordinates as input (the beginning and end of the vector), as it can also take vectors as input. In which case, the arrow goes from the end of the first vector to the end of the second vector, like in the following example.

```
show(arrow(vector([1,1,1]), v, color='blue', width=2),
    aspect_ratio=1)
```

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 1s, 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 5×20 matrix containing the first 100 integers, where only the number of rows is specified.

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

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

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

Sage also allows for constructing 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.

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

We will use the following matrix M to show how Sage can identify its different parts.

```
M = matrix(2,3, [0,1,2,3,4,5])
M
```

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(2)  # 3rd 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()
```

Sage defines the method `is_square()` that returns `True` if the matrix is square. In this case, it returns `False` because matrix M is a 2×3 matrix.

```
M.is_square()
```

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).

Sage also offers the method `submatrix()` that takes the starting row and column indices, and the number of rows and columns to include in the resulting submatrix. The following example shows how to extract a submatrix starting from the first row and first column, and including two rows and two columns.

```
# extract the same top-left 2x2 submatrix
A.submatrix(0, 0, 2, 2)
```

Another indirect way to extract a submatrix is to use the methods `delete_rows()` and `delete_columns()`. For instance, to extract the same top-left 2×2 submatrix of the previous example, we can use the following command.

```
# delete the 3rd row and 3rd column
A.delete_rows([2]).delete_columns([2])
```

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 *Base Rings*, 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 ring is omitted, Sage will simply infer the ring from the datatype of the matrix entries. Sage matrices supports the same rings 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 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 we pass 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 implicitly defines 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 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,-9,9), (y,-9,9), (z,-9,9),
    color='blue')
b = implicit_plot3d(eq2, (x,-9,9), (y,-9,9), (z,-9,9),
    color='red')
c = implicit_plot3d(eq3, (x,-9,9), (y,-9,9), (z,-9,9),
    color='green')
A = a + b + c
A.show(viewpoint=[[0,-9,0],60])
```

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,-9,9), (y,-9,9), (z,-9,9),
    color='blue')
b = implicit_plot3d(eq2, (x,-9,9), (y,-9,9), (z,-9,9),
    color='red')
c = implicit_plot3d(eq3, (x,-9,9), (y,-9,9), (z,-9,9),
    color='green')
A = a + b + c
```

```
A.show(viewpoint=[[0,-9,0],60])
```

By rotating the graph, we can visualize the infinitely many solutions of this system, represented by the line where the three planes intersect.

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,-9,9), (y,-9,9), (z,-9,9),
    color='blue')
b = implicit_plot3d(eq2, (x,-9,9), (y,-9,9), (z,-9,9),
    color='red')
c = implicit_plot3d(eq3, (x,-9,9), (y,-9,9), (z,-9,9),
    color='green')
a + b + c
```

In this last case, we can observe that two of the planes are parallel, so there is no intersection among the three planes.

Chapter 4

System of Equations with Matrices

Systems of linear equations can be represented as matrices to make solving them more efficient as the number of variables increases. This chapter shows how to use Sage to solve systems using the matrix representation.

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.

In the following sections, we will demonstrate how to use Sage to carry out these steps, using our example from **Case I**:

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

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 first 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
```

Then, 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 enter all the coefficients of the augmented matrix directly.

```
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

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.

We can perform some operations in a matrix to reduce it to this form. The following elementary operations transform a system of equations into another one with the same solution:

Scaling A row is multiplied by a nonzero constant.

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

Interchange A row is swapped with another.

The `swap_rows(r1, r2)` method preforms this operation on a matrix by swapping rows `r1` and `r2`.

Replacement A row is added to a multiple of another.

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

Keep in mind that all of the previous methods use zero-based indexing.

We will now perform elementary operations in our augmented matrix to transform it into reduced row echelon form. Recall that our augmented matrix is:

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

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 the matrix is in reduced row echelon form, we can translate it back into a system of equations to explicitly visualize the solution set of the system of equations:

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

Alternatively, Sage has a built in method to do all the previous steps directly. 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()
```

Observe that when we translate it back into a system of equations, we obtain:

$$\begin{cases} x + 2z = 4 \\ y - z = 0 \\ 0 = 0 \end{cases} \quad \Rightarrow \quad \begin{cases} x = -2z + 4 \\ y = z \\ z = z \end{cases}$$

We will see in the next section how to use Sage to interpret this solution.

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()
```

Observe that when we translate it back into a system of equations, we obtain:

$$\begin{aligned} x + 2z &= 4 \\ y - z &= 0 \\ 0 &= 1 \end{aligned}$$

This clearly shows a contradiction.

4.4 Pivots

A leading coefficient in a matrix is the first non-zero entry in a given row. The position of a leading coefficient in the reduced row echelon matrix of A is called a pivot position.

Sage has the `pivot()` method to identify the pivot columns of the reduced row echelon form of a given matrix. Here is an example with **Case I**:

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

Observe that the solution obtained means that there are pivot positions in the first, second and third columns of the reduced matrix. We can easily verify this by revisiting the reduced form of the matrix:

```
Ab.rref()
```

We can clearly see the pivot positions in the first three columns.

Notice that Sage can determine which columns contain the pivot positions in the reduced row echelon form of the matrix without explicitly showing this form.

The pivot positions in the reduced row echelon form of the augmented matrix of a linear system are crucial for determining the nature of its solutions.

Let R and Rb denote the reduced row echelon forms of the coefficient matrix and the augmented matrix, respectively.

- A system is inconsistent exactly when a pivot position appears in the last column of Rb.
- If the system is consistent, then the columns of R containing pivot positions correspond to basic variables (dependent), while the columns without pivot positions correspond to free variables (independent).
- The solution is unique when every column of R contains a pivot position, and there are infinitely many solutions when at least one column of R does not contain a pivot position.

4.4.1 Compatible Unique Solutions (Case I)

Observe that in our previous system there was no pivot position in the last column, so the system was compatible. Moreover, every column of the reduced row echelon form of the coefficient matrix had a pivot position, so the system has a unique solution.

Recall that in Section 3.2, this unique solution was explicitly obtained.

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

This coincides with the pivot criterion, as it implies a unique solution for this matrix.

4.4.2 Compatible Infinitely Many Solutions (Case II)

Let's find the pivot columns for our **Case II** system.

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

We found that there are pivot positions in the first and second columns.

Since there is no pivot position in the last column, the system is consistent. Because there are columns without pivot positions in the rest of the matrix, the system has infinitely many solutions. The first and second columns correspond to the variables x and y, so these are the basic variables, while z is a free variable. Therefore, the solution can be written as a function of the parameter z.

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

This coincides with the solutions we found in Section 3.2, namely a parameterized solution, but in this case Sage defaults to `r1` as a parameter.

```
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)
```

4.4.3 Incompatible (Case III)

Let's find the pivot columns for our **Case III** system.

```
Ab = matrix([          # case iii
        [1, 2, 0, 4], # x + 2y = 4
        [1, 2, 0, 1], # x + 2y = 1
        [1, 0, 2, 4]  # x + 2z = 4
    ])
Ab.pivots()
```

We found that there are pivot positions in the first, second and fourth columns. In particular, there is a pivot position in the last column so the system is incompatible.

This is again consistent with the solution from Section 3.2, as an empty list of solutions was returned.

```
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)
```

4.5 Matrix Equations

We can represent a system of equations as a matrix equation by writing $Ax = b$ where A is the coefficient matrix, x is the vector of variables, and b is the vector of constants.

$$\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix}$$

We can solve a matrix equation in Sage using the method `solve_right()`. Evaluating `A.solve_right(b)` returns a vector x such that $Ax = b$. The components of this vector are the values of a solution of the system of equations.

4.5.1 Case I

Now, let's solve the **Case I** system applying the matrix equation method.

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

We found that the solution of the system is $x = 4,\ y = 0$, and $z = 0$.

4.5.2 Case II

Now, let's solve the **Case II** system applying the matrix equation method.

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

We found that one solution to the system is $x = 4,\ y = 0$, and $z = 0$. This is just ***one*** particular solution when the parameter is zero. Observe that we do not obtain all the infinitely many solutions of the system using this method.

4.5.3 Case III

Now, let's solve the **Case III** system applying the matrix equation method.

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

Observe that in this case, Sage produces an error message. Nevertheless, the error message states that the system has no solutions.

To avoid the error message and continue the execution of the program, we can write:

```
A = matrix([          # case iii
        [1, 2, 0],
        [1, 2, 0],
        [1, 0, 2]
    ])
b = vector([4, 1, 4])
try:
    A.solve_right(b)
except Exception as e:
    print(e)
```

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 $v \cdot w = w \cdot v = \sum_{i=1}^{n} v_i w_i$. The method `dot_product()` helps calculate the dot product of two vectors in Sage.

```
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 serve as mathematical tools for structuring data, encoding relationships between multiple quantities, representing linear transformations, or solving systems of equations.

In this chapter, we will look at common matrix operations used in linear algebra.

6.1 Special Matrices

Some matrices occur frequently enough to be given special names:

- A *zero matrix* is a matrix in which every entry is 0. 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 *ones matrix* is a matrix in which every entry is 1. The command `ones_matrix(m, n)` creates an m by n ones matrix.

  ```
  # 3x4 ones matrix
  ones_matrix(3,4)
  ```

  The ones matrix can be useful for example to add a constant offset to all entries of a matrix.

  ```
  A = Matrix([[1, 2, 3], [4, 5, 6]])
  # Adding an offset of 4 to all entries in A
  A + 4 * ones_matrix(2, 3)
  ```

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

  ```
  # 4x4 ones matrix
  ones_matrix(4)
  ```

- A *diagonal matrix* is a square matrix that has zero entries everywhere outside the *main diagonal*. Note that a square zero matrix is a diagonal matrix. 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])
  ```

  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 of its diagonal entries equal 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)
  ```

- 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 triangular 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.

Note that Sage also offers a special method `random_matrix()` to generate random matrices. For example, the following command generates a random 3×4 matrix with integer entries between 0 and 9.

```
random_matrix(ZZ, 3, 4, x=10)
```

6.2 Operations With Matrices

Matrices can be added, multiplied 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.

$$\textit{Let } A = \left[a_{ij}\right] \textit{ and } B = \left[b_{ij}\right] \textit{ be } m \times n \textit{ matrices.}$$

$$A + B = \left[a_{ij} + b_{ij}\right]$$

```
# 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
```

$$A - B = \begin{bmatrix} a_{ij} - b_{ij} \end{bmatrix}$$

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

$$kA = \begin{bmatrix} ka_{ij} \end{bmatrix}$$

```
# 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.

$$\textit{Let } C = \begin{bmatrix} c_{ij} \end{bmatrix} \textit{ be an } m \times p \textit{ matrix and } D = \begin{bmatrix} d_{ij} \end{bmatrix} \textit{ an } p \times n \textit{ matrix.}$$
$$CD = \begin{bmatrix} d_{ij} \end{bmatrix}$$
$$\textit{where } d_{ij} = \sum_{k=1}^{p} c_{ik} d_{kj}.$$

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

D = matrix([[1, 4, 7, 2, 9],
            [0, 3, 5, 8, 6],
            [9, 1, 0, 4, 2]])

C * D
```

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
```

6.3 Transpose and Conjugate

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()
```

In some fields like signal processing and quantum mechanics, working over complex vector spaces is necessary and common. The *conjugate transpose* matrix is defined as the transpose of the conjugate matrix. In Sage, the conjugate transpose of a matrix is obtained by first computing its conjugate then taking the transpose.

To find the conjugate of a matrix in Sage, we use the `conjugate()` method.

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

Then, we compute the transpose of that conjugate matrix using the transpose method shown earlier.

```
B.transpose()
```

We can also calculate the conjugate transpose directly using the `conjugate_transpose` method.

```
A.conjugate_transpose()
```

The conjugate transpose of a matrix A is called the Hermitian transpose, Hermititan conjugate, or transjugate and is denoted A^*, A^H or $A^\dagger$.

The conjugate transpose matrix is also called the *Adjoint* matrix; however, this terminology will not be used in this book to avoid confusion with the *Adjugate* concept introduced in chapter 8. Historically, "*Adjoint*" was used to refer to both concepts, see historical notes in chapter 8 for more details.

6.4 Trace and Norm

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

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

```
M = matrix([
    [1, 2],
    [4, 6]
])
M.trace()
```

Sage's method `norm()` calculates different norms of a given matrix. By default, it computes the *2-norm* L_2 (also known as the *spectral norm*).

```
# The Euclidean norm of a matrix
A = matrix(RR, [
    [1, 2, 3],
    [3, 4, 6]
])
A.norm()
```

The L_2 norm of a matrix A is defined as the largest singular value of A, which is the square root of the largest eigenvalue of A^TA. We can verify that the value computed by Sage is indeed the largest singular value of A as follows.

```
# Compute the square root of largest singular values of A
sqrt(max([abs(s) for s in
    list((A.transpose()*A).eigenvalues())]))
```

Note 6.4.1 Default Norm of Matrix in Sage. Sage documentation states that the `norm()` method computes the *Euclidean* norm by default, whereas the actual value being returned is for the spectral norm (*L_2 norm*). Sage documentation also implies that *Euclidean* norm is different from the *Frobenius* norm, which is not the case. In fact, the *Euclidean* norm and the *Frobenius* norm are the same and they are both different from the L_2 norm.

For a matrix $A_{n\times m}$, the *Euclidean Norm* (also known as *Frobenius norm* or the *Hilbert-Schmidt norm*) is a vector-induced norm defined as the square root of the sum of the squares of all its entrees $|A|_F = \sqrt{\sum_{i=1}^{n}\sum_{j=1}^{m}|a_{ij}|^2}$.

```
sum([a**2 for a in A.list()]).sqrt()
```

Which is equivalent to the Euclidean norm of the flattened matrix as a vector in $\mathbb{R}^{nm}$.

```
vector([a for a in A.list()]).norm()
```

To compute the Frobenius norm in Sage directly, we need to explicitly pass *frob* as argument to the `norm()` method.

```
A.norm('frob')
```

The Frobenius norm can also be computed using the trace of the product of the matrix and its transpose $|A|_F = \sqrt{\mathrm{tr(A^TA)}}$.

```
(A.transpose() * A).trace().sqrt()
```

Chapter 7

Determinants

In this chapter we define the determinant of a square matrix, and show how to compute it in Sage.

7.1 Determinants

The determinant of a square matrix A is a number $det(A)$ defined by these properties:

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

In Sage, the determinant of a matrix can be computed using the `det()` method:

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

A matrix is *singular* if 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 *rank* of a matrix is the largest dimension of any of its square submatrix with a non-zero determinant. In Sage the rank of a matrix is computed with the `rank()` method:

```
B = matrix([[1,2,3],[2,4,6],[1,0,1]])
print(B.rank())
```

Another way to get the rank of a matrix is to count how may pivots the matrix has.

```
len(B.pivots())
```

7.2 Cramer’s Rule

Cramer’s rule is a theorem in linear algebra that provides an explicit formula for solving a system of linear equations with as many equations as unknowns, provided that the system’s coefficient matrix is non-singular.

Consider a system of n linear equations with n unknowns represented in matrix form as $Ax = b$, where A is the coefficient matrix, x is the column vector of unknowns, and b is the column vector of constants. According to Cramer’s rule, if A is non-singular, then the system has a unique solution given by $x_i = \frac{\det(A_i)}{\det(A)}$, where A_i is the matrix formed by replacing the i-th column of A with the vector b.

To illustrate Cramer’s rule with an example, let’s consider the system of equations from Subsection 4.5.1. Let’s first start by checking if the coefficient matrix A is non-singular and compute its determinant:

```
# Ax=b
b = vector([0, 4, 4])

A = matrix([
    [0, 1, -1],
    [1, 2, 0],
    [1, 0, 1]
])
A.is_singular()  # Check if A is singular
```

As it turned out, the matrix A is non-singular, let’s then compute its determinant:

```
det_A = A.det()
print(det_A)
```

Next, we compute the determinants of the matrices formed by replacing each column of A with the vector b, and then find the values of the unknowns. For the first variable x_1, we replace the first column of A with b to form A_1, compute its determinant, and then find $x_1 = \frac{\det(A_1)}{\det(A)}$.

```
A1 = matrix(A) # Create a copy of matrix A
A1.set_column(0, b)
det_A1 = A1.det()
x1 = det_A1 / det_A
x1
```

We repeat this process for x_2 by replacing the second column:

```
A2 = matrix(A)
A2.set_column(1, b)
det_A2 = A2.det()
x2 = det_A2 / det_A
x2
```

We repeat the same process for x_3 by replacing the third column as follows:

```
A3 = matrix(A)
A3.set_column(2, b)
det_A3 = A3.det()
x3 = det_A3 / det_A
```

```
x3
```

In the above example, we solved the system of equations of Subsection 4.5.1 using Cramer's rule. To verify the solution, we can substitute the values of x_1, x_2 and x_3 back into the original equation:

```
# Verify the solution
x = vector([x1, x2, x3])

A * x == b # notice the the double equal sign
```

Note that Cramer's rule is an iterative process, hence, can be automated using loops and a reusable function.

Let's create a function `solve_system` that takes as input a coefficient matrix A and a constant vector b, and returns the solution vector x using Cramer's rule. The function will first check if the matrix A is non-singular, and if so, it will compute the determinant of A and the determinants of the matrices formed by replacing each column of A with b. Finally, it will return x, the solution vector.

```
### Define the helper functions
def find_x(A, b, detA, i):
    # this helps solve for a single unknown
    Ai = matrix(A)
    Ai.set_column(i, b)
    return Ai.det() / detA

def solve_system(A, b):
    if A.is_singular():
        # System has no unique solution per Cramer's Rule
        return vector([])

    # Iterate to solve for all unknowns
    x = []
    detA = A.det()
    for i in range(len(b)):
        xi = find_x(A, b, detA, i)
        x.append(xi)

    return vector(x)
```

Which then can be used as follows to solve the given system of equations.

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

x = solve_system(A, b)
x
```

Once again, we can verify the solution by substituting the values of x back into the original equation:

```
A * x == b
```

Here is another example for a singular coefficient matrix where Cramer's rule cannot be applied:

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

x = solve_system(A, b)
x  # returns an empty vector since A is singular
```

Note 7.2.1 Floating-point Equality. Because computers represent real numbers using finite binary precision, the result of a floating-point computation is almost never exact. For this reason, testing whether two floating-point expressions are "exactly equal" will often return `false` even when the values should be mathematically identical. For instance, in the previous 5×5 linear-system example: recomputing the solutions of the system in $\mathbb{R}$ would yield a solution vector with floating-point entries:

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

x = solve_system(A, b)
x  # returns solution vector with floating-point entries
```

However the previous equality comparison `A*x == b` would typically fail because of tiny rounding errors.

```
A * x == b
```

For numerical purposes, instead of checking equality, the correct approach is to compare the difference to a small tolerance (for example, 10^{-12}). If the absolute error is below this threshold, then the values *should* be considered equal.

```
all([abs(e)<1E-12 for e in (A * x - b)])
```

Chapter 8

Adjugate Matrix

In this chapter, we will learn about the adjugate matrix, which is the transpose of its cofactors matrix.

8.1 Adjugate Matrix

For a matrix A, the *adjugate* matrix is the transpose of its cofactor matrix C, $adj(A) = C^T$.

The cofactor matrix is based on cofactors and minors. Sage does not have methods to calculate cofactors and minors, but it can directly calculate the adjugate matrix.

In Sage, the `adjugate()` method returns the adjugate matrix.

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

In the next sections, we will explore how to compute step by step the minors, cofactors, and then the adjugate matrix.

8.2 Minors and Cofactors

A minor M_{ij} of a matrix A is the determinant of the submatrix obtained by deleting the i^{th} row and j^{th} column of A.

Currently, Sage does not implement methods 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}. Below is a demonstration how to compute the minor M_{12} of a given matrix A.

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

```
m_12 = A.delete_rows([0]).delete_columns([1]).det()
m_12
```

The cofactor c_{ij} is computed as $c_{ij} = (-1)^{i+j} M_{ij}$, where M_{ij} is the minor obtained by deleting the i^{th} row and j^{th} column of A. The following computes the cofactor c_{12} using the previously computed minor M_{12}.

```
c_12 = (-1)^(1+2) * m_12
c_12
```

8.3 Cofactors Matrix

Given a square matrix $A_{n\times n} = [a_{ij}]$, its cofactors matrix $C_{n\times n}$ is defined as $C = [c_{ij}]$ where each entry c_{ij} is the cofactor of the entry a_{ij} in A.

We can build the cofactor matrix C by repeating the same last two steps, and varying the row and column indices i and j from 1 to n.

```
A = matrix([
    [1,   2,   3],
    [5,   7,  11],
    [13, 17,  19],
])
n = A.nrows()
C = matrix([[(-1)^(i + j) *
    A.delete_rows([i]).delete_columns([j]).det()
    for j in range(n)]
    for i in range(n)])
C
```

Note 8.3.1 Cofactors and 0-Based Indexing. Recall that Sage uses 0-based indexing of lists, vectors, and matrices. If we were to follow the mathematical notation with 1-based indexing, where i and j vary from 1 to n, the formula for the cofactors entries of C would be written as:

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

Another valid expression while keeping 0-based indexing, and i and j vary from 0 to $n-1$ would be:

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

All these expressions are equivalent and correctly computes the cofactors entries of C.

Finally, we take the transpose of C to get the adjugate matrix.

```
Adj = C.transpose()
Adj
```

Note that the adjugate matrix calculated in this way coincides with the one resulted from the built-in method introduced earlier, and we can verify they are indeed equal.

```
Adj == A.adjugate()
```

8.4 Alternative Minors/Cofactors Computation

Although there are no built-in methods in Sage to directly compute minors and cofactors are currently, we can leverage the command `adjugate` to compute them like shown below.

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

From the cofactors matrix, we then have access to cofactors c_{ij}, and from there to the minors m_{ij} like in the following example.

```
c_12 = C[0, 1]
print("c_12:", c_12)

m_12 = (-1)^(1+2)*c_12 # divide/multiply by +/-1 are
    equivalent
print("m_12:", m_12)
```

8.5 Historical Note: Adjoint vs. Adjugate

The terms ”adjoint” and ”adjugate” are often used interchangeably in linear algebra to refer to the same concept: the transpose of the cofactor matrix of a square matrix. However, historically, ”adjoint” has been used in different contexts, such as in functional analysis to denote a different concept related to linear operators. In modern usage, especially in computational tools like Sage, ”adjugate” is the preferred term for the matrix operation involving cofactors.

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

Chapter 9

Inverse Matrix

PLACEHOLDER

9.1 Inverse Matrix

Let A be a square matrix. A matrix B is called an inverse of A if $AB = I$ and $BA = I$, where I is the identity matrix.

If such a matrix exists, we say that A is invertible and we denote the inverse by A^{-1}.

We can ask Sage whether a matrix is invertible using the `is_invertible()` method:

```
A = matrix([      # case i
        [0, 1, -1],
        [1, 2, 0],
        [1, 0, 1]
    ])
A.is_invertible()
```

In Sage, if a matrix is invertible, the inverse of a matrix can be computed using the `inverse()` method:

```
A.inverse()
```

We can check that the products of these matrices are actually the identity:

```
A * A.inverse()
```

```
A.inverse() * A
```

Exponent notation also produces the same result.

```
A^-1
```

Observe that a matrix is invertible over its base ring. Even though the next matrix is invertible, we would obtain a false result if we do not specify the ring:

```
A = matrix([
    [1, 2, 8],
    [9, 8, 9],
    [5, 6, 0]
])
A.is_invertible()
```

By calculating the inverse, we can see that the inverse exists and therefore A is invertible, though the coefficients are not integers:

```
A^-1
```

If we specify the ring to be the rationals, we obtain the correct result:

```
A = matrix(QQ, [
    [1, 2, 8],
    [9, 8, 9],
    [5, 6, 0]
])
A.is_invertible()
```

Next, we will show how to use Sage to implement the different ways in which we can calculate the inverse matrix.

9.1.1 Echelon Method

A matrix is invertible if and only if the reduced row echelon form is the identity matrix. We can find the inverse matrix of A by augmenting A by the identity matrix I and then finding the reduced row echelon form. If the matrix is invertible, the first half of the matrix would be the identity and the second half the inverse matrix.

Let's start by augmenting the matrix.

```
A = matrix([      # case i
        [0, 1, -1],
        [1, 2, 0],
        [1, 0, 1]
    ])
AI = A.augment(identity_matrix(3), subdivide=true)
AI
```

By invoking `rref()`, we obtain the reduced form of the augmented matrix, which allows us to determine whether the original matrix is invertible.

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

Since the left sub-matrix is the identity matrix, the original matrix is invertible and the right sub-matrix is the inverse. We can obtain the inverse using the `submatrix()` method:

```
B = rrefAI.submatrix(0,3)
B
```

This algorithm coincides with Sage's built-in `inverse()` method:

```
A.inverse() == B
```

9.1.2 Adjugate Method

The adjugate matrix can be used to compute the inverse of an invertible matrix using the formula: $A^{-1} = \frac{1}{\det(A)} \cdot \operatorname{adj}(A)$.

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

This algorithm also coincides with Sage's built-in `inverse()` method:

```
A.inverse() == C
```

9.1.3 Comparing Methods

The three methods coincide when the matrix is invertible.

```
I = identity_matrix(3)
A = matrix([      # case i
        [0, 1, -1],
        [1, 2, 0],
        [1, 0, 1]
    ])
A.inverse() == A.augment(I).rref().submatrix(0,3) ==
    A.adjugate() / A.det()
```

Let's see how they differ in the way they handled non-invertible matrices.
In the case of a matrix that is not invertible, the `inverse()` method will raise an exception.

```
M = matrix([   # case ii
    [1, 2, 0],
    [0, 1, -1],
    [1, 0, 2]
])
try:
    M.inverse()
except Exception as e:
    print(e)
```

Using the adjugate method will also raise an exception.

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

While the echelon method will return a matrix whose first half is not the identity.

```
M.augment(I).rref()
```

We can programmatically check for this with the following code:

```
I == M.augment(I).rref().submatrix(0,0,-1,3)
```

To generalize the previous code to any $n \times n$ matrix, keep the first three arguments the same but replace 3 with n.

Chapter 10

Vector Spaces

Linear algebra is the study of vector spaces and linear transformations. This chapter will introduce how to use Sage to work with vector spaces.

10.1 Definition of Vector Spaces

A vector space provides an abstract generalization of the vectors in $\mathbb{R}^n$ defined before. In $\mathbb{R}^n$, vectors can be added and multiplied by real numbers. Abstract vectors will have similar operations, and scalars can be drawn from any field $\mathbb{F}$, not exclusively $\mathbb{R}$.

Note 10.1.1 Fields. Fields have their own set of axioms, but for our purposes we will use pre-defined fields in Sage. So you may think of them as a class in Sage.

A vector space over a field of scalars $\mathbb{F}$ as a nonempty set V of vectors, together with the operations, $+$ vector addition and $\cdot$ scalar multiplication, such that the following axioms are satisfied for all $u, v, w \in V$ and $a, b, 1 \in \mathbb{F}$:

Closure

$$v + u \in V$$
$$a \cdot v \in V$$

Commutativity

$$v + u = u + v$$

Associativity

$$(v + u) + w = v + (u + w)$$
$$(ab) \cdot v = a \cdot (b \cdot v)$$

Additive Identity There is a vector, $\mathbf{0} \in V$ such that $\mathbf{0} + v = v$.

Additive Inverse There exists a vector, $-v \in V$ such that $v + (-v) = \mathbf{0}$.

Multiplicative Identity

$$1 \cdot v = v$$

Distributivity

$$a \cdot (u + v) = a \cdot v + a \cdot u$$
$$(a + b) \cdot v = a \cdot v + b \cdot v$$

As an example, $\mathbb{R}^n$ is a vector space over $\mathbb{R}$ and the set $\mathbb{F}^n$ of n-tuples $(a_1, \ldots, a_n)$ where $a_i \in \mathbb{F}$ forms a vector space over $\mathbb{F}$. We will say that the dimension of these vector spaces is n.

Sage allows for us to create a vector space using the `VectorSpace` command. We need to pass the field as well as the dimension of the space as arguments. For instance, the following method creates the vector space $\mathbb{R}^3$:

```
V = VectorSpace(RR, 3)
V
```

Note 10.1.2 Mantissa. By executing this command, we created a 3-dimensional vector space composed of Real numbers with 53 bits of precision. The 53 bits are referred to as the mantissa, or the significand. The mantissa is the number of significant digits that are in the floating point system. This signifies how precise the number can be.

Let's create now the vector space $\mathbb{C}^2$ of pairs of complex numbers:

```
W = VectorSpace(CC, 2)
W
```

Equivalently, we can define the same vector space as:

```
W = CC ^ 2
W
```

We can check if a vector belongs to a vector space. Like this:

```
v = vector(RR, [1,3,4])
# Ensure that the vector is over the same field
v in V
```

10.2 Subspaces

A subspace of the vector space V is a nonempty subset W of V such that W is a vector space under the same addition and scalar multiplication as on V.

For a nonempty subset W of a vector space V, to check that W is a subspace of V, we need only verify that W satisfies the closure properties:

Addition	If $u, w \in W$, then $u + w \in W$.
Scalar Multiplication	If $a \in \mathbb{F}$ and $u \in W$, then $a \cdot u \in W$.

Sage does not provide a way to check if a *subset* is a subspace. The method `is_subspace` admits as argument a vector space W and check if it is subspace of the ambient space V. In other words, it checks if W is a subset of V.

```
V = VectorSpace(RR, 3)
W = VectorSpace(RR, 2)
W.is_subspace(V)
```

We obtained that $\mathbb{R}^2$ is not a subspace of $\mathbb{R}^3$ since it is not a subset.

Next, we will explore a method for constructing a specific type of subspace within a given vector space.

10.2.1 Spans

A linear combination of a set of vectors $\{v1, \dots, v_k\}$ in a vector space V over the field $\mathbb{F}$ is a vector of the form $a_1v_1 + \cdots + a_kv_k$, where $a_1, \dots, a_k \in \mathbb{F}$.

For example, let's calculate a linear combination of the vectors u and v in $\mathbb{R}^3$:

```
u = vector([0, 1/2, 0])
v = vector([1, -2, 3])
3*u + 2*v
```

The set of all linear combinations of a set of vectors $\{v_1, \dots, v_k\}$ forms a subspace of V called the span of $\{v_1, \dots, v_k\}$, denoted $\text{span}(\{v_1, \dots, v_k\})$.

In Sage, the `span` command can be used to create the span of a set of vectors. Let's find the span of the vectors u and v in $\mathbb{R}^3$.

```
V = VectorSpace(RR, 3)

S = V.span([u, v])
S
```

We can verify that S is a subspace of $\mathbb{R}^3$:

```
S.is_subspace(V)
```

We can now check if other vectors lie within this span. For example, let's test if the vector $u = (2, 0, -2)$ is in the span S.

Observe that the subspace spanned by the set of vectors u, v and $3u + 2v$ is also S:

```
V = VectorSpace(RR, 3)
T = V.span([u, v, 3*u + 2*v])
T
```

We can now check for equality.

```
T == S
```

We see that the new vector added to the generators is redundant. Next, we will see how Sage can detect this redundancy.

10.2.2 Linear Independence

The vectors $v_1, v_2, \dots, v_k$ are **linearly dependent** if there exist scalars $c_1, c_2, \dots, c_k \in \mathbb{F}$, where at least one $c_i \neq 0$, such that

$$c_1v_1 + c_2v_2 + \cdots + c_nv_k = 0$$

Otherwise, we say these vectors are **linearly independent**.

Sage provides the `linear_dependence` method to check if a set of vectors is linearly dependent, in a given vector space, and in that case outputs the coefficients $c_1, c_2, \dots, c_k$ of the linear combination.

We can then test for linear dependence in the context of V.

```
V = VectorSpace(RR, 3)
v = vector([0, 1/2, 0])
u = vector([1, -2, 3])
w = vector([2, -5/2, 6])

L = [v, u, w]
V.linear_dependence(L)
```

We obtained a list of coefficients, indicating that the vectors u, v and w are linearly dependent and the following combination gives the vector zero:

$$1u + (2/3)v + (-1/3)w = 0$$

Observe that the coefficients in the linear combination are not unique. Sage assigns by default 1 as the first coefficient.

If the set of vectors is linearly independent, then Sage returns an empty list:

```
S = [u, v]
V.linear_dependence(S)
```

We can also check directly for linear independence:

```
V.linear_dependence(S) == []
```

Alternatively, we can manually check if a set of vectors is linearly independent by using the matrix methods studied before.

Note 10.2.1 Rank. The rank of a matrix M is the maximal number of linearly independent row vectors (rows viewed as vectors).

```
# Create a matrix from the vectors
M = matrix([u,v,w])

# Check if the rank is equal to the number of vectors
M.rank() == len(M.rows())
```

The result is `True`, so the set $\{v_1, v_2, v_3\}$ is linearly independent. Now let's consider a linearly dependent set.

```
A = matrix([u, w])
A = matrix([u1, u2, u3])
A.rank() == len(A.rows())
```

10.2.3 Basis and Dimensions

A basis for the vector space V is a set of vectors $\{v_1, v_2, \ldots, v_k\}$ that is linearly independent and spans V.

Th `basis` method outputs a basis of the vector space, in echelonized form, as a list of vectors.

```
V = VectorSpace(QQ, 3)
V.basis()
```

Sage implicitly computes the basis and dimension of any subspace generated with the `span` command.

```
V = VectorSpace(QQ, 4)
# This set is linearly dependent
v1 = vector(QQ, [1, 2,  0,  1])
v2 = vector(QQ, [0, 2, -1,  2])
v3 = vector(QQ, [1, 0,  1, -1])
v4 = vector(QQ, [2, 4,  0,  2])
S = V.span([v1,v2,v3,v4])
S
```

The method `basis_matrix` outputs a basis of the vector space, in echelonized form, as rows of a matrix.

```
S.basis_matrix()
```

The `dimension` command simply returns the number of vectors in the basis.

```
S.dimension()
```

Alternatively, we can create the same subspace using the `subspace` method:

```
W = V.subspace([v1,v2,v3,v4])
W
```

We can then verify that W is a subspace of V with the `is_subspace` method.

```
W.is_subspace(V)
```

We can check that we obtain the same object.

```
S == W
```

We can also specify a different basis than the echelonized one given by default, using the `subspace_with_basis` method:

```
u1 = vector([1, 4, -1, 3]) #another basis of the same space S
u2 = vector([2, 2, 1, 0])
T = S.subspace_with_basis([u1,u2])
T
```

This method returns error if we input a list that is not a basis for S.

Observe that we still obtain the same object, only the specified basis changes:

```
S == W == T
```

10.2.4 Extracting a Basis from a Generator

Every spanning set in a vector space can be reduced to a basis of that vector space. We can manually extract a basis from a given generating set $\{v_1, v_2, \ldots, v_n\}$ as follows:

i Construct a matrix with $v_1, v_2, \ldots, v_n$ as its columns.

ii Find the pivot columns.

The columns corresponding to the pivots form a basis of the subspace spanned by the vectors.

For example, let's extract a basis of S from the generator v_1, v_2, v_3, v_4.

```
# Re-initialize our previous definitions
V = VectorSpace(QQ, 4)
v1 = vector(QQ, [1, 2,  0,  1])
v2 = vector(QQ, [0, 2, -1,  2])
v3 = vector(QQ, [1, 0,  1, -1])
v4 = vector(QQ, [2, 4,  0,  2])

A = column_matrix([v1,v2,v3,v4])
A.pivots()
```

We obtain that the pivot columns are the first and the second. Then the vectors v_1 and v_2 forms a basis of S. Since we already know that the dimension of S is 2, this result is consistent. We can also verify that these vectors are linearly independent:

```
V.linear_dependence([v1,v2])
```

10.2.5 Coordinates in a Basis

If $B = \{b_1, b_2, \ldots, b_n\}$ is a basis of the vector space V, then every $v \in V$ can be expressed uniquely as a linear combination:

$$v = c_1 b_1 + c_2 b_2 + \cdots + c_n b_n$$

The scalars $c_1, c_2, \ldots, c_n$ are the coordinates of v with respect to the basis B and we write $[v]_B = (c_1, c_2, \ldots, c_n)$.

Sage provides dedicated methods to calculate the coordinates of a given vector in any basis. To calculate the coordinates in the canonical echelonized basis, we simply use the method `coordinates` and Sage will return the coordinates as a list. For instance, let's calculate the coordinates of v_1 in the canonical basis B of S. Recall that the canonical basis is given as a list:

```
# Re-initialize our previous definitions
v1 = vector(QQ, [1, 2,  0,  1])
v2 = vector(QQ, [0, 2, -1,  2])
v3 = vector(QQ, [1, 0,  1, -1])
v4 = vector(QQ, [2, 4,  0,  2])
S = V.span([v1,v2,v3,v4])
u1 = vector([1, 4, -1, 3])
u2 = vector([2, 2, 1, 0])

B = S.basis()
```

```
B
```

Then the coordinates of v_1 are:

```
S.coordinates(v1)
```

We obtained that the vector $v_1 = b_1 + 2 \cdot b_2$.

```
b1 = B[0] #first vector in the list B
b2 = B[1] #second vector in the list B
1*b1 + 2*b2 == v1
```

By using the method `coordinate_vector` we obtain the same coordinates as a `vector`:

```
v1B = S.coordinate_vector(v1)
v1B
```

Recall the definition of matrix multiplication, this results in $[v_1]_B \cdot M_B = v_1$, where M_B is the canonical basis matrix of S.

```
v1B * S.basis_matrix() == v1
```

10.2.6 Change of Basis

To calculate the coordinates in any other user defined basis, we use a combination of the methods `subspace_with_basis` and `coordinates`. For instance, let's calculate the coordinates of v_1 in the basis $C = \{u_1, u_2\}$ of S:

```
# Re-initialize our previous definitions
V = VectorSpace(QQ, 4)
v1 = vector(QQ, [1, 2,  0,  1])
v2 = vector(QQ, [0, 2, -1,  2])
v3 = vector(QQ, [1, 0,  1, -1])
v4 = vector(QQ, [2, 4,  0,  2])
S = V.span([v1,v2,v3,v4])
u1 = vector([1, 4, -1, 3])
u2 = vector([2, 2, 1, 0])

T = S.subspace_with_basis([u1,u2])
T.coordinates(v1)
```

We obtained that the vector $v_1 = 1/3 \cdot u_1 + 1/3 \cdot u_2$. Once again, we can verify it by manually calculating this linear combination and comparing it with the given vector:

```
1/3*u1 + 1/3*u2 == v1
```

As expected, when we compute the coordinates of the basis vectors themselves, we obtain the standard unit vector coordinates.

```
S.subspace_with_basis([v1, v2]).coordinates(v1)
```

```
S.subspace_with_basis([v1, v2]).coordinates(v2)
```

10.3 Bases

Given a basis for a vector space, any vector in that space can be expressed as a unique linear combination of the basis vectors. The coefficients of this linear combination are called the **coordinates** of the vector with respect to that basis. Changing from one basis to another is a fundamental operation, and it is accomplished using a **change-of-basis matrix**.

10.3.1 Finding Coordinates

If $B = \{b_1, b_2, \ldots, b_n\}$ is a basis for a vector space V, and $v \in V$, the coordinate vector of v with respect to B, denoted $[v]_B$, is the unique vector of scalars $(c_1, c_2, \ldots, c_n)$ such that

$$v = c_1 b_1 + c_2 b_2 + \cdots + c_n b_n$$

This equation can be rewritten in matrix form as $v = P_B\big([v]_B\big)$, where P_B is the matrix whose columns are the basis vectors from B. To find the coordinates, we solve for $[v]_B$ using, $[v]_B = P_B^{-1} v$.

Let's find the coordinates of the vector $v = (4, -3)$ with respect to the basis $B = \{(1, 1), (1, -1)\}$.

```
v = vector([4, -3])
b1 = vector([1, 1])
b2 = vector([1, -1])

# Create the matrix with basis vectors as columns
P_B = matrix([b1, b2]).transpose()

# Solve the system P_B * c = v for the coordinate vector c
v_B = P_B.solve_right(v)
v_B
```

So, $[v]_B = (1/2, 7/2)$.

10.3.2 Change of Basis

Often we need to convert the coordinates of a vector from one basis B to another basis C. This is done using a change-of-basis matrix, $P_{C\leftarrow B}$. The conversion formula is:

$$[v]_C = P_{C\leftarrow B}\big([v]_B\big)$$

The change-of-basis matrix from B to C is given by $P_{C\leftarrow B} = P_C^{-1} \circ P_B$.

Let's use the basis B from before and a new basis $C = \{(2, 1), (1, 0)\}$. We already know $[v]_B = (1/2, 7/2)$. Let's find $[v]_C$ using a change-of-basis matrix.

```
v_B = vector([0.5, 3.5])
```

```
# Define basis C and its matrix P_C
c1 = vector([2, 1])
c2 = vector([1, 0])
P_C = matrix([c1, c2]).transpose()

# Calculate the change-of-basis matrix from B to C
P_C_inv = P_C.inverse()
P_C_from_B = P_C_inv * P_B

# Calculate the new coordinates
v_C = P_C_from_B * v_B
v_C
```

Let's verify this by solving for the coordinates directly:

```
v = vector([4, -3])
P_C.solve_right(v)
```

10.4 Gram-Schmidt Process

The **Gram-Schmidt process** is an algorithm used to create an **orthonormal** set from a linearly independent set of vectors. An orthonormal set of vectors has two key properties:

- Every vector in the set is orthogonal to every other vector.
- Every vector has a length of 1.

Suppose $v_1, v_2, \ldots, v_n$, is a linearly independent set of vectors, the algorithm can be described as follows. We begin by finding the orthogonal set:

$$u_k = v_k - \sum_{j=1}^{k-1} \frac{v_k \cdot u_j}{||u_j||^2} u_j$$

We then normalize these vectors to obtain the orthonormal set:

$$e_k = \frac{u_k}{||u_k||}$$

10.4.1 Finding an Orthogonal Basis

We will first implement Gram-Schmidt manually. In the following example, we use the basis $\{v_1, v_2\}$ of a subspace of $\mathbb{R}^3$.

```
v1 = vector([1, 1, 0])
v2 = vector([1, 2, 2])
v1, v2
```

If

$$p_2 = \frac{v_2 \cdot u_1}{||u_1||^2} u_1.$$

Then we can compute the orthogonal set by:

$$u_1 = v_1, \quad u_2 = v_2 - p_2$$

```
# Step 1
u1 = v1

# Step 2: u2 = v2 - p2
p2 = v2*u1 / norm(u1)^2 * u1
u2 = v2 - p2

#orthogonal basis
u1, u2
```

The Gram-Schmidt algorithm is implemented in Sage as a method for matrices. Therefore, we need to write a matrix where the rows are the vectors of the original set. Sage returns two matrices, the vectors in the orthogonal basis are the rows of the first matrix. The second matrix contains the coefficients in the linear combinations at each step of the process. Let's start by constructing the matrix from the set of linearly independent vectors $\{v_1, v_2\}$.

```
v1 = vector([1, 1, 0])
v2 = vector([1, 2, 2])
A = matrix([v1, v2])
A
```

Now we apply `gram_schmidt()` method and consider the first matrix returned.

```
B, mu = A.gram_schmidt()
B
```

The matrix B contains the orthogonal basis as rows. Let's check that the basis $\{u_1, u_2\}$ obtained manually coincides with this one.

```
C = matrix([u1, u2])
C == B
```

10.4.2 Finding the Orthonormal Basis

For the last step of the algorithm, we can manually normalize the orthogonal basis obtained.

```
e1 = u1 / norm(u1)
e2 = u2 / norm(u2)
e1, e2
```

Alternatively, we can use the option `orthonormal=True` in the `gram_schmidt()` method.

```
v1 = vector(QQbar,[1, 1, 0])
v2 = vector(QQbar,[1, 2, 2])
A = matrix([v1, v2])
A
```

```
B, mu = A.gram_schmidt(orthonormal=True)
B
```

Let's write the vectors e_1, e_2 obtained manually as rows of a matrix M and compare it with the matrix B of orthonormal vectors obtained directly.

```
M = matrix([e1, e2])
M
```

```
B == M
```

Chapter 11

Eigenvectors and Eigenvalues

PLACEHOLDER.

11.1 Definition

An **eigenvector** of a square matrix A is a non zero vector whose direction is unchanged when multiplied by A. Formally, for a vector v, this relationship is expressed as:

$$Av = \lambda v$$

Here, λ is a scalar known as the **eigenvalue** associated with the eigenvector v.

First, we will show how to use Sage to calculate the eigenvalues of a matrix, and then we will proceed with the calculations of its eigenvectors.

11.2 Eigenvalues

In this section, we will learn how to compute eigenvalues of a matrix.

11.2.1 Manual Calculation of Eigenvalues

Since the equation $Av = \lambda v$ is equivalent to $(A - \lambda I)v = 0$, the eigenvalues of A are precisely the scalars λ for which this equation has nontrivial solutions. The determinant of the coefficient matrix of this homogeneous system must be zero. Therefore, we require $\det(A - \lambda I) = 0$.

The nth-degree polynomial $\det(A - \lambda I)$ is called the **characteristic polynomial** of A. The eigenvalues of A are the solutions of the **characteristic equation**:

$$\det(A - \lambda I) = 0.$$

Let's start by solving this equation manually, to calculate the eigenvalues. First, we enter our matrix A:

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

We declare λ as a variable.

```
var('□')
```

We then construct the matrix $B = A - \lambda I$.

```
B = A - □ * identity_matrix(3)
B
```

Next, we define the characteristic equation $\det(B) = 0$.

```
ceq = B.det() == 0
ceq
```

Finally, we solve the characteristic equation.

```
solve(ceq)
```

Sage returns the solution of the equation, which are the eigenvalues of A. By using the `lhs()` method, we obtain the left-hand side part of the equation, namely, the characteristic polynomial.

```
cpoly = ceq.lhs()
cpoly
```

Now we factor the characteristic polynomial, so that we can identify each eigenvalue and determine how many times it appears as a root. The number of times an eigenvalue occurs as a root of the characteristic polynomial is called its **algebraic multiplicity**.

```
cpoly.factor()
```

11.2.2 Sage Calculation of Eigenvalues

Sage provides the `eigenvalues()` method to compute the eigenvalues directly and return them as a list.

```
A = matrix([
    [0, 1, 1],
    [1, 0, 1],
    [1, 1, 0]
    ])
A
A.eigenvalues()
```

Observe that we obtained the same eigenvalues as before, each repeated according to its algebraic multiplicity.

To calculate the characteristic polynomial, Sage provides the `characteristic_polynomial()` method. Note that this method computes the polynomial

$$p = \det(\lambda I - A),$$

which may differ from our definition of the characteristic polynomial by a sign, depending on the dimension n of the matrix. Indeed,

$$\det(\lambda I - A) = (-1)^n \det(A - \lambda I).$$

To compute the polynomial p, we can use the `characteristic_polynomial()` or `charpoly()` methods. Using either method will yield the same result.

```
p = A.characteristic_polynomial(var='□')
p
```

We can check now that the characteristic polynomial *cpoly* we computed by hand before, coincides with this polynomial p returned by the `characteristic_polynomial()` method multiplied by $(-1)^n$.

```
bool(cpoly == (-1)^3 * p)
```

Note that if we want to test whether the symbolic expression q is equal to the symbolic expression r, we need to type `bool(q == r)`.

11.3 Eigenvectors

In this section, we will learn how to compute eigenvectors of a matrix.

Recall the definition of an eigenvector which we used to solve for λ. Using these eigenvalues we can create a matrix equation of the form:

$$(A - \lambda I)v = 0$$

We can obtain v by substituting a value of λ, we will chose $\lambda = 6$.

```
A = matrix([
    [6, 0],
    [4, 2],
    ])
B = A - 6 * identity_matrix(2)
B
```

Then using the `right_kernel()` method Sage can solve this equation for us. Since Sage uses the left-kernel while our formula uses right multiplication, we must specify to use right-kernel. Then using the `right_kernel()` method Sage can solve this equation for us. Since Sage uses the left-kernel while our formula uses right multiplication, we must specify to use right-kernel.

```
B.right_kernel()
B.right_kernel()
```

We can find all eigenvectors of A with the `eigenvectors_right()` method.

```
A.eigenvectors_right()
```

The output is a list containing elements of the form `(e, [v_1, v_2, ... v_n], m)`, where `e` is the eigenvalue, `v_1` through `v_n` are the corresponding eigenvectors, and `m` is how many times `e` appears as a root in the characteristic polynomial (also referred to as algebraic multiplicity).

Chapter 12

Diagonalization and LU Decomposition

Many problems in linear algebra become simpler when a matrix is written in a special form. Two of the most important such forms are *diagonal form* and *triangular form*. In this section, we introduce matrix diagonalization and LU decomposition.

12.1 Diagonalization of a Matrix

A square matrix A is said to be *diagonalizable* if it can be written in the form $P^{-1}AP = D$, where D is a diagonal matrix whose diagonal entries are eigenvalues of A, and P is an invertible matrix whose columns are eigenvectors of A (Note that P^{-1} exists only if the eigenvectors are linearly independent). Equivalently, this relation can be written as $A = PDP^{-1}$.

Conceptually, diagonalization represents a *change of basis*. In the basis formed by the eigenvectors of A, the linear transformation associated with A acts simply by scaling each coordinate by an eigenvalue.

Note that not every square matrix can be diagonalized. A matrix A is diagonalizable if and only if it has enough linearly independent eigenvectors to form a basis of the vector space. In particular, an $n \times n$ matrix must have n linearly independent eigenvectors.

When diagonalization is possible, it is useful for computing matrix powers, solving systems of differential equations, and understanding the structure of linear transformations. To check if a matrix is diagonalizable, Sage provides the built-in function `is_diagonalizable()`.

```
A = matrix(QQ, [
    [4, 1],
    [0, 2],
])

A.is_diagonalizable()
```

To perform diagonalization of a diagonalizable matrix A, Sage provides the matrix built-in function `diagonalization()` returning a tuple (P, D) where P is the matrix of eigenvectors and D is the diagonal matrix of eigenvalues.

```
A = matrix(QQ, [
    [4, 1],
    [0, 2],
])

D, P = A.diagonalization()

print(D)
print()
print(P)
```

Alternatively, we could leverage Sage built-in `eigenmatrix_right()` method to diagonalize a matrix:

```
D, P = A.eigenmatrix_right()
print(D)
print()
print(P)
```

We can then verify that indeed $A * P = P * D$.

```
A * P == P * D
```

Similary, we can assert that $D = P^{-1} * A * P$

```
D == P.inverse() * A * P
```

A frequent mistake is to assume that every square matrix is diagonalizable. This is not true. A matrix may have eigenvalues but still fail to have enough linearly independent eigenvectors. Such matrices are sometimes called *defective*.

For example, a matrix with a repeated eigenvalue may have an eigenspace of dimension smaller than the algebraic multiplicity of that eigenvalue. In this case, no matrix P with independent eigenvectors exists, and diagonalization is impossible.

```
# A matrix with a repeated eigenvalue
A = matrix(QQ, [
    [1, 1],
    [0, 1],
])

A.eigenvalues()
A.eigenvectors_right()

# Attempt diagonalization
A.is_diagonalizable()
```

12.2 LU Decomposition

The LU decomposition of a matrix A is a factorization of the matrix into a product of a lower triangular matrix L (often with ones on the diagonal) and an upper triangular matrix U. Mathematically, this can be expressed as $A = L \times U$. This decomposition is useful for solving systems of linear equations and computing the determinant of the matrix.

LU decomposition is closely related to Gaussian elimination. The matrix L records the multipliers used during elimination, while U is the resulting row-echelon (triangular) matrix.

Note that not every matrix admits an LU decomposition without modification. A breakdown occurs when a zero appears in a pivot position during Gaussian elimination. In such cases, row exchanges are required.

To handle this situation, we introduce *pivoting*. Pivoting consists of reordering the rows of A so that a suitable nonzero entry is used as a pivot. This leads to the decomposition $PA = LU$, where P is a permutation matrix representing the row swaps.

Pivoting improves numerical stability and ensures that an LU-type decomposition exists for a much larger class of matrices.

```
# Define a matrix that requires pivoting
A = matrix([[0, 2],
            [3, 4]])

# Compute the LU decomposition with pivoting
P, L, U = A.LU()

print(P)
print()

print(L)
print()

print(U)
print()

# Verify the decomposition
P * A == L * U
```

LU decomposition is a fundamental tool for solving linear systems, computing determinants, and performing efficient numerical computations. Unlike diagonalization, LU decomposition applies to a broader class of matrices, especially when pivoting is allowed.

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)
print()

print(L)
print()

print(U)
print()
```

And for a slightly compact format:

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

Note that every square matrix over the complex numbers can be written in an upper triangular form (for example, via the *Schur* decomposition), but this does not imply that the matrix is diagonalizable.

In numerical computations, LU decomposition without pivoting can be unstable. Even when an LU decomposition exists theoretically, small pivot values can lead to large rounding errors in floating-point arithmetic.

For this reason, practical algorithms almost always use *partial pivoting*, leading to a decomposition of the form $PA = LU$. Ignoring pivoting can result in inaccurate solutions when solving linear systems numerically.

```
# A matrix that is theoretically invertible
# but problematic without pivoting
A = matrix(RDF, [[1e-16, 1],
                 [1,     1]])

# LU decomposition with pivoting
P, L, U = A.LU()
print(P)
print()

print(L)
print()

print(U)
print()

# Verify the decomposition
P * A - L * U
```

Once again, it is important to distinguish between exact arithmetic (as used by Sage with symbolic or rational entries) and numerical computation with floating-point numbers. A decomposition that is exact in theory may behave poorly when implemented numerically without proper safeguards.

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:

[1] 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.

[2] SageMath, the Sage Mathematics Software System (Version 10.2), The Sage Developers, 2024, https://www.sagemath.org.

[3] Beezer, Robert A., et al. The PreTeXt Guide. Pretextbook.org, 2024, https://pretextbook.org/doc/guide/html/guide-toc.html.

[4] Zimmermann, Paul. Computational Mathematics with SageMath. Society For Industrial And Applied Mathematics, 2019.

Index