Linear Algebra with SageMath

Learn math with open-source software

Linear Algebra with SageMath

Learn math with open-source software

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

September 17, 2025

Website: GitHub Repository[1]

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

Preface

PLACEHOLDER

Hellen Colman
Chicago, January 2025

Acknowledgements

We would like to acknowledge the following peer-reviewers:

- [Name], [Affiliation]

We would like to acknowledge the following proof-readers:

- [Name], [Affiliation]

From the Student Authors

PLACEHOLDER
Allaoua Boughrira and Samuel Lubliner

Authors and Contributors

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

Hellen Colman
Math Department
Wright College
hcolman@ccc.edu

Samuel Lubliner
Computer Science
Wright College
sage.oer@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're new to programming or aiming to expand your mathematical toolkit. There are various ways to run Sage, including SageMathCell, CoCalc, and a local installation. The simplest way to start is by using the SageMathCell embedded in this book. We will also cover how to use CoCalc, a cloud-based platform for running Sage in a collaborative environment.

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

Join us as we explore the capabilities of Sage!

1.1 Intro to Sage

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

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

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

1.1.1 Sage as a Calculator

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

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

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

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

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

Try the following examples:

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

```
100 - 1
```

```
3*4
```

Sage supports two exponentiation operators:

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

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

Division in Sage:

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

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

Integer division and modulo:

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

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

1.1.2 Variables and Names

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

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

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

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

Python has a set of reserved keywords that cannot be used as variable names:

`False`, `None`, `True`, `and`, `as`, `assert`, `async`, `await`, `break`, `class`, `continue`, `def`, `del`, `elif`, `else`, `except`, `finally`, `for`, `from`, `global`, `if`, `import`, `in`, `is`, `lambda`, `nonlocal`, `not`, `or`, `pass`, `raise`, `return`, `try`, `while`, `with`, `yield`.

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

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

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

1.2 Data Types

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

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

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

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

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

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

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

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

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

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

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

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

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

```
l[0]
```

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

```
len(l)
```

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

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

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

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

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

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

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

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

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

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

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

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

1.3 Flow Control Structures

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

- **Assignment** statements store values in *variables*. These let us reuse results and build more complex expressions step by step. An assignment is performed using the `=` operator as discussed earlier (see Subsection 1.1.2). Note that Sage also supports compound assignment operators like `+=`, `-=`, `*=`, `/=`, and `%=` which combine assignment with basic arithmetic operations (addition, subtraction, multiplication, division and modulus).
- **Branching** uses *conditional statements* like `if`, `elif`, and `else` to execute different blocks of code based on logical tests.
- **Loops** such as `for` and `while` let us iterate over some data structures and also repeat blocks of code multiple times. This is useful when processing sequences, performing computations, or automating repetitive tasks.

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

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

1.3.1 Conditional Statements

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

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

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

1.3.2 Iteration

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

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

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

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

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

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

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

1.3.2.1 List Comprehension

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

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

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

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

1.3.3 Other Flow Control Structures

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

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

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

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

1.4 Defining Functions

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

Let's dive in and make some mistakes together!

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

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

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

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

Let's correct the variable name:

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

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

```
print(Hi)
```

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

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

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

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

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

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

Here is another example of a `NameError`:

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

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

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

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

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

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

```
len(l)
```

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

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

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

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

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

```
factorial(-5)
```

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

```
factorial(5)
```

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

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

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

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

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

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

1.8 Documentation

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

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

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

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

```
Set?
```

```
Set??
```

```
factor?
```

```
factor??
```

1.9 Miscellaneous Features

Sage is feature rich, and the following is a brief introduction of 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!")
```

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

Reading from a file:

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

1.9.2 Executing Shell Commands in Sage

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

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

```
!ls -la
```

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

Sage supports structured data formats such as CSV and JSON.

Generating a CSV file using shell command:

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

Reading a CSV file in Sage:

```
import csv

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

1.9.4 Using External Libraries in Sage

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

Using NumPy for numerical computations:

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

1.10 Run Sage in the browser

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

1. Navigate to Sage website[1].
2. Click on Sage on CoCalc[2].

[1] https://www.sagemath.org/
[2] https://cocalc.com/features/sage

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

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

Chapter 2

Vectors and Matrices

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

2.1 Working with Vectors in SageMath

A vector is a fundamental object in mathematics and computer science. In this section, you will learn how to create and work with vectors using SageMath, even if you are completely new to programming or mathematics software. Think of vectors as a way to represent points, directions, or any collection of numbers in a structured way.

What you will learn:

- How to create vectors.
- How to determine the size of a vector.
- How to calculate a vector's magnitude (length).
- How to convert a vector to a list and vice versa.
- How to build new vectors from existing ones.
- How to check what type a vector is (e.g., integer, rational).
- How to visualize vectors in 2D and 3D.

2.1.1 Creating a Vector

A vector is a list of numbers that represents a point in space or a direction with a specific length. For example, the coordinates (x, y) on a map can be represented as a 2-dimensional vector. In SageMath, creating a vector is straightforward: use the `vector()` function with a list of numbers enclosed in square brackets `[]`.

```
# Create a 3-dimensional vector
v = vector([10, 20, 30])
v
```

This creates a vector with three components: 10, 20, and 30. You can think of this as coordinates (10, 20, 30) in 3D space, or as an arrow pointing from the origin (0,0,0) to that point.

2.1.2 Finding Vector Size

The number of components in a vector is called its *dimension* or *degree*. Use the `.degree()` method to find it.

```
# Let's find out how many components our vector has
v = vector([11, 22, 33])
print("Our vector:", v)
v.degree()
```

This should return `3`, telling us we are working with a 3-dimensional vector.
Good to Know:
You can use also the standard Python len() function to get the same result

2.1.3 Calculate the Magnitude of a Vector

The magnitude (or length) of a vector is calculated using the formula: $||v|| = \sqrt{v_1^2 + v_2^2 + ... + v_n^2}$. In SageMath, this is done with the `.norm()` method.

```
# Calculate the magnitude of a vector
v = vector([11, 22, 33])
v.norm()
```

For a decimal approximation of the magnitude, you can use the `.n()` method:

```
# Get a decimal approximation of the length
v.norm().n()
```

This gives us the distance from the origin to the point represented by our vector.

2.1.4 Working with Vector Components

You can access a specific component of a vector using index notation using square brackets `[]`.

Good to Know: SageMath uses 0-based indexing, meaning the first element is at index (position) `0`. This might feel strange at first, but you will get used to it!

```
# Get the first component (at position 0)
v = vector([15, 25, 2])
v[0]
```

```
# Get the second component (at position 1)
v = vector([15, 25, 2])
v[1]
```

```
# Get the third component (at position 2)
v = vector([15, 25, 2])
v[2]
```

2.1.5 Convert a Vector to a List

You can see all components as a Python `list` by using the `.list()` method. A list is an ordered collection of numbers, such as `[1, 2, 3]`. The return value

is a Python `list`, so you can use any standard Python list methods on the returned value, such as `append()` or `sort()`.

```
v = vector([15, 25, 2])
components = v.list()
type(components)
```

2.1.6 Building new Vectors from Existing Ones

You can create new vectors by combining existing ones. For example, we can add a fourth component to our 3D vector `v` by converting it to a list, appending a value, and creating a new vector.

```
# Create a new 4D vector by appending 44
new_v = vector(v.list() + [44])
new_v
```

Now we have created a 4-dimensional vector: (11, 22, 33, 44).

2.1.7 Specifying Number Types

Vectors can be defined over different number systems (called *fields* or *rings*). SageMath uses special abbreviations for different number systems:

- `ZZ` - Integers (e.g., -2, -1, 0, 1, 2)
- `QQ` - Rational numbers (e.g., 1/2, 2/3, 5/4)
- `RR` - Real numbers (including decimals: 3.14, -0.5, 2.0)
- `CC` – Complex numbers: numbers that can have `i`(the square root of -1), e.g., 2 + 3i, -1 - i

Let's see examples of each:

```
# ZZ (integer vector)
v_int = vector(ZZ, [1, 2, 3])
v_int.base_ring()
```

```
# QQ (rational numbers)
v_rat = vector( QQ, [1/2, 2/3, 3/5])
v_rat.base_ring()
```

```
# RR (real numbers)
v_real = vector( RR, [3.14, -0.5, 2.0])
v_real.base_ring()
```

```
# CC (complex numbers)
# Sage uses i to represent the imaginary unit number.
v_cplx = vector([1.2 + i * 2.3, 3.5 - i * 5.7])
v_cplx.base_ring()
```

Do not worry if complex numbers seem confusing. You might not need them right away, but it is good to know they are available when you do!

2.1.8 Visualizing Vectors

One of the best ways to understand vectors is to see them. Geometrically, a vector can be visualized as an arrow starting from the origin and pointing to the point $P = (v_1, v_2, ..., v_n)$. In lower dimensions ($n \leq 3$), this representation is easy to imagine, but in higher dimensions it becomes harder to picture. In those cases, we focus on abstract concepts rather than geometric representation.

You can use `show()` in Sage or `.plot()` in Python to display the vector.

For example, let's create and visualize a simple 2D vector:

```
# Create and display a smaller 2D vector
vector([3, 2]).plot(color='red', thickness=2,
    figsize=[6,6]).show()
```

This code will display an arrow pointing from the origin to the point (3,2). Here is what each part does:

- `vector([3, 2])`: Creates a 2D vector with components 3 and 2.
- `.plot()`: Converts the vector into a visual plot (an arrow).
- `color='red'`: Makes the arrow red for better visibility.
- `thickness=2`: Sets the thickness of the arrow line.
- `figsize=[6,6]`: Sets the overall size of the figure.
- `.show()`: Displays the plot immediately.

You can experiment by changing the numbers inside the vector. For example, you can add a third value to create a 3D vector, or modify the existing values to change the direction and length of the arrow. Each time you click the Sage "Evaluate" button, the plot will update automatically to reflect your changes.

2.1.9 Putting it all together

Let's work through a complete example that uses several concepts we have learned:

```
# Create a vector
u = vector([1, 2, 3])
print("Vector u:", u)
print("Dimension:", u.degree())
print("Magnitude:", u.norm().n())
print("First component:", u[0])
print("All components:", u.list())

# Extend the vector
u_extended = vector(u.list() + [4])
print("Extended vector:", u_extended)
print("New dimension:", u_extended.degree())
```

2.2 Working with Matrices in SageMath

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.

In this section you will learn:

- How to create matrices of different sizes
- How to access specific elements in a matrix
- How to find matrix dimensions and properties
- How to work with matrix rows and columns
- How to create matrices with different number types

2.2.1 Creating Matrices

Creating a matrix in SageMath is straightforward. You use the `matrix()` function and provide lists of numbers for each row.

```
# Create a 3x4 matrix (3 rows, 4 columns)
M = matrix([
    [11, 13, 17, 19],
    [23, 29, 31, 37],
    [41, 43, 47, 53]
])
M
```

You can also create a matrix from vectors:

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

# Create a matrix from these vectors
M = matrix([v1, v2, v3])
M
```

Another way is to specify the number of rows, columns, and entries:

```
# Create a 2x3 matrix by specifying dimensions and entries
M = matrix(2, 3, [11, 22, 33, 44, 55, 66])
M
```

Sage can also infer the missing dimension from the number of entries:

```
# A 5x20 matrix with the first 100 integers
M = matrix(5, list(range(100)))
M
```

Entries can also be generated programmatically using list comprehensions:

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

2.2.2 Accessing Matrix Elements

Access individual entries using row and column indices (0-based indexing):

```
a_23 = M[1, 2]   # Second row, third column
a_23
```

2.2.3 Working with Rows and Columns

SageMath provides helpful methods for rows, columns, and diagonals:

```
M.rows()       # Get all rows
M.columns()    # Get all columns
M.row(1)       # Get 2nd row
M.column(2)    # Get 3rd column
M.diagonal()   # Get main diagonal
```

2.2.4 Matrix Dimensions

```
M.nrows()       # Number of rows
M.ncols()       # Number of columns
M.dimensions()  # Both dimensions
```

2.2.5 Vectors vs. Matrices

A single list of numbers creates a vector-like object, but it is NOT a vector.

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

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

Compare their types:

```
print("v␣is␣of␣type", type(v))
print("m␣is␣of␣type", type(m))
print("Are␣v␣and␣m␣identical?", v == m)
```

2.2.6 Specifying Number Types

Matrices can use different number systems. Sage can infer the field automatically or you can specify it.

```
# Integer matrix
m_int = matrix(ZZ, [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
])
```

```
m_int
```

```
m_int.base_ring()  # Check number type
```

Supported number systems:

- `ZZ` - Integers
- `QQ` - Rational numbers
- `RR` - Real numbers
- `CC` - Complex numbers

2.2.7 Creating Matrices with Patterns

You can create matrices using patterns and list comprehensions:

```
M = matrix(3, 4, [i + j for i in range(3) for j in range(4)])
M
```

2.2.8 Putting It All Together

A complete example combining several concepts:

```
# Create a 3x4 matrix
M = matrix([
    [1, 2, 3, 4],
    [5, 6, 7, 8],
    [9, 10, 11, 12]
])
print("Matrix:")
print(M)
print("\nDimensions:", M.dimensions())  # blank line before
    next output
print("\nElement␣at␣row␣1,␣column␣2:", M[1, 2])
print("\nSecond␣row:", M.row(1))
print("\nThird␣column:", M.column(2))
print("\nMain␣diagonal:", M.diagonal())

# Create a 2x3 rational matrix
M_rational = matrix(QQ, [
    [1/2, 2/3, 3/4],
    [4/5, 5/6, 6/7]
])
print("\nRational␣matrix:")
print(M_rational)
print("\nNumber␣type:", M_rational.base_ring())
```

Chapter 3

System of Equations

PLACEHOLDER.

3.1 Solving Equations

The `solve` function algebraically solves an equation, system of equations, or inequality. By default, Sage assumes the equation is equal to 0. Let's solve for x in the equation $8 + x = 0$.

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

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

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

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)
```

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 a equation is given between square brackets, indicating that the data type is a list. We can access the solution by **indexing** 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)

# Index the zero-th element of the list
# In this case the list has only one element.
# However, other equations may have multiple solutions and
    therefore multiple elements in the list.
S[0]
```

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()
```

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')
```

Now let's define a system of equations.

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

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

show(eq1)
```

```
show(eq2)
```

In this case, the first argument to `solve` is a list of equations. Next, we specify the variables to solve 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]
```

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

```
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.

```
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 solve a system of three equations with three variables.

```
var('x␣y␣z')
eq1 = x+y+z==6
eq2 = 2*x-z==-1
eq3 = x-y-z==-4
solve([eq1,eq2,eq3],x,y,z)
```

We can plot the equations in 3D-space using `implicit_plot3d`.

```
a = implicit_plot3d(eq1,(x,0,5), (y,0,5), (z,0,5))
b = implicit_plot3d(eq2,(x,0,5), (y,0,5), (z,0,5))
c = implicit_plot3d(eq3,(x,0,5), (y,0,5), (z,0,5))
a + b + c
```

Let's add some color to the plot so we can distinguish the equations.

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

The following is an example of a system with infinitely many solutions. We can express the general solution as a function of 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)
```

Chapter 4

System of Equations with Matrices

PLACEHOLDER.

4.1 Row Reduction

A matrix is in **reduced 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.

Let's use Sage to find the reduced echelon form of the following system of equations:

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

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.

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

Define the coefficient matrix.

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

print(coeff_matrix)
```

Define the constants vector.

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

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

```
a = coeff_matrix.augment(constants)
print(a)
```

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

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

```
a == A
```

Use the `rref()` method on the augmented matrix to get the reduced echelon form.

```
R = A.rref()
print(R)
```

We can also use Sage to help us with the individual steps of the row reduction process. Let's begin again with the augmented matrix.

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

Use the `add_multiple_of_row(i, j, s)` method to add s times row j to row i. The arguemnts use 0 based indexing.

Since we already have a leading 1 in the first row, the next step is to multiply the first row by -1 and add it to the third row.

```
A.add_multiple_of_row(2, 0, -1)
print(A)
```

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

```
A.add_multiple_of_row(2, 1, 2)
print(A)
```

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

```
A.add_multiple_of_row(0, 1, -2)
print(A)
```

Now that we have the system in reduced echelong form, we can refer to the variabels we can find the solution.

4.2 Systems of Equations with Matrices

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

Let's solve the following system of equations:

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

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

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

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

```
b = vector([4, 0, 4])
print(b)
```

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

```
A.solve_right(b)
```

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

$$x = 4,\ y = 0,\ z = 0$$

Chapter 5

Vectors

Vectors can be thought of as the building blocks of many mathematical and geometric ideas. In this chapter, we'll explore more vector operations.

5.1 Operations with Vectors

The key detail about a vector is that it provides both a magnitude and a direction in a given space. A vector in R^n is an ordered collection of n numbers, represented as $(v_1, v_2, ..., v_n)$, where each v_i is a number in R.

The space R^n refers to an n-dimensional space where each vector corresponds to a *point* or a *direction*. In the two-dimensional space R^2, for example, vectors are represented a pair (x, y), and in three-dimensional space R^3, vectors are represented by triples (x, y, z).

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

5.1.1 Vector Arithmetic

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])

# Adding vectors
print(v + w)
```

```
# Subtracting vectors
print(v - w)
```

A vector can be multiplied by a scalar using the `*` operator. The result is a new vector with each component multiplied by the scalar.

```
scalar = 2
print(scalar * v)
```

5.2 Operations on Vectors

Vectors do also support non-arithmetic operations, and operations that apply to the vector itself and its components. Some of these operations are shown below.

5.2.1 Normalized Vector

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

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

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

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

5.2.2 Cross and Dot Products of Vectors

Another form of vector multiplication is the *dot* product, where two vectors v and w are multiplied with each other to produce a *scalar* value. The dot product of two vectors v and w is defined as the sum of the products of their corresponding components. The dot product is commutative and yields a scalar value $v \cdot w = w \cdot v = \sum_{i=1}^{n} v_i w_i$.

```
v = vector([1, 2, 3])
w = vector([4, 5, 6])
print(v.dot_product(w))  # Output: 32
```

Note that the *dot* product in Sage is implemented in more than one way. All the following statements are equivalent and produce the same result.

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

The cross product of two vectors v and w is a vector that is perpendicular to both v and w. The cross product is defined only in 3D space and is calculated using the determinant of a matrix formed by the unit vectors i, j, k and the components of the vectors v and w. In Sage, the cross product is calculated using the `cross_product` method.

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

5.2.3 Conjugate of a Complex Vector

The complex conjugate of a vector is found by taking the conjugate of each of its components. Note the use of `i` as the the imaginary number in Sage.

```
# Defining a complex vector
v = vector([1 + 2*i, 3 - i])

# Computing the complex conjugate
v.conjugate()
```

Chapter 6

Matrices

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

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

6.1 Special Matrices

Some matrices occur frequently enough to be given special names:

- The *zero matrix* contains only zeros and acts as the additive identity.

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

- A *diagonal matrix* has zero entries everywhere outside the *main diagonal*. Note that a diagonal matrix may have some or all its diagonal entries equal to 0 (recall that the main diagonal runs from the top left to the bottom right of the matrix, while the secondary diagonal runs from the top right to the bottom left).

```
# 3x3 diagonal matrix
diagonal_matrix([2, 4, 6])
```

  Diagonal matrices are simple to multiply and often appear in eigenvalue problems which we'll see in upcoming chapters.

- An *identity matrix* is a diagonal matrix with all its diagonal elements equals to 1.

```
# 5x5 diagonal matrix
identity_matrix(5)
```

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

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

- Two other common types of matrices are the *Upper* and *Lower* triangular matrices:

  An Upper Triangular Matrix is a square matrix where all entries below the main diagonal are zero.

  ```
  # 3x3 Upper triangular matrix
  Matrix([[1, 2, 3], [0, 4, 5], [0, 0, 6]])
  ```

  A Lower Triangular Matrix is a square matrix where all entries above the main diagonal are zero.

  ```
  # 3x3 Lower triangular matrix
  Matrix([[1, 0, 0], [2, 4, 0], [3, 5, 6]])
  ```

  Sage does not have these last two as predefined command. They can however be obtained by leveraging the `LU()` method that we will see in the next section.

6.2 Operations With Matrices

Matrix arithmetic extends familiar operations like addition and multiplication to the world of matrices. We'll learn how to add, subtract, and multiply matrices, explore properties of these operations, and see how they're used to solve systems of equations, represent compositions of transformations, and work with data in structured ways.

6.2.1 Matrix Arithmetic

Matrices can be added, subtracted, and multiplied. However, unlike matrix multiplication, the addition and subtraction of matrices are performed element-wise. The identity matrix serves as the multiplicative identity for matrices, and the zero matrix serves as the additive identity.

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

# Matrix addition.
A + B
```

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

Multiplication with matrices comes in several forms. Scalar multiplication scales every entry in a matrix by a constant. Matrix-vector multiplication treats the vector as a column and combines it with the matrix using *dot* products. Matrix-matrix multiplication is only defined when the number of columns in the first matrix matches the number of rows in the second, and is performed by computing dot products of rows and columns.

```
# Scalar multiplication
A = matrix([[1, 2], [3, 4]])
3 * A
```

```
# Matrix-vector multiplication
v = vector([1, 2])
A * v
```

```
# Matrix-matrix multiplication
B = matrix([[0, 1], [1, 0]])
A * B
```

Note that to use the result of these operations, you would need to assign them to a variable.

6.3 Operations On Matrices

Matrices also support non-arithmetic, unary operations; operation that are applied to the matrix itself and its entrees. Some of these operations are shown below.

6.3.1 Determinant and Inverse of a Matrix

The determinant of a square matrix is a scalar value that provides important information about the matrix, such as whether it is invertible and how it scales space during transformations. For an $n \times n$ matrix A, the determinant is denoted as $\det(A)$ or $|A|$. If the determinant is zero, the matrix is singular and does not have an inverse. In Sage, the determinant of a matrix can be computed using the `det` method:

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

Note that the determinant of the identity matrix is 1, and the determinant of the zero matrix is 0.

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

```
# Computing the inverse of matrix A
A.inverse()
```

6.3.2 Transpose and Conjugate of a Matrix

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:

```
# Transpose of a matrix
A.transpose()
```

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

```
# Defining a complex matrix
A = matrix(CC, [[1 + 2*I, 3 - I], [-2*I, 4]])

# Computing the complex conjugate
A.conjugate()
```

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

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

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

6.3.3 LU Decomposition of a Matrix

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

In Sage, the LU decomposition can be computed using the `LU()` method:

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

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

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

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

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

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

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

```
print('-'*7+'\n',P.det())
```

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

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

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

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

And for a slightly compact format:

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

Chapter 7

Equations of Lines and Planes

This chapter introduces lines and planes in 3D space.

7.1 Lines in Vector Form

Learn how to write vector, parametric, and symmetric equations of lines.

7.2 Line Through Two Points

Learn how to find the equation of a line passing through two given points.

7.3 Planes with a Normal Vector

Learn how to write equations of planes using a point and a normal vector.

Chapter 8

Orthogonality

PLACEHOLDER.

Chapter 9

Linear Transformation From R Pow N to R Pow M

PLACEHOLDER.

Chapter 10

Eigenvectors and Eigenvalues

PLACEHOLDER.

Chapter 11

Diagonalization

PLACEHOLDER.

Chapter 12

General Vector Spaces

PLACEHOLDER.

Chapter 13

Linear Independence

PLACEHOLDER.

Chapter 14

General Linear Transformations

PLACEHOLDER.

References

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

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

Colophon

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

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

Index