Symbolic Computation with Python and SymPy

Fourth Edition

Davide Sandonà

Notices
While the author have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

In particular:

1. Some code snippets come from SymPy source code. Also, a few pictures show SymPy documentation. This material is subject to SymPy license[1].
2. A couple of examples comes from Stack Overflow[2]. This material is subject to the Creative Common Share-Alike license. The particular type of license will be specified in the examples.

Version History
First edition: November 2020
Second edition: September 2021

- Improved *Section 5.3.3 - Evaluation with lambdify()*.
- Added *Exercise - Divide Term By Term* in Chapter 7.
- Modified *Chapter 9 - Expression Manipulation Part 3*:
  - Removed the previous exercise: now this chapter only deals with trigonometry.
  - Moved old *Section 7.5* into *Section 9.1*.
  - Created a new trigonometry exercise in *Section 9.2*.
- Added *Implicit Differentiation* in Chapter 11.
- Modified *Chapter 22 - Plotting Module and Interactivity* to use the *SymPy Plotting Backends* module.

Third edition: February 2023

- Added section *4.5 Exercise - Numerator and Denominator*.
- Added paragraph to section *5.3.2 - Evaluation with evalf()*.
- Added section *5.4 - Simplification of numbers*.
- Edited *Exercise - Introduction to Pattern Matching* in Chapter 7.
- Modified introduction to *Section 8.4 - Solvers* to account for *linsolve(), nonlinsolve(), nsolve()*.
- Modified *Chapter 9 - Expression Manipulation Part 3*:
  - Added *Exercise - Behavior of solve() with numbers*.
  - Added *Exercise - Numerical solution with nsolve()*.
- Modified *Chapter 15 - The Equation class* to use a new module.
- Edited *Section 17.2 - Systems of Equations and Linear Algebra* to account for *linsolve()*.
- Updated *Section 20.1.1 - Coordinate Systems*.
- Updated examples in *Chapter 22 - Plotting Module and Interactivity*.

Fourth edition: September 2023

- Modified *Chapter 6 - Expressions*:
  - Mentioned *evaluate* context manager in *Expression Evaluation and UnevaluatedExpr class*.
  - Added section *Polynomials*.

[1] `https://docs.sympy.org/latest/aboutus.html#license`
[2] `https://stackoverflow.com`

  - Added section *Parsing*.
  - Added section *Polynomials and performance* to the *Advanced Section*.
- Modified *Chapter 7 - Expression Manipulation Part 2*:
  - Added *Exercise - n-th root*.
- Modified *Chapter 9 - Expression Manipulation Part 3*:
  - Renamed *Exercise - Missing Trigonometric Identity* to *Exercise - Trigonometric Identities*.
  - Added sub-section *The Fu simplification module* to *Exercise - Trigonometric Identities*.
  - Added *Exercise - Roots of a polynomial*.
- Modified *Chapter 12 - Integrals*:
  - Added section *Limitations*.
- Modified *Chapter 13 - Expression Manipulation - Part 4*:
  - Revisited excercises to take into account improvements of SymPy.
  - Added section *Numerical Integration*.
- Modified *Chapter 15 - The Equation class*:
  - Added *Example - Electric Circuit*.
- Modified *Chapter 16 - Differential Equations* to take into account improvements of SymPy.
- Modified *Chapter 17 - Matrices*:
  - Added explanation for DomainMatrix.
  - Improved legibility of *Advanced Section*.
- Modified *Chapter 19 - Expression Manipulation - Part 5*:
  - Removed exercise.
- Modified *Chapter 20 - Vectors*:
  - Added explanation about the differences between *sympy.vector* and *sympy.physics.vector*.
  - Added section *Vector Integration*.
- Modified *Chapter 21 - Assumptions* to take into account improvements of SymPy.
- Modified *Chapter 22 - Plotting module and interactivity* to take into account improvements of the *SymPy Plotting Backend* module.
- Added *Chapter 24 - Dynamical Systems and Simulations*.

ISBN: 9798489815208

To my family for their unwavering support.

Contents

About the Author

Davide Sandonà is an aerospace engineer and software developer. He became interested in Python a few years ago, at the dawn of the machine learning era. Since then, he happily explored different open source libraries and frameworks, determined to get the most out of them. Davide's interests range from Computer Vision to Geospatial Analytics to aerospace-related engineering topics.

To comment, ask technical questions about the book or to report any error, fill the form at https://dsandona.space/contact

Preface

Motivation for this Book

Python is a very popular, easy-to-learn general-purpose programming language with a thriving ecosystem of libraries that allows it to be used over a wide range of contexts, for example, web development, system administration, internet of things, machine learning and scientific computing in general.

When it comes to scientific computing with Python, the main libraries that we (as students, engineers, researchers and data scientists) should be aware of are:

- NumPy: add support for multi-dimensional arrays and matrices along with the necessary functionalities to create and operate on these data structures.
- SciPy: built on top of NumPy, it provides functionalities to perform optimization, integration, interpolation, linear algebra, signal processing, etc.
- Matplotlib: a plotting library for Python and NumPy.
- Pandas: built on top of the three previous libraries, it offers data structures and operations to manipulate numerical tables and data series. It is very useful for working with tabular data.
- SymPy: add support for symbolic mathematics. It aims to become a full-featured Computer Algebra System (CAS, from now on).

Python, as well as the libraries mentioned above, is free and open-source. There are also a lot of other scientific and engineering libraries developed on top of them, thus making the Python ecosystem a viable alternative to commercial applications like Matlab and Mathematica.

This book's focus is exclusively on SymPy and symbolic computations. We all probably have a good mathematical background from high school or university courses, which we use to solve problems in our field of work. In an ideal world, a CAS would be easy to learn and use, and should require only a minimum amount of programming skills. After all, a mathematical expression is just a combination of numbers and variables (also known as symbols) using operations and functions. In reality, specifically when referring to SymPy, the learning curve is probably steeper compared to the aforementioned numerical libraries. The reasons for this are manifold and will become apparent as we will learn how to use it. These are all experience-based observations that lay the foundations for this book:

- NumPy, SciPy and Pandas are specifically developed to work with numerical data types, whereas SymPy is specifically developed to work with symbolic data types. Fundamentally, numerical libraries are not able to operate on symbolic data types; similarly, SymPy is not able to operate on numerical data types. If our application needs both numerical and symbolic computation, we absolutely need to understand how to make the different libraries work together.
- SymPy is a strongly typed library: there are different types of numbers, different types of symbols, different types of operations, etc., that must work together in order to compute the final result. To be successful with SymPy, the user must understand how all the different types relate together and, to achieve that, a basic programming knowledge is required.
- While the official documentation is overall extensive, the quality varies from module to module. More so, the specific information we might be interested in may be scattered all around; we could find some bits of information in the *Modules reference*[3], in the *Tutorial*[4] and also in the *Gotchas and Pitfalls*[5]. There is indeed a chance we could miss something important.
- The examples provided in the *Tutorial* as well as in the *Modules reference* are very basic. This makes perfect sense as they are easy to follow; however, as soon as we try to apply the same concepts to our problems (most likely to be more difficult), things could get complicated really quickly.
- The way SymPy computes results may be very different from what we learned at school: the internal algorithms are designed to be efficient rather than intuitive. This could lead to unexpected results, requiring us to invest time investigating what happened. It will become apparent when dealing with trigonometry and integrals.
- The way we usually write and manipulate expressions with pen on paper might not be directly applicable to a CAS. More so, the results of a computation performed with SymPy might be in a different form from our expectations. Fortunately, SymPy is all about *expression manipulation*: we can modify the expression to obtain something that satisfies our need. In order to that, we must get acquainted with the way SymPy deals with expressions and its different manipulation functionalities.

While it is definitely possible to learn SymPy the hard way, that is by tinkering with our specific mathematical problems and exploring the documentation as we need it (thus learning small pieces of SymPy each time), this approach requires a substantial investment of time and resources that should arguably be better spent on solving actual problems rather than learning the library. By following this approach, we would undoubtedly encounter several *"I wish I knew that from the beginning!"* moments, which usually happen after spending a considerable amount of time and energy. The situation becomes even worse if we are occasional users of SymPy as we could forget important things in between sessions: this could be further amplified by the lack of understanding of how this library works.

[3] https://docs.sympy.org/latest/modules/index.html
[4] https://docs.sympy.org/latest/tutorial/index.html
[5] https://docs.sympy.org/latest/gotchas.html

Therefore, much like we first learn the alphabet before writing, much like we first learn general mathematics before applying it to our everyday problems, by recognizing that SymPy is a tool at our disposal, it is the opinion of the Author that a better approach to effectively learn this library is to understand its building blocks and how they relate together. In contrast with the previous approach, this book requires an initial investment of time that will be hugely paid back once our problems get harder. Whether we are just trying to solve an integral or we are developing a model describing a physical system, ultimately this approach allows the users to focus on their tasks rather than constantly exploring the documentation.

Who should read this book

This book is for:

- Engineers and Scientists, who needs a time-efficient learning path.
- Students are frequently introduced to *new amazing softwares* and left alone figuring out how to use them, which very often result in far-from-optimal scenarios. The most common is the one in which they *reinvent the wheel* because they were unaware that a particular feature was already implemented. Given the huge amount of available features, SymPy is arguably the hardest Python scientific-library to master. Hence, this book is also meant for students, who will be able to spend more time studying their courses rather than learning a software.

  A secondary but not less important objective is to bring students with a more diversified background closer to software development. Historically, many contributions to SymPy comes from *Google Summer of Code* projects, in which the dominant student-background has been *Computer Science*. As SymPy grows larger and larger, it is of paramount importance to have feedbacks from the users of topic-specific modules. Hopefully, not only they will be able to accurately elaborate which features need improvements, but they will become active contributors as well.
- Anyone who loves software development. SymPy is a marvelous piece of software engineering, which provides a wonderful playground to explore and understand *Object Oriented Programming*. By using this software and exploring its source code, we will learn what works great, what needs to be improved and, equally important, we will surely become better developers.

Prerequisites

To follow this book, the Reader should have a basic knowledge of Python: in particular, understanding the different data types, creating variables and functions, understanding the *if-else* construct, using *for/while* loops, be comfortable with the list comprehension syntax, etc. Knowledge of numerical computation with NumPy, SciPy and visualization with Matplotlib is assumed for later chapters.

There will also be sections in which the Reader is required to understand the basic concepts of *Object Oriented Programming*, which are explained in Appendix A.3. The Author strongly suggests to read this Appendix right after *Chapter 1*.

How to read this book

This book is not meant to replace the official documentation! On the contrary, it is meant to provide a logical, smooth, incremental and time-saving learning path in order to get the most out of this symbolic library. It has been written in a tutorial style targeting SymPy version 1.12. Even though future versions might introduce slight changes, it is very likely that the fundamental concepts will remain the same.

Obviously, each one of us have different needs and applications when it comes to symbolic computing. For example, students might be interested in checking the solution of a particular math or physics exercise, whereas engineers might be interested in building a model describing a particular system; scientists will be focused on their research domain, etc. Whatever our application is, the basic building blocks of symbolic computing with SymPy are the same:

- the different data types that will be related together in our mathematical expressions;
- the manipulation functions used to modify the expressions.

With that in mind, this book will focus on the most common aspects of mathematics, specifically understanding how SymPy deals with mathematical expressions, expression manipulation, calculus, differential equations, multi-dimensional entities, linear algebra, etc. While SymPy provides different modules targeting specific fields (for example, physics, geometry, number theory, statistics, etc.), they will not be covered here. Instead, thanks to the knowledge acquired through this book, the Reader will be able to quickly explore and get the most out of them.

The book is organized in chapters covering three layers of information:

1. Each chapter is going to focus on a specific topic, even though, due to the nature of symbolic computation, it is often difficult to draw a clean boundary between different topics. Some chapters will explain well-defined topics, others will introduce several things related to the main topic.
2. Thanks to the exercises contained in the series of chapters *"Expression Manipulation"*, the Reader will acquire the necessary skills to successfully use SymPy.
3. At the end of some chapters we will find a section named *"Advanced Topics"* which is meant to be optional but highly recommended. Since SymPy is an open-source project, we are going to explore its source code to understand the internal mechanisms. We will also visualize how the different SymPy objects are related together thanks to simplified UML class diagrams. These sections will be particularly useful to extend SymPy functionalities or to build libraries on top of it. As a matter of fact, SymPy is still under active development and functionalities are being added in each release: it may happens that the features we are looking for are not yet implemented but, by understanding the internal mechanisms, we do have a chance to built them ourselves.

Let's quickly see what this book offers.

- In Chapter 1, we will setup the working environment and get acquainted with Jupyter Notebook. We will also download the accompanying materials (notebooks and source code).

- From Chapter 2 to Chapter 9 we will explore the foundations of this library. The Author strongly encourages the Reader not to skip them: while some of them might be a little tedious, they offer a basic understanding of the library that it is often overlooked, yet essential to successfully use SymPy.
- From Chapter 10 to Chapter 14 we will explore calculus-related functionalities.
- From Chapter 16 to Chapter 20 we will explore the multi-dimensional functionalities, namely matrices, arrays and vectors.
- In Chapter 21 and Chapter 22 we will explore assumptions and plotting respectively.
- In Chapter 23 we will explore the *Printing Module*, which allows to convert any symbolic expression to a specific representation and customize what we see on the screen. We will also explore the *Code Generation module*, which allows to convert a symbolic expression to C or Fortran code, compile it and load the executable in order to maximize the performance of numerical evaluation.
- In Chapter 24 we will learn how to generate equations of motion of dynamical systems and set up numerical simulations.
- In Appendix A we will explore the main concepts related to Object Oriented Programming.
- Finally, Appendix B contains several cheat sheets, that is, tables containing the most common commands. These will be extremely useful to beginners and occasional users.

While most chapters are meant to be read in succession, Chapter 15, Chapter 21 and Chapter 22 can be tackled right after the first 9 chapters. Without further ado, let's get started!

Chapter 1

Setting up the environment

We start by reading the installation guidelines[1] of SymPy. Here, we check which Python version is officially supported: at the time of writing this book, Python 3.8, 3.9, 3.10, 3.11 and PyPy.

For beginners, the easiest way to start with Python and scientific computing is by downloading and installing Anaconda[2]: this application is a collection of tools and libraries (also known as packages) that are going to be installed in a single step, thus avoiding the pain of installing each package separately. We just need to download the correct version for our operating system: select the newest Python version compatible with SymPy, *Python 3.11* at the time of writing this book. Then, follow the installation instructions[3]. What is Anaconda going to install into our system?

- Python 3.11;
- *conda* package manager[4], which quickly runs and updates packages and their dependencies, and easily creates, saves, loads and switches between environments on your local computer;
- a lot of packages, including Jupyter Notebook, SymPy, NumPy, Matplotlib, which we are going to use throughout the book.

Once the installation is complete, don't forget to verify that everything works fine[5]. In particular, when viewing the output of the command `conda list`, make sure that SymPy is at the latest version, 1.12 at the time of writing this book. Alternatively, if the Reader has previous experiences with Python, it is also possible to install the aforementioned packages with *pip3*, the standard package manager for Python.

[1] https://docs.sympy.org/latest/install.html
[2] https://www.anaconda.com/products/individual
[3] https://docs.anaconda.com/anaconda/install/
[4] https://docs.conda.io/en/latest/
[5] https://docs.anaconda.com/anaconda/install/verify-install/

1.1 Jupyter Notebook

There are several technologies that allow us to write and execute Python code for computing purposes:

- Python shell: accessible from a terminal (or a *Command Prompt* in Windows) with the command `python` or `python3`. It provides basic functionalities that are quite limited for computing;
- Any modern source code editor with an integrated terminal (for example, Visual Studio Code, Atom, Sublime, etc.): after installing the necessary extensions to easily work with Python, these editors are great to write Python library code. However, they usually don't support interactive computing. Also, there are better alternatives to work with SymPy;
- IPython shell: accessible from a terminal with the command `ipython`. It provides interactive computing capabilities, significantly improving the experience in relation to the basic Python shell;
- Jupyter Notebook[6]: a web-based interactive computational environment to create Jupyter notebook documents. With them we can write texts, insert images, write and execute code and visualize the results of the computation all in a single document. We can save the notebook and reopen it later to continue our work. It is a great tool to tinker with computing problems! Differently from a source code editor where we would save the code into a file, then execute the code and eventually save the outputs into different files (for example, creating pictures from plots), with Jupyter Notebook we usually work on a single file, thus simplifying the workflow. Also, thanks to the Latex code generated by SymPy, the notebooks are capable of rendering the mathematical expressions, thus substantially improving the visualization experience.

Hence, we are going to use Jupyter Notebook, which also requires a modern web browser, like Mozilla Firefox or Google Chrome. To start this application we open a terminal (or command prompt) and run the command:

```
jupyter notebook
```

If Jupyter is not present but we previously installed Anaconda, run the following command:

```
conda install -c conda-forge notebook
```

If Jupyter is not present and we didn't install Anaconda, run the following command:

```
pip3 install notebook
```

The command `jupyter notebook` will launch a server process running on the local machine. A web page will open in our browser having the following address:

`http://localhost:8888/tree`

[6] `https://jupyter.org`

Here, *localhost* indicates that the server is running on the local machine, *8888* is the port number used by the notebook to communicate with the server, *tree* indicates the root directory from where we launched the application. The web page will be similar to Figure 1.1: it is called Jupyter *dashboard*.

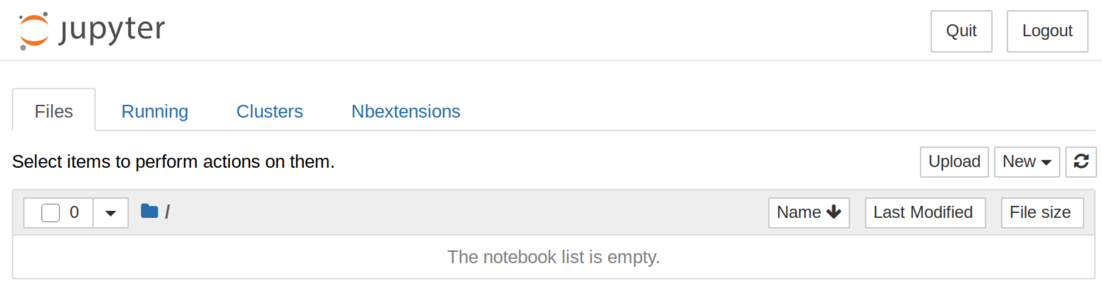

Figure 1.1: Jupyter Dashboard

In Figure 1.1 we can see four tabs:

- *Files*: it shows a list of files and folders contained in our base folder. The figure shows an empty directory. Eventually, we can click over a file to open it and explore its content.
- *Running*: This will display a list of running notebooks.
- *Clusters*: We are not interested in this tab.
- *Nbextensions*: most likely, the Reader won't have this tab enabled, yet. It provides a set of useful extensions that are going to improve the user experience.

Before diving further into the notebooks, we need to install and setup a few modules. Let's close Jupyter server process by clicking the *"Quit"* button on the top-right corner.

1.1.1 Downloading the accompanying material

This book's accompanying material contains:

- the Jupyter notebooks associated to each chapter or section of the book;
- the file `sympy_utils.py` containing the source code of the custom classes and functions that we will implement.

To download this material, let's open the following link:
https://github.com/Davide-sd/sympy-book
Then click a green button named *Code*, and then *Download ZIP*. Finally, let's extract the content to a folder of our choosing, for example *Document*.
Alternatively, if the utility `git` is installed in our system, we can open a terminal window, move into any folder of our choosing and run the following command:

```
git clone https://github.com/Davide-sd/sympy-book.git
```

1.1.2 Installing the SymPy Plotting Backends module

As of SymPy version 1.12, plotting capabilities are rather limited. However, the Author implemented a more advanced and feature-rich plotting module, which will be used throughout the book and, in particular, in Chapter 22. Among the features, we can:

- create plots with Matplotlib, Bokeh, Plotly and k3D-Jupyter.
- easily plot lines, surfaces, vector fields, complex numbers, complex functions, geometric entities and more.
- get a better understanding of our symbolic expressions thanks to parametric interactive plots with widgets.

All of this comes with a price: many dependencies will be downloaded, for an approximate download size of about 125MB. Let's list the most important ones:

- `ipympl`[7]: it enables partial interactivity with Matplotlib.
- `panel`[8]: let us create interactive applications with widgets and plots.
- `bokeh`[9], `plotly`[10], `k3d`[11]: three interactive plotting libraries.
- `vtk`[12]: used to compute 3D streamlines, which is a great functionality to better understand 3D vector fields.

Detailed information about the installation process are given in the official documentation[13].

If Jupyter was installed with Anaconda, run the following commands:

```
conda install -c conda-forge sympy_plot_backends
conda install -c anaconda scipy notebook
conda install -c conda-forge adaptive
conda install -c conda-forge panel
conda install -c anaconda ipywidgets
conda install -c conda-forge ipympl
conda install -c bokeh ipywidgets_bokeh
conda install -c conda-forge colorcet
conda install -c conda-forge k3d msgpack-python
conda install -c plotly plotly
conda install -c conda-forge vtk
```

If Jupyter was installed with pip3, run the following command:

```
pip install sympy_plot_backends[all]
```

[7] https://github.com/matplotlib/ipympl
[8] https://panel.holoviz.org/index.html
[9] https://bokeh.org/
[10] https://plotly.com/python/
[11] https://github.com/K3D-tools/K3D-jupyter/
[12] https://vtk.org/
[13] https://sympy-plot-backends.readthedocs.io/en/latest/install.html

1.1.3 Installing the Algebra with SymPy module

Algebra with SymPy[14] is a module that implements an `Equation` class supporting mathematical operations. In particular, it is possible to add, subtract, divide or multiply two equations side by side. We will use it in Chapter 15.

If Jupyter was installed with Anaconda, run the following command:

```
conda install -c anaconda pip
pip install Algebra-with-SymPy
```

If Jupyter was installed with pip3, run the following command:

```
pip install Algebra-with-SymPy
```

1.1.4 Installing the PyDy module

PyDy[15] stands for Python Dynamics. It will be used in Chapter 24 to generate numerical functions of symbolic equations of motion, and to visualize dynamical systems.

If Jupyter was installed with Anaconda, run the following command:

```
conda install -c conda-forge pydy
```

If Jupyter was installed with pip3, run the following command:

```
pip install pydy
```

1.1.5 Setting up Nbextensions

Let's enable the tab *Nbextensions* seen before in Figure 1.1. We need to install the package `jupyter_contrib_nbextensions`[16] which contains the additional extensions. So, let's move to the terminal.

If Jupyter was installed with Anaconda, run the following command:

```
conda install -c conda-forge jupyter_contrib_nbextensions
```

If Jupyter was installed with pip3, run the following command:

```
pip3 install jupyter_contrib_nbextensions
```

Then, whether we used Anaconda or pip, to make the extensions available to Jupyter run the following command:

```
jupyter contrib nbextension install --user
```

Finally, let's start Jupyter again with the command `jupyter notebook` and let's click on the tab *Nbextensions*. We should see a list of extension, something like Figure 1.2.

[14] https://github.com/gutow/Algebra_with_Sympy
[15] https://pydy.readthedocs.io/en/stable/index.html
[16] https://jupyter-contrib-nbextensions.readthedocs.io/en/latest/install.html

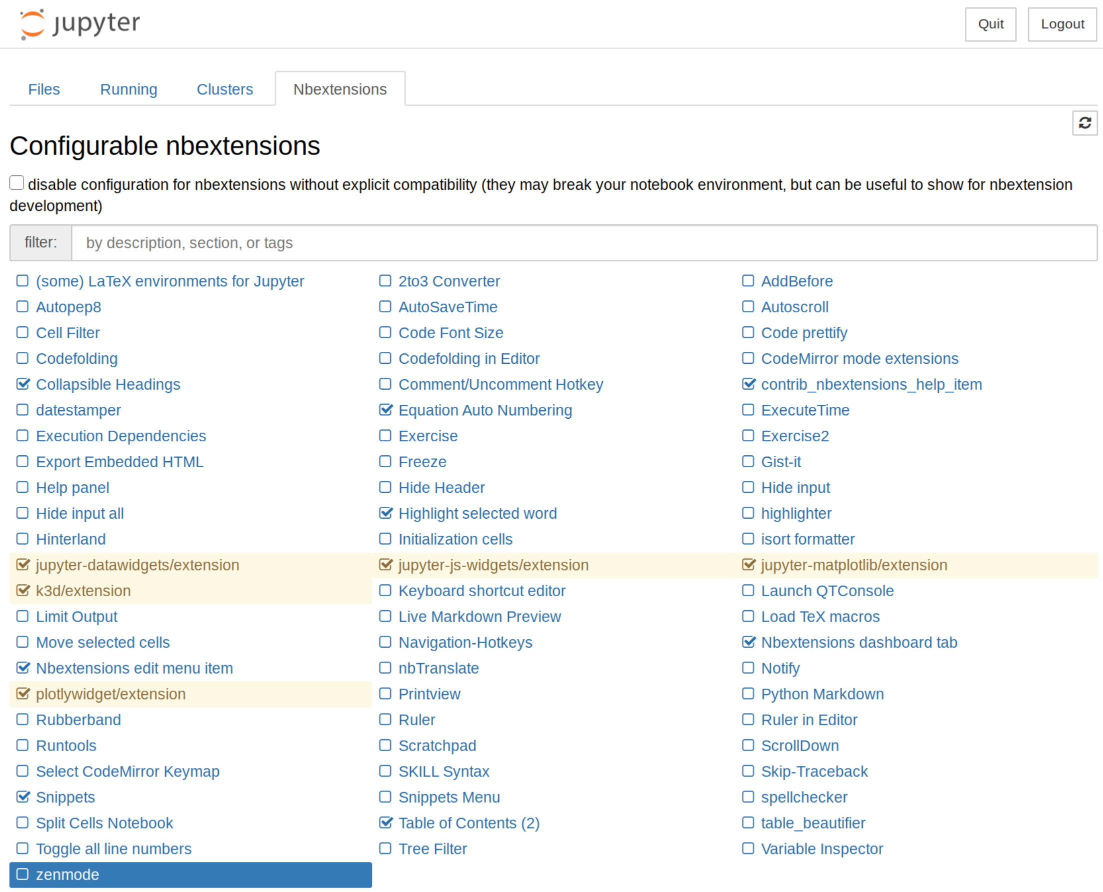

Figure 1.2: *Nbextensions* Settings page

As we can see there are a lot of extensions; two of them are highly recommended:

- *Highlight selected word*: this makes it easier to spot the selected word when we write a lot of code.
- *Snippets*: this allows to easily insert snippets of code (that is, code that is used frequently) with a mouse click. This is perfect to insert import statements.

Let's set up the Snippets extension: by clicking over Snippets, a configuration section should appear. Here, the extension is telling us to edit the following file (run the following command on the terminal):

```
$(jupyter --data-dir)/nbextensions/snippets/snippets.json
```

Note for Windows users: to find the path of that file, just run the command "`jupyter --data-dir`", then add the remaining part of the aforementioned path to the result printed on the screen. So, let's open that file with a text editor and replace the content with the following:

```
{
  "snippets" : [
    {
      "name" : "sympy-module",
      "code" : [
        "%matplotlib widget",
        "import sympy as sp",
        "from sympy.interactive import printing",
        "printing.init_printing(use_latex=True)"
      ]
    },
    {
      "name" : "sympy-all",
      "code" : [
        "%matplotlib widget",
        "from sympy import *",
        "init_printing(use_latex=True)"
      ]
    }
  ]
}
```

These two snippets are almost identical. The reader will use one of them at the beginning of each new notebook for the entirety of this book. Let's break them down:

- `%matplotlib widget`: this is a magic line used to tell Jupyter to wrap Matplotlib plots with interactive controls (zoom, pan, etc.) provided by the `ipympl` module (previously installed).
- `import sympy as sp` or `from sympy import *`: import the module or the entire content of the module. If we are using a notebook to perform symbolic computations only, we can use the snippet *"sympy-all"*. On the other hand, if we also need to perform numerical computations with other libraries, for example *NumPy*, it is better to keep things separated and use the snippet *"sympy-module"*. This is because SymPy and NumPy define their respective versions of mathematical functions, for example `sin`, `cos`, `tan`, etc., which cannot be mixed together. Refer to Appendix A.2 to understand *namespaces and scopes*.
- `from sympy.interactive import printing` and `printing.init_printing(use_latex=True)`: these are going to force SymPy to output Latex code that will be nicely rendered on the screen, thus providing an excellent user experience.

Once the file has been saved, quit Jupyter by clicking the dedicated button and then restart it.

1.1.6 Installing the antlr4 module

The *antlr4* module is a powerful parser which allows SymPy to easily convert Latex code to symbolic expressions.

If Jupyter was installed with Anaconda, run the following command:

```
conda install -c conda-forge antlr-python-runtime
```

Chapter 4

Expression Manipulation - Part 1

Now that we understand how to create symbols and functions, it is the perfect time to have some fun with *expression manipulation*. Obviously, we don't know yet all the details that allow us to be successful with this topic. More so, we will see that there is quite a gap between being able to manipulate expressions on paper and doing the same with SymPy.

Expression manipulation is a pillar of SymPy: whatever our computation is, the results are likely not to be in the form that we were hoping for. For example, integrals can compute very long results involving many operations and little to none collection of terms. Whether we are just interested in obtaining a result worth to be inserted into a document, or reducing the number of operations of an expression so that it can be efficiently evaluated, this is a topic that we have to master.

This chapter will introduce a considerable amount of new information; hence the Reader might feel overwhelmed. Do not worry: expression manipulation is a topic that requires patience, time and understanding of the different processes. We will explore it over several other chapters in this series titled *"Expression Manipulation - Part X"*; this is just the beginning. By the end of this series, we will be as good at SymPy's expression manipulation as we were doing it on paper, maybe even better.

Without further ado, let's get started.

4.1 Cheat Sheets are really useful

A great starting point to understand the complexities related to expression manipulation is the section *Simplify* of the online tutorial[1]. It is strongly recommended for the Reader to explore it, since we are not going to repeat a lot of topics covered there. For example, dealing with power simplification, logarithm simplification, understanding that assumptions have an active role on the kind of manipulation we can expect to perform.

However, somewhere in the middle of that tutorial, the Reader might get worried. *Are we supposed to remember all of those functions? Do we need to flick through the documentation every*

[1] `https://docs.sympy.org/latest/tutorial/index.html`

time we forget something? What's the relationship between the different functions? Are they working together somehow?

Considering that SymPy is only one of the tools at our disposal, we are likely going to spend a relatively short amount of time with it. Therefore, it is unrealistic to think that we will be able to remember everything. From a time-efficiency point of view, it is also unrealistic to go through the huge documentation every time we forget something.

A possibly better approach is to create a *cheat sheet*, that is, a very useful handy table filled with the most common operations, something like Table 4.1. This is part of a larger SymPy cheat sheet that we can find in Appendix B. However, a cheat sheet is no useful if we don't understand what's in there, so let's break it down.

SIMPLIFICATION
simplify(expr, rational=False, inverse=False, doit=True)
radsimp(expr, symbolic=True, max_term=4)
ratsimp(expr)
trigsimp(expr, method="matching | groebner | combined | fu")
combsimp(expr)
powsimp(expr, deep=False, combine="all | base | expr", force=False)
powdenest(expr, force=False, polar=False)
nsimplify(expr, constants=(), tolerance=None, full=False, rational=None, rational_conversion="base10 | exact")
factor(expr, deep=True, fraction=True)
together(expr, deep=False, fraction=True)
cancel(f, *gens, **args)
logcombine(expr, force=False)

EXPANSION
expand(expr, e, deep=True, modulus=None, power_base=True, power_exp=True, mul=True, log=True, multinomial=True, basic=True, complex=False, func=False, trig=False)
expand_mul(expr, deep=True)
expand_log(expr, deep=True, force=False, factor=False)
expand_func(expr, deep=True)
expand_trig(expr, deep=True)
expand_complex(expr, deep=True)
expand_multinomial(expr, deep=True)
expand_power_exp(expr, deep=True)
expand_power_base(expr, deep=True, force=False)

SUBSTITUTION
expr.**subs**(old, new, simultaneous=False)
expr.**xreplace**({k_old: v_new})
expr.**replace**(query, value, map=False, simultaneous=True, exact=None)

COLLECTION
collect(expr, syms, func=None, evaluate=None, exact=False, distribute_order_term=True)
rcollect(expr, evaluate=None)
collect_sqrt(expr, evaluate=True)
collect_const(expr, *vars, Numbers=True)
logcombine(expr, force=False)

SEARCH / FIND
expr.**find**(query, group=False)
expr.**has**(*patterns)
expr.**match**(pattern, old=False)

INFORMATION
expr.**args**
expr.**atoms**(*types)
expr.**free_symbols**
expr.**func**

OTHERS
fraction(expr, exact=False)
rewrite(*args, **hints)
sympify(obj, *args)

Table 4.1: Cheat Sheet - Expression Manipulation

When it comes to manipulation, on top of the standard arithmetic operations we can also perform expansion, collection, substitution and simplification. For each of these topics, SymPy provides a general function as well as many other specialized functions. Note: we are not going to describe each function, otherwise this book would mirror the documentation, which is left for the Reader to explore. Speaking of exploring, in order to read the documentation of a particular function we can run the command: `help(name_of_the_function)`.

Expansion. The general function is `expand()`[2]. Many keyword arguments are available to control the expansion process: to expand a given feature we just set the respective `keyword=True`. The cheat sheet shows the default value of the different options: by default `expand()` is going to do a lot of stuff. However, also by default some options are turned off: for example, expanding trigonometric functions. Should we need to expand only a particular feature, we

[2] https://docs.sympy.org/latest/modules/core.html#sympy.core.function.expand

better use one of the specialized functions: these are wrapper functions that are going to call `expand()` with only the keyword argument of the interested feature set to `True`.

Collection. The general function is `collect()`[3] that provides several options to control the process. This function is used to collect symbols and general expressions, but not numbers or combining logarithms; for that, we have to use the specialized functions. By exploring the source code, it turns out that `rcollect()` is calling `collect()`, whereas `collect_sqrt()` is calling `collect_const()`.

Substitution. The general method is `subs()`[4], whereas `xreplace()` and `replace()` allow for different level of controls, as we will understand in the exercises.

Simplification. The general function is `simplify()`[5] that provides several options to control the process. This function applies several simplification techniques and it is usually good for interactive sessions. However, if we are designing functions or classes that requires simplification steps, we better use the specialized functions, because the actual implementation of `simplify()` may change over time, whereas the specialized functions are more likely to remain the same. As with collection, under *Simplification* we find several functions that are not called by `simplify()`, for example `factor()`, `ratsimp()`, `powdenest()`, `cancel()`, `nsimplify()`, but they perform useful simplification steps.

Search/Find, Information, Others. We will explore them in details during this series of chapters. For the moment we remember the property `arg` exposed by every SymPy object: it returns the arguments (or terms) of any expression. The property `free_symbol` returns a set of symbols that makes up the expression.

From Table 4.1 it is also evident that some methods and properties must be called directly from the actual expression, for example `expr.subs()`, `expr.xreplace()`, `expr.args`. However, `simplify()`, `expand()`, `factor()` and `collect()` are so useful that we can also call them directly from the expression, for example `expr.simplify(**kwargs)` or `expr.expand(**kwargs)`. In doing so, we can easily chain together multiple simplification steps, for example `expr.expand().collect().simplify()`, allowing for a better interactive experience.

Also, in Table 4.1 we can spot some keyword arguments that require a string, for example `combine="all|base|expr"`. In these cases, the different options are separated by the pipe character, `|`, with the first option being the default.

Finally, among all the keyword arguments, we can spot `deep` quite frequently. To understand it, let's consider this expression which is composed of nested expressions:

```
x, y = symbols("x, y")
expr = (x * (x + x * (sin(x) + 2)))
expr
```

[3] https://docs.sympy.org/latest/modules/simplify/simplify.html#sympy.simplify.radsimp.collect
[4] https://docs.sympy.org/latest/modules/core.html?highlight=subs#sympy.core.basic.Basic.subs
[5] https://docs.sympy.org/latest/modules/simplify/simplify.html#sympy.simplify.simplify.simplify

Solution. Let's explore the hand written solution first:

$$\begin{aligned}\alpha\beta+\alpha+\beta+1 &= \alpha\left(\beta+1\right)+\beta+1 \\ &= \left(\beta+1\right)\left(\alpha+1\right)\end{aligned} \tag{4.14}$$

Now, let's try to do the same with SymPy:

```
alpha, beta = symbols("alpha, beta")
expr = alpha * beta + alpha + beta + 1
expr
```

$$\alpha\beta+\alpha+\beta+1 \tag{4.15}$$

We start by collecting alpha:

```
r = expr.collect(alpha)
r
```

$$\alpha\left(\beta+1\right)+\beta+1 \tag{4.16}$$

and then we collect `beta + 1`:

```
r = r.collect(beta + 1)
r
```

$$\left(\alpha+1\right)\left(\beta+1\right) \tag{4.17}$$

This was pretty easy! After an expression manipulation, it is always a good idea to test if the result is mathematically equivalent to the original expression:

```
r.equals(expr)
```

True

It is interesting to note that prior to SymPy version 1.6.0, this exercise required a more convoluted procedure to be solved, as it was impossible to collect additive terms like `beta + 1`. This shows that improvements to SymPy are made in each release.

4.4 Exercise - Substitution and Solve

In the following exercise the Reader should not focus on the physics of the problem, but on the expression manipulation steps. Here we are going to compare a "handwritten" solution to a fully SymPy-based one. We will intentionally take the long route to solve it: in doing so, we will face the most common beginner mistakes but, at the same time, we will also gain a tremendous amount of experience that will help us in our every-day problems. Let's start!

Consider a pressure vessel constituted by a cylindrical body with spherical end caps, as pictured in Figure 4.1. The volume is given by:

$$V=\pi\frac{D^2}{4}L+\frac{4}{3}\pi\frac{D^3}{8} \tag{4.18}$$

where L is the length of the cylindrical portion, D is the diameter. The pressure vessel has a thin skin of a given material; its mass is given by:

$$M = \frac{\pi D^2 \rho(T) p}{2\sigma_y(T)} \left(L + \frac{D}{2} \right) \tag{4.19}$$

where p is the internal pressure, $\rho(T)$ is the temperature dependent density of the material of the vessel, $\sigma_y(T)$ is the temperature dependent stress level on the thin walls.

Derive an expression that for a given pressure vessel volume V yields the L/D ratio which minimizes the pressure vessel mass.

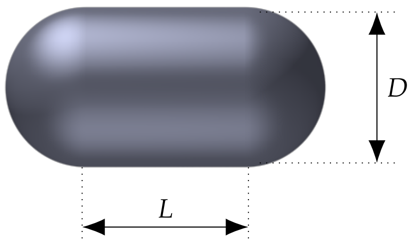

Figure 4.1: Pressure vessel dimensions

4.4.1 "Handwritten" Solution

The exercise is asking us to rewrite the mass of the pressure vessel in terms of the volume V and the ratio L/D, so that we can later do a minimization analysis. We are dealing with real physical quantities: they are all greater or equal than zero. To compute a solution, the following assumptions are used:

- We know the value of V.
- We don't know the values of D and L.
- We assume the range of values for L/D over which we will perform the minimization, for example $L/D \in [0, 10]$. The number 0 is motivated by the fact that it can very well be $L = 0$ (resulting in a spherical tank); the number 10 is arbitrary, we can go as high as we like.
- We know the value of the internal pressure, p.
- We assume the value of the temperature to be fixed. Therefore, we will treat ρ and σ_y as constant symbols, not as functions.

Let's start by rewriting the volume and mass equations as functions of the parameter L/D:

$$\begin{aligned} V &= \pi \frac{D^2}{4} L + \frac{4}{3} \pi \frac{D^3}{8} \\ &= \pi \frac{D^2}{4} L + \frac{2}{3} \pi \frac{D^3}{4} \\ &= \pi \frac{D^2}{4} \left(L + \frac{2}{3} D \right) \\ &= \pi \frac{D^3}{4} \left(\frac{L}{D} + \frac{2}{3} \right) \end{aligned} \tag{4.20}$$

$$\begin{aligned} M &= \frac{\pi D^2 \rho p}{2 \sigma_y} \left(L + \frac{D}{2} \right) \\ &= \frac{\pi D^3 \rho p}{2 \sigma_y} \left(\frac{L}{D} + \frac{1}{2} \right) \\ &= \frac{\pi D^3 \rho p}{2 \sigma_y} \left(\frac{L}{D} + \frac{1}{2} \right) \end{aligned} \tag{4.21}$$

Since we don't know the value of D, we manipulate Equation (4.20) so that:

$$D^3 = \frac{4V}{\pi \left(\frac{L}{D} + \frac{2}{3} \right)} \tag{4.22}$$

Finally, we insert Equation (4.22) into (4.21):

$$M = \frac{\rho p}{\sigma_y} \left(\frac{L}{D} + \frac{1}{2} \right) \frac{2V}{\left(\frac{L}{D} + \frac{2}{3} \right)} \tag{4.23}$$

As for the minimization, we will do it later with SymPy!

4.4.2 Basic Expression Manipulation

Let's start by defining the necessary symbols with the appropriate assumptions and writing the expressions (4.18) and (4.19):

```
D, L = symbols("D, L", real=True, positive=True)
rho, p, sigma = symbols("rho, p, sigma", real=True, positive=True)
Vs, Ms = symbols("V, M", real=True, positive=True)
V = pi * (D / 2)**2 * L + Rational(4, 3) * pi * (D / 2)**3
M = pi * D**2 * rho * p / (2 * sigma) * (L + D / 2)
display(V, M)
```

$$\begin{gathered} \frac{\pi D^3}{6} + \frac{\pi D^2 L}{4} \\ \frac{\pi D^2 p \rho \left(\frac{D}{2} + L \right)}{2 \sigma} \end{gathered} \tag{4.24}$$

There are a few things to note:

- Symbols `Vs` and `Ms` represent volume and mass, whereas variables `V` and `M` represent the expressions of volume and mass respectively!
- With `Rational(4, 3)` we have created a rational number: we will learn all about numbers in the next chapter.
- In the expression `V`, the rational number $4/3$ was then multiplied with $1/2^3$, thus producing $1/6$.

Before starting with the manipulation, it is a good idea to create a copy of the original expressions; we will use them to verify that the manipulated expression is mathematically equivalent to the initial copy:

```
Vcopy = V
Mcopy = M
```

To fully use the interactive environment, we will chain different commands together. Also, we will override an expression only when we will be happy with the result.

Starting from Equation (4.18), our goal is to reach something similar to Equation (4.20). Much as we did by hand, we can start by factoring the expression. The `factor()`[7] method takes a polynomial and factors it into irreducible factors over the rational numbers:

```
V.factor()
```

$$\frac{\pi D^2 (2D + 3L)}{12} \tag{4.25}$$

Then, we divide by D^3 so we get D in the denominator. We will multiply it back at the end:

```
V.factor() / D**3
```

$$\frac{\pi (2D + 3L)}{12D} \tag{4.26}$$

To apply the division to both terms in the numerator we use `expand()`[8]:

```
(V.factor() / D**3).expand()
```

$$\frac{\pi}{6} + \frac{\pi L}{4D} \tag{4.27}$$

The `collect()`[9] function is used to collect additive terms of an expression:

```
(V.factor() / D**3).expand().collect(pi)
```

[7] https://docs.sympy.org/latest/modules/polys/reference.html#sympy.polys.polytools.factor
[8] https://docs.sympy.org/latest/modules/core.html#sympy.core.function.expand
[9] https://docs.sympy.org/latest/modules/simplify/simplify.html#sympy.simplify.radsimp.collect

$$\pi \left(\frac{1}{6} + \frac{L}{4D}\right) \tag{4.28}$$

In the previous step, both terms contained the constant π. Earlier we said that `collect()` is not able to collect numbers, so what is going on? Turns out that `pi` is not really a number, as we will learn in the next chapter! Alternatively, we could have used the `collect_const(V, pi)` function.

Finally, we need to remember to multiply by D^3:

```
V = (V.factor() / D**3).expand().collect(pi) * D**3
V
```

$$\pi D^3 \left(\frac{1}{6} + \frac{L}{4D}\right) \tag{4.29}$$

This result is good enough for our purposes. Note that we have overridden the initial expression.

At this point, it is really a good idea to perform a sanity check:

```
Vcopy.equals(V)
```

True

This means that we have correctly manipulated the expression, since it is mathematically identical to the one we started with.

It is important to realize that expression manipulation is not an exact science; we can reach any goal following different approaches. For example, we could have achieved the same result with the commands:

```
(V / D**3).simplify().collect(pi) * D**3
```

Note the lack of `factor()`! Also, in this case `simplify()` only applied an expansion.

Let's now try to do the same with the mass expression from Equation (4.19). We would like to obtain something similar to Equation (4.21). We start by dividing by D^3 so we get D in the denominator. We will multiply it back at the end:

```
M / D**3
```

$$\frac{\pi p \rho \left(\frac{D}{2} + L\right)}{2D\sigma} \tag{4.30}$$

To apply the division to both terms in the numerator we use `expand()`[10]:

```
(M / D**3).expand()
```

$$\frac{\pi p \rho}{4\sigma} + \frac{\pi L p \rho}{2D\sigma} \tag{4.31}$$

The `collect()`[11] function is used to collect additive terms of an expression. In this case we are going to collect a group of symbols:

[10] https://docs.sympy.org/latest/modules/core.html#sympy.core.function.expand

[11] https://docs.sympy.org/latest/modules/simplify/simplify.html#sympy.simplify.radsimp.collect

```
(M / D**3).expand().collect(pi * rho * p / sigma)
```

$$\frac{\pi p\rho\left(\frac{1}{4}+\frac{L}{2D}\right)}{\sigma} \tag{4.32}$$

Finally, we need to remember to multiply by D^3:

```
M = (M / D**3).expand().collect(pi * rho * p / sigma) * D**3
```

$$\frac{\pi D^3 p\rho\left(\frac{1}{4}+\frac{L}{2D}\right)}{\sigma} \tag{4.33}$$

This result is good enough for our purposes. Again, a sanity check:

```
Mcopy.equals(M)
```

```
True
```

4.4.3 Substitution and Solve

We are now going to manipulate Expression (4.29) to obtain Expression (4.22). Let's start by writing the volume equation in the form of *"(left-hand side) - (right-hand side)"*:

```
Veq = Vs - V
Veq
```

$$-\pi D^3\left(\frac{1}{6}+\frac{L}{4D}\right)+V \tag{4.34}$$

We would like to solve this algebraic equation for D^3. SymPy exposes the `solve()`[12] function to solve different types of equations (we will explore it in Section 8.4). The first parameter must be the equation, the second is the symbol we would like to solve the equation for.

We would be tempted to do something like this:

```
solve(Veq, D**3)
```

It will probably take a few seconds (up to a few minutes, depending on the machine) to complete the computation, eventually giving us a list of three possible results, each one looking terribly long. We are not even going to copy the results since it will probably cover an entire page! If the computation appears to take a long time, stop the process by clicking the *"Interrupt the kernel"* button on the top toolbar of Jupyter Notebook.

Here, we asked SymPy to solve `Veq` for `D**3`, which is an expression, not a symbol (we will learn more about expressions in Chapter 6). To convince ourselves about this fact:

```
print("Is D**3 a symbol? {}".format((D**3).is_symbol))
print("Is D**3 an expression? {}".format(isinstance(D**3, Expr)))
```

[12] https://docs.sympy.org/latest/modules/solvers/solvers.html#sympy.solvers.solvers.solve

...This is a preview...

That's it for a simple introduction to numerical evaluation of symbolic expressions. Here we have just scratched the surface about `lambdify()`: it is so important that we will discuss it in more details in Section 7.6, Section 7.7 and Section 23.3.

5.4 Simplification of numbers

Very often our symbolic expressions will contain numbers. The type of numbers (particularly `Float` and `Rational`) might influence the execution of symbolic algorithms. In particular:

- Some algorithms might raise strange errors.
- Other algorithms might produce unexpected or wrong results.
- Other algorithms might take a very long time to conclude their task.

Consider the following basic example:

```
expr = exp(2 * x) - exp(2.0 * x)
expr
```

$$e^{2x} - e^{2.0x} \tag{5.23}$$

Clearly, `expr` is equal to zero, but SymPy didn't perform the simplification. Moreso, if we execute `simplify(expr)` it will return `expr`. This happens because the number `2.0` has been converted to an object of type `Float`, which is not an exact entity: there are rounding errors. In this simple case, we can ask `simplify()` to convert `Float` numbers to `Rational` numbers, after which the simplification should occur:

```
simplify(expr, rational=True)
```

$$0 \tag{5.24}$$

The same result could be achieved with the more powerful `nsimplify()`[16] function, which can be used to convert numbers to a simpler form. Let's consider another expression containing combinations of symbols and numbers:

```
expr = 2.5 * x + 3.33 * x**2 + 7.7777 * x**3
nsimplify(expr)
```

$$\frac{77777x^3}{10000} + \frac{333x^2}{100} + \frac{5x}{2} \tag{5.25}$$

As can be seen from the output, the floating-point numbers have been converted to rational numbers, which are exact quantities. Now, let's apply this function to a floating-point approximation of $\sqrt{2}$:

[16] https://docs.sympy.org/latest/modules/simplify/simplify.html#sympy.simplify.simplify.nsimplify

```
nsimplify(1.4142135)
```

$$\frac{2828427}{2000000} \tag{5.26}$$

As before, the floating-point number has been converted to a rational number. But we can do more: `nsimplify()`'s optional second argument is a list of constants that should be contained in the result. For example:

```
nsimplify(1.4142135, [sqrt(2)])
```

$$\sqrt{2} \tag{5.27}$$

Another example:

```
nsimplify(8.66025403784439, [sqrt(3)])
```

$$5\sqrt{3} \tag{5.28}$$

Sometimes the algorithm fails to produce the simplification. For example, we might want to replace `3.14` with π:

```
nsimplify(3.14, [pi])
```

$$\frac{157}{50} \tag{5.29}$$

This happens because the algorithm chose a very low tolerance: by default, 10^{-15} is used for objects of type `Float`. So, we can specify an appropriate tolerance:

```
r1 = nsimplify(3.14, [pi], tolerance=1e-02)
r1
```

$$\pi \tag{5.30}$$

Note what happens when the tolerance is decreased:

```
tolerances = [1e-03, 1e-04, 1e-05, 1e-06]
expr = 3.14
print("%-8s %-22s %-16s %-12s" % ("tol", "symbolic result", "numeric result",
 ↪  "approximation error"))
for t in tolerances:
    r = nsimplify(expr, [pi], tolerance=t)
    a = [t, r, r.n(), abs(expr - r.n()) / expr]
    print("%-8.e %-22s %-16f %12.e" % tuple(a))
```

```
tol      symbolic result        numeric result   approximation error
1e-03    pi                     3.141593                5e-04
1e-04    3/89 + 88*pi/89        3.140002                5e-07
1e-05    -41/100 + 113*pi/100   3.140000                1e-07
1e-06    157/50                 3.140000                0e+00
```

`pi` is included in the first three symbolic results, but they are different from each other. The approximation error is inversely proportional to the tolerance. Eventually, it reaches zero when the floating-point number is converted to a rational number. Ultimately, we are responsible to decide if 3.14 should be replaced by π or by a rational number, depending on our application.

Again, there are several methods but the one we are interested in is the following:

```
    def _eval_evalf(self, prec):
        return Float._new(self._as_mpf_val(prec), prec)
```

This method is called by SymPy when the function `evalf()` is invoked. Let's consider for example the object `pi`, which is an instance of the `Pi` class. Since `Pi` inherits from `NumberSymbol`, it exposes the `_eval_evalf()` method. Let's say we execute the command `pi.evalf()`, then SymPy is internally going to call `pi._eval_evalf(prec)` specifying the needed precision. Inside `_eval_evalf()`, we see the call to the `_as_ mpf_val(prec)` method, defined in the `Pi` class, which is going to return the numerical value. Ultimately, a SymPy's `Float` number is returned.

`NumberSymbol` also defines other methods, but we are not interested in them at this moment.

Finally, from what we have seen so far, it follows that the object `pi` is neither an instance of `Symbol`, nor an instance of `Number`. We can verify it:

```
isinstance(pi, Symbol), isinstance(pi, Number)
```

```
(False, False)
```

`pi` is a different object altogether: it disguises itself to be a number thanks to the class-attribute `is_number=True`.

5.5.3 Creating a custom Constant class

Now that we understand how `pi` and `E` work, we can build our custom `Constant` class. What exactly should this class allow us to do?

1. It should not be specific like `Pi` or `Exp1`, which only defines the number `pi` or `E` respectively. `Constant` should allow us to define arbitrary constants. Consequently, we will have to pass to the constructor both the numerical value as well as the "name" to visualize on the screen.
2. For simplicity, we will only consider integer or real constants.

The complete code is contained in the Jupyter Notebook file, which the Reader should have downloaded in Section 1.1.1. Let's see what the code does:

```
class Constant(NumberSymbol):
    is_real = True
```

We start by sub-classing `NumberSymbol`, which allows our instances of `Constant` to be evaluated. We are not going to create a singleton class because `Constant` should represent an arbitrary constant. The only class attribute we are going to define is `is_real`: this makes sense because the other attributes, like `is_positive` and `is_negative`, depend on the actual value of the constant. Therefore, they must be instance attributes!

Next, we override the constructor which was previously defined in the `AtomicExpr` class. We need to do it in order to pass custom parameters into the class:

```
def __new__(cls, value, name, latex="", pretty=""):
```

The parameters are quite self-explanatory:

1. `value`: this is the numerical value of the arbitrary constant.
2. `name`: it will be used to visualize the name of the symbol when calling the `print()` function.
3. `latex` (optional): it will be used to render the constant on the screen. If not provided, `name` will be used instead.
4. `pretty` (optional): it will be used to render the constant when using the `pprint()` function. If not provided, `name` will be used instead. Keep in mind that it is possible to visualize Unicode strings with this parameter.

Next, we make sure only numbers of type `int` and `float` are allowed to be used, thus enforcing the condition defined above. If the type of `value` is different, an exception will be raised.

```
        if not isinstance(value, (int, float)):
```

Next, the actual object is created and a few instance attributes are attached to it, so that the parameters are available in the entire object:

```
        obj = AtomicExpr.__new__(cls)
        obj._value = value
```

Then, we make sure the evaluation returns the value provided in the constructor:

```
    def _as_mpf_val(self, prec):
        return mlib.from_float(self._value, prec)
```

The library *mpmath* is still used to produce the required float precision.

Next, the approximation interval is defined:

```
    def approximation_interval(self, number_cls):
        if issubclass(number_cls, Integer):
            return (Integer(math.floor(self._value)), Integer(math.ceil(self._value)))
```

We used the functions `ceil` and `floor` from the library *math* to produce the numerical integers "bounding" the constant's value.

Finally, we have defined a few methods that are used by the different printers: they are quite self explanatory. The only enigmatic method is `_sympyrepr`: this will be called by the `srepr()` function, used to get a detailed string representation of a given object.

It's now time to perform a little test of our first custom class:

```
t = Constant(6.5, "tau", r"\tau", u"\N{Greek Small Letter Tau}")
t, t.evalf()
```

$$(\tau,\ 6.5) \tag{5.31}$$

Here we created the constant `t`, named `"tau"`. Note that we also used the Unicode name for improved printing when using the `pprint()` function; it is in the form `u"\N{UNICODE SYMBOL NAME}"` where the character `u` represent a Unicode string. A quick online search is usually enough to find the Unicode name of particular symbol.

We displayed the "symbolic" form as well as the evaluated number. It appears to be correct. Let's try the different print functions:

```
print(t)
```

tau

```
pprint(t)
```

$$\tau$$

```
srepr(t)
```

"Constant(value=6.5, name='tau', latex='\tau ')"

And now, let's run a few queries on it:

```
print(t.is_real, t.is_positive, t.is_negative, t.is_zero)
```

True True False False

As we have seen, we didn't set these instance attributes, which depend on the actual value of the constant. Their values are recomputed every time we ask for a particular assumption's value.

Now, let's try to use this constant:

```
x = symbols("x")
expr = 2 * x + t
expr
```

$$2x + \tau \tag{5.32}$$

```
expr.evalf(subs={x: 1})
```

$$8.5 \tag{5.33}$$

The evaluation appears to be correct! Let's try to solve the equation `expr = 0`:

```
solve(expr, x)
```

$$\left[-\frac{\tau}{2}\right] \tag{5.34}$$

The solution appears to be correct.

That's it for our first custom class. Keep in mind that it is just a starting point and it really needs a lot more testing to be considered safe-to-use. That's left to the interested Reader.

```python
a == b, hash(a), hash(b)
```

```
(True, 2, 2)
```

Here, the two numbers are equal, which also means their hashes are equal even though they are of different types. Let's try something similar with SymPy:

```python
c = Float(2)
d = Integer(2)
c == d, type(c), type(d)
```

```
(True, sympy.core.numbers.Float, sympy.core.numbers.Integer)
```

```python
hash(c), hash(d)
```

```
(2328901186649660154, 2)
```

It's perfectly fine if the Reader sees different hashes on the screen. The important thing to note is that `c` and `d` are equal even though their hashes are different (because their hashable contents are different). Clearly, this is an inconsistent behavior.

As of SymPy version 1.12, we can directly compare Python numbers with SymPy numbers:

```python
a == c, b == d
```

```
(True, True)
```

However, due to the different hashes we need to be extremely careful when mixing SymPy numbers and Pythons numbers into collections. Let's consider a Python set:

```python
{a, c}
```

```
{2.0, 2.0}
```

Here, we could say that we have broken the set: it clearly contains the number `2.0` multiple times, which should not happen. However, they have different hashes, hence they are included in the set.

The takeaways from this discussion are as follows:

- we need to be careful when dealing with SymPy numbers
- we also need to be extremely careful when mixing SymPy and Python objects into collections!

6.5.2 SymPy's Class Diagram

Thus far, we have talked a lot about the different types of objects interacting together in a mathematical expression: we have seen different types of numbers, symbols and functions. We have also seen the classes responsible for addition, multiplication and power operations, which can be combined together to represent subtraction and division.

A picture is worth a thousand words, though: if we were to explore SymPy source code to understand the relationships between the different classes seen thus far, we would come up with an UML class diagram. An extremely simplified version of it is shown in Figure 6.5, where a lot of things are missing for the sake of clarity; nonetheless, it gives a clear picture of SymPy's internal structure.

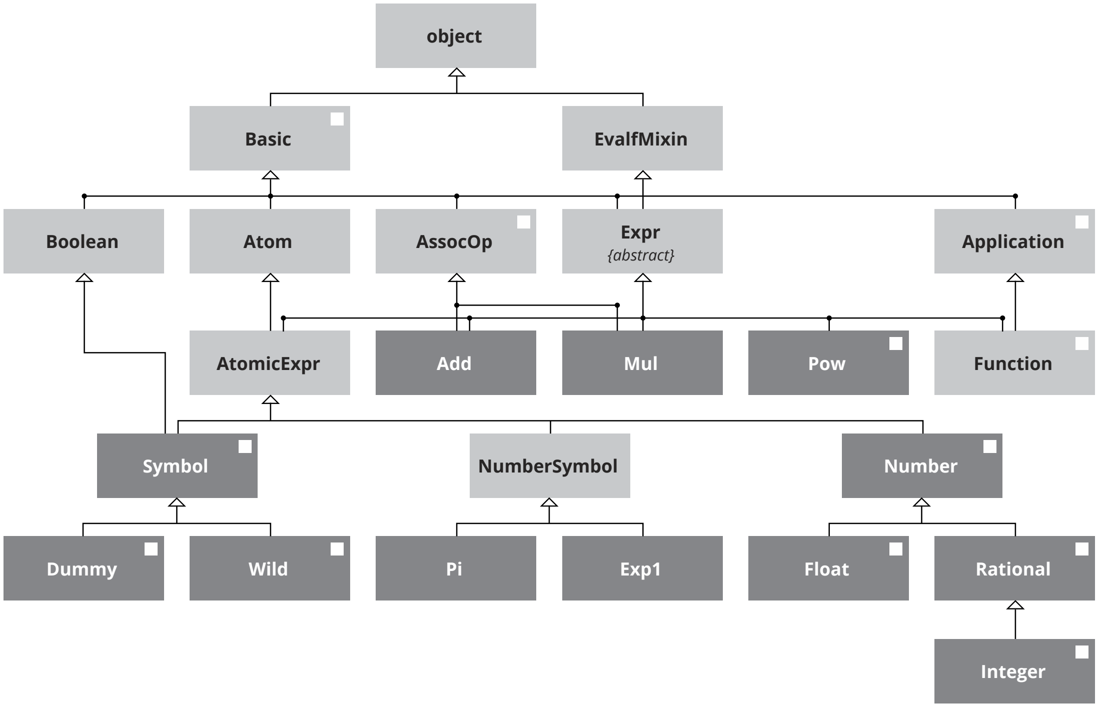

Figure 6.5: Simplified UML class diagram of SymPy expressions

In this diagram:

- Rectangles represent classes.
- Light gray rectangles represent base classes or mix-in classes: they provide functionalities to their sub-classes. Generally, they are not meant to be instantiated directly (except for `Function`). Note that there is nothing stopping us from instantiating them; for example, if `x` is an instance of `Symbol`, we can create the object `Expr(S(3), x)`. What does it represent? It is neither an addition, nor a multiplication or a power, it is not a function either. It is just a collection of terms that makes little to no sense.
- Dark gray rectangles represent classes that we can actually instantiate. Whether we should, that depends on our tasks.
- The arrows represent inheritance relationships: a given class can inherit functionalities from one or more parent classes. In practice, this also means that an instance of a given class is also an instance of the parent classes.
- Some classes have a little white square on the top-right corner, which represent the constructor method (note: this is the Author's concise way to represent this information; it

is not UML's way). As we can see, the `Symbol` class implements its constructor. On the other hand, the `Pi` class doesn't: when we instantiate the constant `pi`, Python will move through the Method Resolution Order, eventually executing the constructor defined in the `Basic` class.

Since this is a simplified diagram, let's list what's missing:

- Firstly, only class names are shown: attributes and methods are not considered. Also, only inheritance relations are displayed.
- Meta-classes are not displayed: for example, the `Singleton` meta-class is missing, which is used in the sub-classes of `NumberSymbol`.
- From SymPy version 1.7.0, the `Basic` class also inherits from the `Printable` class. It is not really important for our purposes, so it hasn't been included in the diagram.
- There are so many different types of functions that they would probably occupy an entire page. Therefore, only the `Function` base class is shown.
- The classes `One`, `Half`, `Infinity`, etc., which are sub-classes of `Number`, `Rational`, `Integer`.
- The classes `GoldenRatio`, `TribonacciConstant`, `EulerGamma`, `Catalan` which are sub-classes of `NumberSymbol`.
- The `UnevaluateExpr` class, which is a sub-class of `Expr`.

What does this diagram tell us?

- In Python, and consequently in SymPy too, everything is an object and almost everything has attributes and methods. That's clearly indicated by the inheritance from the Python's class object.
- The numerical evaluation functionality, exposed by the `evalf()` method, is implemented in the `EvalfMixin` class. Since this is a parent class to `Expr`, `evalf()` is available to every type of expressions: we can even call it on a symbol, even though it would return the symbol itself.
- `Add` and `Mul` inherit from `AssocOp`, which implements the logic for associative/ non-associative, commutative/anti-commutative operations.
- We can easily see that `Pi`, `E`, etc., are not instances of the `Number` class, even though they posses a numerical value.
- `Symbol` also inherits from `Boolean`: we will understand this inheritance in Section 8.2.

6.5.3 The Basic class

Let's look at the source code of the `Basic` class, starting from its constructor:

```
    def __new__(cls, *args):
        obj = object.__new__(cls)
        obj._assumptions = cls.default_assumptions
        obj._mhash = None  # will be set by __hash__ method.
```

```
((w1**w3)**w2).subs(d)
```

$$8^x \tag{7.25}$$

Here we used the matching dictionary to simplify the power object! Note that we could have obtained the same result with the powsimp()[7] function.

Finally, it is important to realize that match() consider the expression as a whole: it doesn't do an argument by argument match. Let's consider this example:

```
p1, p2 = symbols("p1, p2", cls=Wild)
expr = exp(x**2 + 1) + cos(x**2 + 1)
expr.match(exp(p1))
```

Here, None is returned: the expression is not an exponential function! Now:

```
expr.match(exp(p1) + p2)
```

$$\left\{p_1 : x^2 + 1,\ p_2 : \cos\left(x^2 + 1\right)\right\} \tag{7.26}$$

Here, we see that the expression is an addition between an exponential term and something else.

7.4.3 The find() method

The find()[8] method is used to retrieve all the occurrences of a given pattern. It returns an object of type set. Let's consider the following example:

```
x, y = symbols("x:y")
expr = x**2 + 2 * x + 2 * y + 2
expr
```

$$x^2 + 2x + 2y + 2 \tag{7.27}$$

Let's suppose we would like to collect together $2x$ and $2y$. We can try the collect_const()[9] function:

```
collect_const(expr, 2)
```

$$x^2 + 2\left(x + y + 1\right) \tag{7.28}$$

However, it collected a third element, which we were not interested in. We could perform this task manually, with:

```
expr.subs((2 * x) + (2 * y), Mul(2, x + y, evaluate=False))
```

[7] https://docs.sympy.org/latest/modules/simplify/simplify.html#sympy.simplify.powsimp.powsimp
[8] https://docs.sympy.org/latest/modules/core.html#sympy.core.basic.Basic.find
[9] https://docs.sympy.org/latest/modules/simplify/simplify.html#sympy.simplify.radsimp.collect_const

$$x^2 + 2(x+y) + 2 \tag{7.29}$$

This is fine for this simple example, but in real life we are most likely going to encounter much more difficult expressions, thus the risk of introducing typing errors increases exponentially. More so, we will not take full advantage of SymPy functionalities.

Let's create a couple of wild symbols and extract the multiplications:

```
w1, w2 = symbols("w1, w2", cls=Wild, exclude=[1])
expr.find(w1 * w2)
```

$$\left\{2x, x^2, 2y\right\} \tag{7.30}$$

Why did we also get x^2? It turns out that SymPy is able to recognize that it is equivalent to $x \cdot x$. We can verify it with:

```
(x**2).match(w1 * w2)
```

$$\{w_1 : x,\ w_2 : x\} \tag{7.31}$$

Therefore, we need to be more specific with the wild symbols. Here, we will create `w1` to match numbers:

```
w1 = Wild("w1", exclude=[1], properties=[lambda e: e.is_number])
w2 = Wild("w2", exclude=[1])
r = expr.find(w1 * w2)
r
```

$$\{2x, 2y\} \tag{7.32}$$

At this point, we sum up those terms to represent the subexpression that we would like to modify:

```
r = Add(*list(r))
r
```

$$2x + 2y \tag{7.33}$$

Finally:

```
expr.subs(r, r.factor())
```

$$x^2 + 2(x+y) + 2 \tag{7.34}$$

Let's consider another example:

```
x, y, z = symbols("x, y, z")
expr = y * (x**2 + 4 * x + 4) + z * (x**2 * y**2 + 2 * x**2 * y * z + x**2 * z**2)
expr
```

$$y\left(x^2 + 4x + 4\right) + z\left(x^2y^2 + 2x^2yz + x^2z^2\right) \tag{7.35}$$

There are two quadratic expressions that we would like to factor. We can try with:

```
expr.factor()
```

$$x^2y^2z + 2x^2yz^2 + x^2y + x^2z^3 + 4xy + 4y \tag{7.36}$$

This is not the result we were looking for. In fact, the expression has only been expanded, not factored. We have to try something different.

Note that those quadratic forms are additions of three terms. Hence, we can use the `find()` method to select the additive terms of the expression:

```
expr.find(Add)
```

$$\left\{y\left(x^2 + 4x + 4\right) + z\left(x^2y^2 + 2x^2yz + x^2z^2\right), x^2 + 4x + 4, x^2y^2 + 2x^2yz + x^2z^2\right\} \tag{7.37}$$

Obviously, it selected too much. We can use a wild symbol and apply the correct properties, for example:

```
w1 = Wild("w1", properties=[lambda e: isinstance(e, Add) and (len(e.args) == 3)])
r = expr.find(w1)
r
```

$$\left\{x^2 + 4x + 4, x^2y^2 + 2x^2yz + x^2z^2\right\} \tag{7.38}$$

Here, we selected additive terms having 3 arguments! Now, we can create a substitution dictionary:

```
d = {k: k.factor() for k in r}
d
```

$$\left\{x^2 + 4x + 4 : (x+2)^2,\ x^2y^2 + 2x^2yz + x^2z^2 : x^2(y+z)^2\right\} \tag{7.39}$$

Finally:

```
expr.subs(d)
```

$$x^2z(y+z)^2 + y(x+2)^2 \tag{7.40}$$

7.4.4 Wild functions

Similarly to `Wild` symbols, a `WildFunction` matches any function with its arguments. Its constructor only requires two arguments:

- `name`: the usual name of the function, displayed on the screen;
- `nargs`: the number of arguments or a tuple to match a range of arguments.

For example:

```
x = symbols("x")
f = Function("f")(x)
expr = f + x * cos(4 * x) + sin(x**2) - exp(x)
expr
```

13.3 Exercise - The Integral.transform() method

Solve the following integral:

$$\int Ce^{-Ea} \sinh\left(\sqrt{Eb}\right) dE \tag{13.37}$$

where $a, b, C, E \in \mathbb{R}$ and $a, b, E \geq 0$.

13.3.1 Solution

Let start by creating the symbols and expression:

```
a, b, Ee = symbols("a, b, E", real=True, positive=True)
C = symbols("C", real=True)
expr = C * exp(-a * Ee) * sinh(sqrt(b * Ee))
```

Note that we used the variable `Ee` instead of `E` to represent a symbol of name `"E"`, in order to avoid overriding the name `"E"` in the global namespace which is currently assigned to SymPy's special symbol/number `E`.

All we need to do is:

```
integrate(expr, Ee)
```

$$C \int e^{-Ea} \sinh\left(\sqrt{E}\sqrt{b}\right) dE \tag{13.38}$$

Whenever the `integrate()` method returns an unevaluated integral, that means none of the algorithms were able to evaluate it. We may think that we run out of options; however, we still have to try expression manipulation!

Our expression contains the hyperbolic `sinh` function, which can be rewritten in terms of exponential functions:

```
expr = expr.rewrite(exp)
expr
```

$$C\left(\frac{e^{\sqrt{E}\sqrt{b}}}{2} - \frac{e^{-\sqrt{E}\sqrt{b}}}{2}\right) e^{-Ea} \tag{13.39}$$

We saw in the previous exercises that it is a good idea to expand the expression and simplify the powers:

```
expr = expr.expand().powsimp()
expr
```

$$-\frac{Ce^{-\sqrt{E}\sqrt{b}-Ea}}{2} + \frac{Ce^{\sqrt{E}\sqrt{b}-Ea}}{2} \tag{13.40}$$

Now let's integrate:

```
integrate(expr, Ee)
```

$$\frac{C\left(\int -e^{-\sqrt{E}\sqrt{b}}e^{-Ea}\,dE + \int e^{\sqrt{E}\sqrt{b}}e^{-Ea}\,dE\right)}{2} \tag{13.41}$$

Again, no luck. What could possibly be wrong with our expression?

Looking at Expression (13.40), the argument of the exponential function has the form $\sqrt{E} + E$ (disregarding the constants). Would SymPy be able to integrate an exponential function with a different argument, like $E + E^2$? Let's try:

```
integrate(exp(-(Ee + Ee**2)), Ee)
```

$$\frac{\sqrt{\pi}e^{\frac{1}{4}}\operatorname{erf}\left(E + \frac{1}{2}\right)}{2} \tag{13.42}$$

The integral was evaluated! It contains erf (), the error function[1]. It seems that if the integration symbol (E in our case) appears in the argument of the exponential function with a rational exponent ($\sqrt{E} = E^{1/2}$ in our case), the algorithms will have a hard time to solve the integral.

It would be nice if we could easily apply a change of variable, for example setting $x = \sqrt{E}$. Luckily, the `Integrate` class exposes the `transform()`[2] method to do just that. If we were dealing with a definite integral, `transform()` would also modify the limits of integration (if needed). Let's see how we can use it to perform a *u-substitution*:

```
x = symbols("x", real=True, positive=True)
i = Integral(expr, Ee).transform(sqrt(Ee), x)
i
```

$$\int 2x\left(-\frac{Ce^{-ax^2-\sqrt{b}x}}{2} + \frac{Ce^{-ax^2+\sqrt{b}x}}{2}\right)\,dx \tag{13.43}$$

As we can see, the expression has been modified accordingly. Note that we created the symbol `x` with the same assumptions of the symbol `Ee`.

To evaluate the integral:

```
r = i.doit()
r
```

$$\begin{aligned} &- C\left(-\frac{e^{-ax^2}e^{-\sqrt{b}x}}{2a} - \frac{\sqrt{\pi}\sqrt{b}e^{\frac{b}{4a}}\operatorname{erf}\left(\sqrt{a}x + \frac{\sqrt{b}}{2\sqrt{a}}\right)}{4a^{\frac{3}{2}}}\right) \\ &+ C\left(-\frac{e^{-ax^2}e^{\sqrt{b}x}}{2a} + \frac{\sqrt{\pi}\sqrt{b}e^{\frac{b}{4a}}\operatorname{erf}\left(\sqrt{a}x - \frac{\sqrt{b}}{2\sqrt{a}}\right)}{4a^{\frac{3}{2}}}\right) \end{aligned} \tag{13.44}$$

The computation will take several minutes, eventually producing the correct result shown above. Finally, to complete the task we need to substitute back $\sqrt{E}$:

[1] https://en.wikipedia.org/wiki/Error_function

[2] https://docs.sympy.org/latest/modules/integrals/integrals.html#sympy.integrals.integrals.Integral.transform

```
r.subs(x, sqrt(Ee))
```

$$- C \left(- \frac{e^{- \sqrt{E} \sqrt{b}} e^{- E a}}{2 a} - \frac{\sqrt{\pi} \sqrt{b} e^{\frac{b}{4 a}} \operatorname{erf}{\left(\sqrt{E} \sqrt{a} + \frac{\sqrt{b}}{2 \sqrt{a}} \right)}}{4 a^{\frac{3}{2}}}\right) + C \left(- \frac{e^{\sqrt{E} \sqrt{b}} e^{- E a}}{2 a} + \frac{\sqrt{\pi} \sqrt{b} e^{\frac{b}{4 a}} \operatorname{erf}{\left(\sqrt{E} \sqrt{a} - \frac{\sqrt{b}}{2 \sqrt{a}} \right)}}{4 a^{\frac{3}{2}}}\right) \tag{13.45}$$

We are extremely happy about the outcome; however, an important question remains: why did the computation take so long? A reasonable assumption is that the function to be integrated was not expanded before evaluation.

We can try a different approach to solve this beast: starting from Expression (13.40), using the linearity property of integrals, we are going to apply the change of variable and integration to the separate terms:

```
i = expr.func(*[Integral(a, Ee).transform(sqrt(Ee), x) for a in expr.args])
i
```

$$\int \left(- C x e^{- a x^{2} - \sqrt{b} x}\right)\, dx + \int C x e^{- a x^{2} + \sqrt{b} x}\, dx \tag{13.46}$$

Then, to evaluate it:

```
r = i.doit()
r
```

In a matter of a few seconds, we will get the previous results, Expression (13.44). That's a huge improvement in performance thanks to expression manipulation!

Chapter 15

The Equation class

Previously we have seen that we can represent equations in two ways:

1. an expression in the form of *"(left-hand side) - (right-hand side)"* (the equal to zero part is implied).
2. `Equality` objects, thus specifying the left and right-hand sides. However, relationals do not inherit from the `Expr` class, hence these objects do not support mathematical operations. We are severely limited on the amount of manipulations we can apply to them.

We are now going to explore the `Equation` class, implemented in the *algebra_with_sympy*[1] module first installed in Section 1.1.3. This class allow us:

- To write equations by specifying the left-hand side (LHS) and the right-hand side (RHS).
- To perform mathematical operations on equations. For example, adding the same expression to both sides of the equation, or adding two equations side by side.
- To evaluate derivatives and integrals.
- To apply the most common manipulation techniques, namely `simplify()`, `collect()`, `factor()`, `expand()`, etc.
- To apply any function to both sides, for example the `sin` function.
- To numerically evaluate the equation.
- To nicely print the equation on the notebook.

Let's briefly discuss the necessary *import* statements and some optional configurations:

```
from sympy import *
from algebra_with_sympy import *
algwsym_config.output.label = False
algwsym_config.output.solve_to_list = True
```

The order of the *import* statements is important. With the second one, we are importing:

[1] https://github.com/gutow/Algebra_with_Sympy

- the `Equation` class.
- the `solve` function, which is just a wrapper to SymPy's `solve` function and it is capable of dealing with objects of type `Equation`.
- the `algwsym_config` object, which can be used to configure the module. By default:
  - equations will be shown with a label on the RHS. Depending on the use case, this can be either useful or annoying. Hence, with the third line of code we are going to hide the labels.
  - the `solve` function implemented in this module returns a `FiniteSet` containing the solutions, which can't be indexed. Hence, the last line of code forces the the `solve` function to return a list of solutions.

It is left to the Reader to change these options and explore the different outputs. Let's quickly introduce the most common functionalities:

```
a, b, c, x = symbols("a, b, c, x")
eq1 = Equation(a + b, c * x)
eq2 = Equation(x**2 - 4 * x, a + b)
display(eq1, eq2)
```

$$a + b = cx \qquad x^2 - 4x = a + b \tag{15.1}$$

Similarly to the `Equality` class, we can use the following attributes to get the LHS, the RHS and to swap both sides of an equation:

```
display(eq1.lhs, eq1.rhs, eq1.reversed)
```

$$a + b \qquad cx \qquad cx = a + b \tag{15.2}$$

When the `solve` function receives objects of type `Equation`, it will return a list of results of this type:

```
res = solve(eq1, x)
res
```

$$\left[x = \frac{a+b}{c}\right] \tag{15.3}$$

We can apply any mathematical operation on both sides of the equation simultaneously:

```
eq1 + 2
```

$$a + b + 2 = cx + 2 \tag{15.4}$$

We can also combine two equations side by side with a mathematical operation, for example:

```
eq1 / eq2
```

$$\frac{a+b}{x^2-4x} = \frac{cx}{a+b} \tag{15.5}$$

Alternatively, we can use the applylhs or applyrhs methods to apply an operation to a particular side. Here, we are going to square the RHS of the first equation:

```
eq1.applyrhs(lambda t: t**2)
```

$$a+b = c^2x^2 \tag{15.6}$$

We can use the subs method to look and replace something on both sides of the equation:

```
eq1.subs(c, 2)
```

$$a+b = 2x \tag{15.7}$$

The subs method of an Equation also accepts objects of the same type: the LHS represents the pattern to look for, the RHS represents the substitution:

```
eq1.subs(eq2.reversed)
```

$$x^2 - 4x = cx \tag{15.8}$$

As always, when it comes to do real world testing of new functionalities it is always a good idea to start with something easy with which we are comfortable. In the following sections we will learn how to use the Equation class and understand the differences between working with "pen and paper" versus SymPy. The unfamiliar Reader should pay attention to the involved manipulation techniques, not on the physics of the problem.

15.1 Example - Electric Circuit

Figure 15.1 shows a simple electric circuit containing a resistance, an impedance, two capacitors and one voltage source. We would like to study the evolution in time of i_L, V_{C_1}, V_{C_2}.

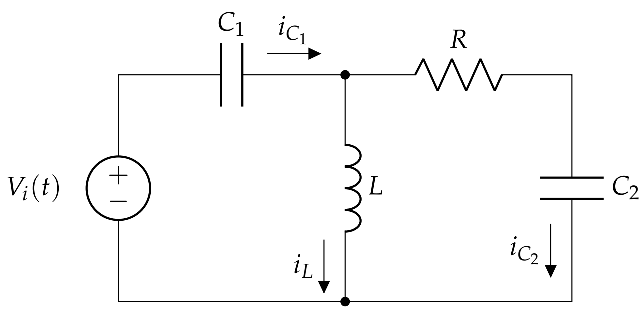

Figure 15.1: Electric Circuit

We start by defining the necessary symbols and functions:

Chapter 17

Matrices and Linear Algebra

When it comes to dealing with multi-dimensional arrays of symbolic expressions, SymPy provides different modules:

1. `sympy.matrices`: it implements the functionalities to work with two-dimensional arrays of symbolic expressions and linear algebra.
2. `sympy.tensor`: it implements the functionalities to work with multi-dimensional arrays of symbolic expressions, to perform tensor products, contractions and derivatives with respect to arrays.
3. `sympy.vector`: it provides the classes to work with vectors, that is, 3-dimensional entities having both a magnitude and a direction defined in a coordinate system.

At the time of writing this book, these three modules are not really designed to work together; however, a few methods are available to convert an object (a matrix, a tensor or a vector) from one module to the other. In this chapter we are going to explore the `matrices` module and the linear algebra capabilities provided by SymPy. In the next chapters we will look into tensors and vectors.

Let's start from the definition: a matrix is a rectangular array of numbers, symbols or expressions. If the matrix has only one column, we refer to it as a *column vector*. If the matrix has only one row, we refer to it as a *row vector*. Keep in mind that in this chapter we are talking about matrices!

Table 17.1 shows the main classes available to work with matrices. The first four classes represent *explicit matrices* in which we can access and eventually modify the elements. *Mutable* or *Immutable* refers to the capability to modify the single entries after the matrix has been created. *Dense* or *Sparse* refers to whether to store in memory all elements or only the non-zeros. The `MatrixExpr` class allows to write expressions involving matrix-symbols in which the elements are not yet defined. The `DomainMatrix` class is a recent addition aiming to improve performance of many matrix operations, in which a domain (Section 6.3.3) is associated to a matrix (conceptually, it is similar to NumPy's *dtype*). As of SymPy 1.12, this class is under heavy development, therefore it won't be covered here. However, in future SymPy versions, it is very likely that the algorithms implemented into this class will also be used by the aforementioned matrix classes, thus making them much faster.

Class Name	Alias
MutableDenseMatrix	Matrix
ImmutableDenseMatrix	ImmutableMatrix
MutableSparseMatrix	SparseMatrix
ImmutableSparseMatrix	
MatrixExpr	
DomainMatrix	

Table 17.1: Main classes of matrices

17.1 Explicit Matrices

17.1.1 Basic Usage

In the following section, we are going to *extend* the content of this documentation page[1]. Specifically, we will learn how to work with the `MutableDenseMatrix` and `ImmutableDenseMatrix` classes.

As we will see, there are quite a lot of attributes and methods to be aware of; chances are that we will forget something over time. Hence, a basic cheat sheet can be found in Appendix B.7 and Appendix B.8. The interested Reader should go through SymPy documentation and extend it.

Generally, we are going to create a (mutable dense) matrix with its alias `Matrix`:

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

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \tag{17.1}$$

We can also specify the number of rows and columns followed by a list of entries:

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

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \tag{17.2}$$

To create a column or a row vector:

```
Matrix([1, 2, 3]), Matrix([[1, 2, 3]])
```

$$\left(\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}, \begin{bmatrix} 1 & 2 & 3 \end{bmatrix} \right) \tag{17.3}$$

We can also create special types of (mutable) matrices, namely *ones*, *zeros*, *eye*, *diag*, by calling their respective methods. For `ones()`, `zeros()` and `eye()`, the methods accept either:

- one argument, specifying the size of the square matrix;

[1] https://docs.sympy.org/latest/modules/matrices/matrices.html

Chapter 21

Assumptions

In Section 2.1.2 we introduced assumptions on objects of type `Symbol`. They are defined in the module `sympy.core.assumptions`, which is also referred to as the *"old assumptions module"*.

In this chapter we will explore the *"new assumptions module"*, `sympy.assumptions`, to understand its capabilities, its limitations and why SymPy currently implements two assumption modules. We will try to maintain the discussion "light" on technical details, so that anyone can follow.

21.1 New Assumptions Module

21.1.1 The Need for New Assumptions

So far we have used the *old assumptions module* in which assumptions are set on symbols and are stored in the expressions. For example:

```
x, y = symbols("x, y", positive=True)
expr = x + y
expr.is_positive, expr.is_zero, expr.is_negative
```

```
(True, False, False)
```

When querying for a given attribute, like `expr.is_positive`, SymPy's inference engine will try to compute the attribute's value by calling the respective _eval_is_<ASSUMPTION_NAME>() method defined on the different classes. In our example, it called the `_eval_is_positive()` method defined in the `Expr` class. If it's not possible to determine the value, `None` will be returned. These assumptions are used all across SymPy codebase, for example:

- In core classes like `Add`, `Mul` and `Pow` where arguments are evaluated and eventually modified.
- They are responsible for evaluation of functions, for example `cos(n * pi)` where `n` is a number.
- In more complex routines, like `simplify()`, etc.

- In different solvers.

They are so embedded into the core that at one point in time (many years ago), the following observations were made:

1. It became clear that refactoring the code was getting harder and harder.
2. There were performance concerns due to evaluation. For example, let's consider `sqrt(x**2)` when `x` is real; this is going to call `Pow(Pow(x, 2), S.Half)`. Inside `Pow`'s constructor the assumptions on `x` are checked, eventually evaluating the result to `Abs(x)`. Do we really need this automatic evaluation? This might be fine for short expression, but for longer ones it definitely adds up computational time.
3. while being very useful they are also quite limited. In short, they are focused on numbers and sets, with attributes like `is_positive`, `is_real`, etc. Let's suppose we have two symbols, `x` and `y`: we can assume `x > 0` and/or `y > 0`, but we cannot assume `x > y`.

Therefore, efforts were made to develop a new module in which assumptions were separate entities from the object that was being assumed about. Meet the *"new assumptions module"*, `sympy.assumptions`, first introduced in 2009. The original idea was to remove the "old assumptions" and replace them with the "new assumptions"; many attempts have been made over the years, all of them failed. There are several reasons for this, here we only mention the most important ones: first, "old assumptions" are used everywhere in the core, thus requiring a tremendous amount of work; second, "new assumptions" were not an option in the first place, being very slow (a lot of improvements have been made since then, dramatically improving the situation. Still, the "old assumptions" are fastest).

Without further ado, let's explore the new assumptions.

21.1.2 The "New Assumptions Module"

With the new assumptions module[1], assumptions can be made on expressions, not just symbols. They are not limited to numeric and set attributes, like `is_positive`, `is_real`, etc. As a matter of fact, we can use them to query matrices about their nature, for example if it is symmetric, diagonal, etc.

We already mentioned that, within this module, assumptions are separate entities from the object that is being assumed about; let's understand what that means. From a user's point of view, the module exposes three pillars:

1. `Q`: this class is used to create predicates. A predicate is a relation (or function) over some argument, used to assign a property to the argument or to relate two or more arguments to each other[2]. Let's consider the following example:

```
x = Symbol("x")
p = Q.positive(x)
print(type(p), p.args)
```

[1] https://docs.sympy.org/latest/modules/assumptions/index.html#module-sympy.assumptions

[2] https://en.wikipedia.org/wiki/Predicate_(mathematical_logic)

22.5 Parametric-Interactive Plots

It is often difficult to fully understand the behaviur of a symbolic expression by inspection, especially when several parameters are involved. This is particularly true when studying physics and engineering problems.

The plotting functions exposes the `params` keyword argument. When it is set, it creates parametric interactive plots with widgets, that is, interactive controls such as sliders, buttons, spinners, etc. We can think of any symbolic expression as a collection of symbols:

- some symbols represent the domain of the function, which will be discretized according to some strategy.
- all other symbols represent *symbolic parameters*. Eventually, they will be substituted by numeric values and the expression will be numerically evaluated over the discretized domain.

The widgets are created with *ipywidgets*[16], but Holoviz's *Panel*[17] is supported too. As a consequence, parametric-interactive plots only work within Jupyter Notebook. A design goal for this functionality was to be easy to use: for basic stuffs the Reader doesn't need to know anything about *ipywidgets* or *Panel*. However, when dealing with *Panel*, a little introduction to *Param*[18] is necessary in order to improve customization. As the name suggests, *Param* let us create parameters, which are extended Python attributes to support types, ranges, and documentation, which will then be used by *Panel* to create widgets. *Param*'s documentation does a good job illustrating the concepts of operation. Let's import this library:

```
import param
```

Each symbolic parameter must receive exactly one numeric value. Therefore, only widgets that returns exactly one numeric value can be used. Here is a list of *Param*'s parameters that, once rendered as widgets, satisfy that condition (Figure 22.4):

- `param.Integer` or `param.Number`: represents an integer or a floating point number, respectively. If bounded (having valid minimum and maximum values) it will be rendered as a slider, otherwise it will be rendered as a spinner.
- `param.ObjectSelector`: represents a list of possible choices. It will be rendered as a drop-down list.
- `param.Boolean`: it will be rendered as a check box.

We are now ready to explore a couple of examples.

22.5.1 Example - Fourier Series Approximation

Let's consider a sawtooth wave[19], defined as:

[16] https://ipywidgets.readthedocs.io/en/latest/examples/Widget%20List.html
[17] https://panel.holoviz.org/index.html
[18] https://panel.holoviz.org/user_guide/Param.html
[19] https://mathworld.wolfram.com/SawtoothWave.html

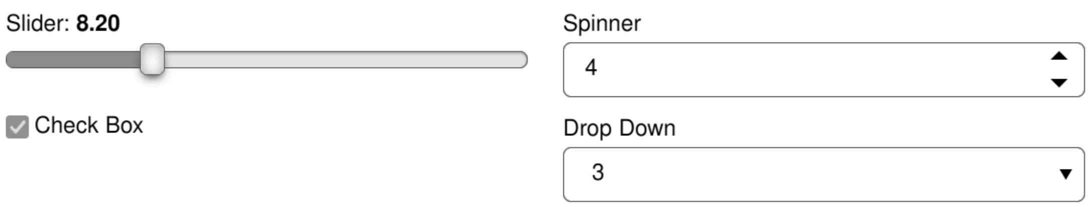

Figure 22.4: Widgets producing one numeric output value

$$S(x) = \text{frac}\left(\frac{x}{T}\right) \tag{22.1}$$

where T is the period and x is the domain. We'd like to understand its Fourier Series approximation[20]:

$$f(x) = \frac{1}{2} - \frac{1}{\pi}\sum_{n=1}^{\infty}\frac{1}{n}\sin\left(\frac{2\pi n x}{T}\right) \tag{22.2}$$

Let's start by creating the above expressions:

```
x, T, n, m = symbols("x, T, n, m")
sawtooth = frac(x / T)
# Fourier Series of a sawtooth wave
fs = S(1) / 2 - (1 / pi) * Sum(sin(2 * n * pi * x / T) / n, (n, 1, m))
```

Note that we stopped the Fourier Series at `m` rather than infinity, because `m` represents the upper bound of the approximation. The higher the value of `m`, the more accurate the approximation. In the above expressions:

- `T` is a positive float number.
- `m` is a positive integer.

This is all the necessary code in order to create a parametric interactive plot:

```
from bokeh.models.formatters import PrintfTickFormatter
formatter = PrintfTickFormatter(format="%.3f")

plot(
    (sawtooth, (x, 0, 10), "f"),
    (fs, (x, 0, 10), "approx"),
    params = {
        T: (2, 0, 10, 80, formatter),
        m: param.Integer(3, bounds=(1, None), label="Sum up to n ")
    },
    imodule = "panel",
    xlabel = "x",
    ylabel = "y",
    backend = PB
)
```

[20] https://mathworld.wolfram.com/FourierSeriesSawtoothWave.html

Let's break down the different pieces:

- First, we created a `formatter` object: it will be used to format the value shown by the slider. It is particularly useful when dealing with small floating point numbers. If a formatter is not used, the slider's value will be rendered with two decimal places. Note: this functionality is only supported by *Panel*, because *ipywidgets* doesn't provide any way to format the value shown by the slider.
- `(sawtooth, (x, 0, 10), "f")`: the arguments are the usual tuples specifying the expression, the range and an optional label to be used in the legend. This is a simple 2D example, but vector fields, functions of two variables, geometric entities and complex functions are supported as well.
- `params`: this dictionary maps *symbolic parameters* to some widgets.

  We created an integer parameter associated to `m`, having a default value of 3, minimum value of 1, no maximum value and a custom label. *Panel* will then render it with a spinner.

  Instead of creating a `param.Number` associated to `T`, we are using a shortcut to create a slider. The tuple must have the form *(default, min, max, N, tick_format, label, spacing)* where the first three elements are mandatory. In particular:

  - *N* is the number of steps in the slider.
  - *tick_format* is the formatter to be used on the value shown by the slider. This only works with *Panel* and should not be provided if we are using *ipywidgets*.
  - *label*: set the visible label of the slider. If not provided, the latex representation of the symbol will be used instead.
  - *spacing*: the default value is `"lin"`, which creates a linear spacing between the steps of the slider. Alternatively, it can be `"log"` to use a logarithmic spacing.

- `imodule="panel"` selects *Panel* as the interactive widget library. If not provided, *ipywidgets* will be used instead.
- Finally, the usual keyword arguments to customize the plot are provided.

The resulting interactive application is shown in Figure 22.5. A few observations are in order:

- By default, the widgets are shown on top of the plot and are automatically layed out from left to right. With many parameters a *grid* will be created. We can control the number of columns with the `ncols` keyword argument, whose default value is 2.
- The same performance considerations are valid: Matplotlib is the fastest at rendering 2D plots, followed by Bokeh while Plotly is the slowest. For 3D plots, K3D-Jupyter is the fastest, followed by Plotly while Matplotlib is the slowest.
- When no label is provided to a parameter, the Latex representation of the symbol will be used instead. *ipywidgets* is capable of rendering latex labels. However, *Panel* might not. We can turn off the Latex labels by setting `use_latex=False`: in this case the string representation of the symbol will be used. If the result is not good enough, we might need to manually set the labels for each parameter, avoiding Latex code altogether.

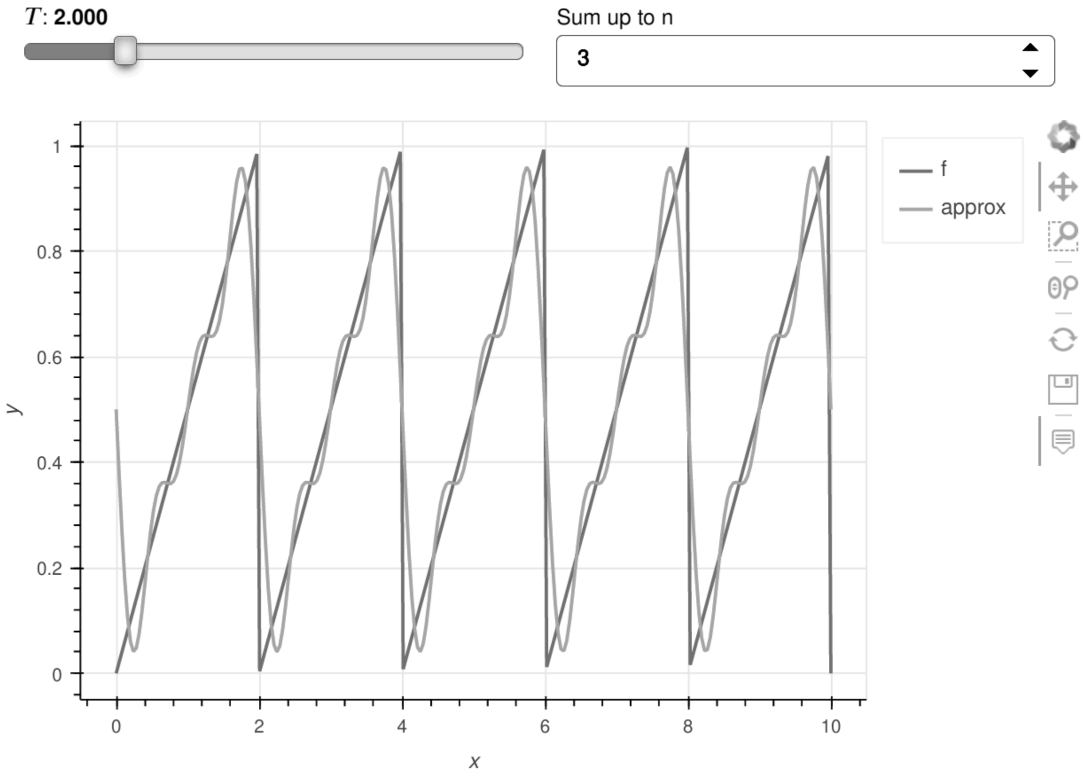

Figure 22.5: Resulting interactive application with `BokehBackend`

22.5.2 Example - Temperature Distribution

Let's consider an annular rod of nuclear fuel. Let:

- L: the length of the rod.
- z: the position along the rod, $0 \leq z \leq L$. It represents the domain of our plot, whereas all other variables should be considered as parameters.
- r_i and r_o: radius of the inner and outer walls, respectively. It must be $r_i < r_o$. A cooling fluid is going to traverse the cylindrical volume for $r < r_i$, also known as the inner channel.
- P_{ave}: average thermal power generated by the rod.
- $\dot{m}$: mass flow rate of the cooling fluid.
- T_{in}: temperature of the cooling fluid at the inlet.
- k: thermal conductivity of the nuclear fuel.
- c_p: specific heat of the nuclear fuel.

Chapter 23

Printers and Code Generation

SymPy offers two important modules:

1. *Printing module*[1]: useful to generate an appropriate representation of any SymPy expression. It exposes different *printer* classes which can be categorized into:

   - Representation printers: they convert a SymPy expression to some representation intended to be shown to the user.
   - Code printers: they convert a SymPy expression to a target programming language. This is very convenient as it saves a time-consuming step in which we could possibly insert typing errors.

2. *Code generation module*[2]: built on top of the printing module, it creates compilable and executable code starting from symbolic expressions in a variety of programming languages. For example, we can use this module to speed up the numerical evaluation of our expressions.

23.1 The Printing Module

Figure 23.1 shows a simplified UML class diagram of the printing module. As we can see, there is a huge number of classes, each one responsible for a different representation. Luckily, we don't have to instantiate them directly, instead we can use the convenient wrapper functions listed in Table 23.1. So far, we have used:

- The string representation, visualized with the `print()` function. Internally, it is going to call the `Basic.__str__()` method, which is going to use `sstr()`, a wrapper that instantiate the `StrPrinter` class.
- The repr form, visualized with `srepr()`, a wrapper that instantiates the `ReprPrinter` class.

[1] https://docs.sympy.org/latest/modules/printing.html
[2] https://docs.sympy.org/latest/modules/codegen.html

Chapter 24

Dynamical Systems and Simulations

In Chapter 20 the main differences between *sympy.vector* and *sympy.physics.vector* modules were introduced. We are now going to explore the latter, which serves as the building block to generate equations of motion of dynamical systems (*EoM* from now on) using the *sympy.physics.mechanics* module. After the EoM are generated, they can be numerically integrated in order to extract information about the system.

But first, let's adjust our expectations. It is assumed that the Reader is familiar with dynamical system theory. Moreover, as of SymPy version 1.12, there is no visual editor to easily draw our systems: everything must be coded appropriately and attention must be placed in order to not introduce errors. Currently, two supported ways are available to configure a dynamical system, with a third one being experimental:

1. The oldest and traditional way, which consists in manually creating everything: reference frames, points (and setting their velocities), particles and rigid bodies, loads, etc. Correctly setting the velocities is crucial in order to create EoM.
2. The *joints framework*[1], where the joints create the connection between the bodies. Many types of joints are available: pin joint, prismatic joint, weld joint, spherical joint, etc. In short, the location of a joint is defined with respect to the center of mass of two bodies. This framework has pros and cons:

    - One advantage is that it automatically sets the velocities of the points in the rigid bodies, reducing the likelihood of introducing typing errors.
    - Another advantage is that it automatically creates kinematic differential equations, which again reduces the likelihood of introducing typing errors.
    - As of SymPy version 1.12, the documentation of the different types of joints is well written. However, there are only two examples about using this framework, placed in different sections (hence, hard to find), which do not cover the framework in sufficient details.

[1] https://docs.sympy.org/latest/modules/physics/mechanics/joints.html

- This frameworks is built on top of the traditional way and it should simplify the coding of dynamical system. This is generally true for open loop systems with no dependent generalized coordinates. However, things might get complicated if closed kinematic loops are present in the system. This concept will be clarified with the examples of this book.
- Another *disadvantage* is that setting up the dynamical system with this framework requires about the same amount of code (if not more) than the traditional approach.

3. An experimental `System` class, built on top of the *joints framework*, which should simplify the coding of dynamical systems. As of SymPy version 1.12, this is not available to general users (it can be found on the development version on GitHub), hence it won't be discussed any further.

Once the system has been properly formulated, Lagrangian Mechanics or Kane's Method can be used to generate the EoM:

- `KanesMethod`[2]. These are the general steps needed to appropriately code the dynamical system following the traditional approach:

  1. Define the generalized coordinates and speeds, and choose which ones are independent and which ones are dependent.
  2. Create the reference frames and points. Set their coordinates and velocities.
  3. Create the kinematic differential equations.
  4. Define the configuration constraints, velocity constraints and acceleration constraints, if the system requires them.
  5. Define the particles and/or rigid bodies of the system.
  6. Create the loads, which is a list of tuples of the form *(point of application, force or torque vector)*.
  7. Create a `KanesMethod` object and generate the EoM.

- `LagrangesMethod`[3]. These are the general steps needed to appropriately code the dynamical system following the traditional approach:

  1. Define the generalized coordinates.
  2. Create the reference frames and points. Set their coordinates and velocities.
  3. Define the holonomics and nonholonomics constraints, if the system requires them.
  4. Define the particles and/or rigid bodies of the system.
  5. Create the Lagrangian by defining the potentials of the system.
  6. Create a `LagrangesMethod` object and generate the EoM.

[2] https://docs.sympy.org/latest/modules/physics/mechanics/api/kane_lagrange.html#sympy.physics.mechanics.kane.KanesMethod

[3] https://docs.sympy.org/latest/modules/physics/mechanics/api/kane_lagrange.html#sympy.physics.mechanics.lagrange.LagrangesMethod

The choice of the method is personal, although sometimes it is easier to formulate a problem with one method rather than the other. All examples of this book will use Kane's Method.

The *sympy.physics.mechanics* module is quite large, so it won't be possible to cover all functionalities here. For example, the linearization of EoM won't be explored. The computation of velocities, angular velocities and moments of inertia of rigid bodies will only be mentioned. At the time of writing this chapter, new classes representing actuators are being implemented, hence, they won't be covered here. Instead, this chapter is designed to bring the Reader up to speed with the coding of dynamical systems in order to generate EoM and extract useful information from them.

The extractaction of information from EoM will be done with the *PyDy* module (installed in Section 1.1.4), which provides functionalities to quickly integrate EoM using SciPy's `odeint()` as well as a useful, albeit limited, framework to visualize the motion of our dynamical system.

As of SymPy version 1.12, the official documentation definitely needs improvements: it is mainly good for setting up very simple problems; it doesn't even mention the extraction of information, nor it points to PyDy. PyDy's documentation is good, but it assumes that we are already familiar with the *mechanics* module. A very good starting point to learn this module is given by Florian-Čermàk[4]. Another great resource is Gosh et al.[5], from which a couple of examples have been adapted and further extended in order to introduce the joints framework.

Finally, all the following sections are meant to be followed in order. The first examples will be very well explained. Moving on, the Author won't repeat previously explained concepts.

24.1 Frames of Reference and vectors

Let's start with a few definitions:

- A **vector** is a geometric object that has magnitude (or length) and direction.
- A **frame of reference** (or reference frame) is an entity from which we observe vector quantities. A vector $\vec{v}$ is fixed in a reference frame N if none of its properties ever change when observed from N. The properties of the same vector can change if observed from a different reference frame. A right-handed orthonormal basis vector set is assigned to each reference frame. For example, $\hat{n}_x, \hat{n}_y, \hat{n}_z$ is assigned to N. These basis vectors can describe the orientation of N relative to other reference frames.
- A **coordinate system** describes the motion of points or bodies in a reference frame. The mechanics module uses a Cartesian coordinate system aligned with the basis vectors of a reference frame.

First, we import the necessary functionalities:

```
from sympy import *
import sympy.physics.mechanics as me
me.init_vprinting()
```

[4] Solving Multibody Dynamics Problems Using Python. Pavel Florian, Roman Čermàk. Design of Machines and Structures, Vol. 7, No. 1 (2017), pp. 15-22.

[5] Undergraduate dynamics using the logic of multibody dynamics - Indigenous code and an open-source software. Sunavo Ghosh, Arghya Nandi, Sumanta Neogy. Computer Applications in Engineering Education, Volume 30, Issue 1, January 2022.

In particular, `me.init_vprinting()` initializes a different Latex printer which uses a compact notation for time derivatives. For example:

```
t = symbols("t")
f = Function("f")(t)
f.diff(t, 2) + f.diff(t)
```

$$\dot{f} + \ddot{f} \tag{24.1}$$

Let's instantiate the `ReferenceFrame` class. To retrieve the basis vectors of a reference frame we call the `x, y, z` attributes:

```
N = me.ReferenceFrame("N")
N.x, N.y, N.z
```

$$\left(\hat{\mathbf{n}}_{\mathbf{x}},\ \hat{\mathbf{n}}_{\mathbf{y}},\ \hat{\mathbf{n}}_{\mathbf{z}}\right) \tag{24.2}$$

Typically, we use N to denote a Newtonian (or inertial) reference frame. We can also create and orient other reference frames. The most important methods to set an orientation are:

- `orient_axis`: sets the orientation a reference frame with respect to a parent reference frame by rotating through an angle about an axis fixed in the parent reference frame.
- `orient_body_fixed`: rotates a reference frame relative to the parent reference frame by three successive body fixed simple axis rotations (Euler angles).
- `orient_explicit`: sets the orientation of a reference frame relative to a parent reference frame by explicitly setting the direction cosine matrix.

All of them (and a few more) are well documented online. Let's create a new reference frame and rotate it through an angle θ about the $\hat{n}_z$ axis:

```
theta = symbols("theta")
A = me.ReferenceFrame("A")
A.orient_axis(N, N.z, theta)
A.dcm(N)
```

$$\begin{bmatrix} \cos(\theta) & \sin(\theta) & 0 \\ -\sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{24.3}$$

The above output shows the direction cosine matrix representing the orientation of A with respect to N. Since we are dealing with a right-handed reference frame, a positive angle θ corresponds to a rotation along the counter-clockwise direction. Let's now create a new reference frame rotated through an angle ϕ about the $\hat{a}_x$ axis:

```
phi = symbols("phi")
B = me.ReferenceFrame("B")
B.orient_axis(A, A.x, phi)
B.dcm(A), B.dcm(N)
```

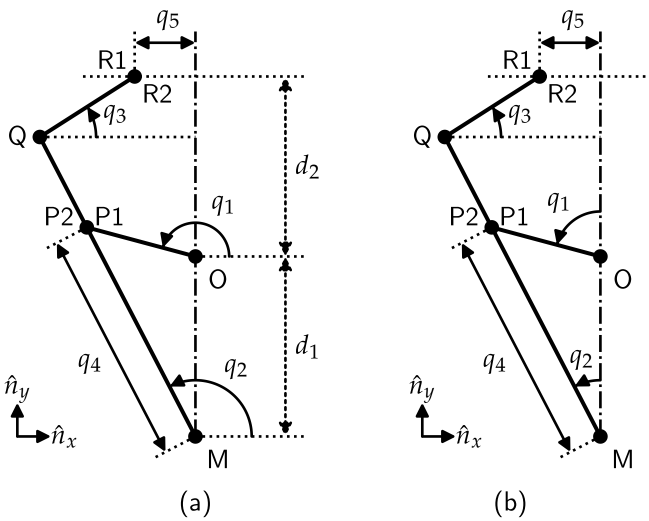

Figure 24.7: Schematic of the quick return mechanism. (a): the x-axis of each reference frame is aligned with the length of a rod. (b): the y-axis of each reference frame is aligned with the length of a rod.

```
# reinsert the previously removed values in order to visualize the system
sys.constants.update({d1: 8, d2: 8})

scene = Scene(N, O, crank_vf, rod_MQ_vf, rod_QR1_vf, slider_q4, slider_q5,
↪  times=sys.times)
scene.states_symbols = coordinates + speeds
scene.constants = sys.constants
scene.states_trajectories = results
scene.display_jupyter(axes_arrow_length=5)
```

24.7 Rolling Disk

A disk has six degrees of freedom in a three dimensional space: the position of the center of mass, $[x_G, y_G, z_G]$ and the yaw (ψ), tilt $(\theta)$and pitch (ϕ) angles (Figure 24.8). Considering a rolling disk without slipping, the system only has 5 degrees of freedom because the vertical position of the center of mass depends on the tilt angle: $z_G = r\cos\theta$. Then, for practical purposes, the following degrees of freedom are selected: $[x_{C_p}, y_{C_p}, \psi, \theta, \phi]$ where the first two are the coordinates of the contact point.

To describe the motion of the disk, 4 reference frames are introduced:

1. A: the ground fixed frame, where $\hat{a}_z$ is the vertical direction.
2. B: a reference frame rotated by the yaw angle ψ about $\hat{a}_z$, where $\hat{b}_x$ is tangential to the path of the contact point, C_p.

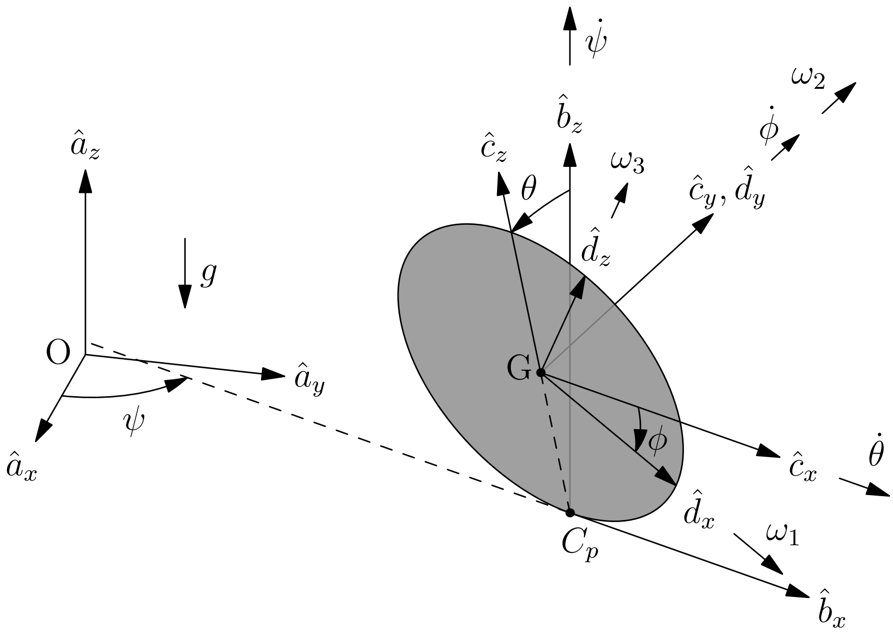

Figure 24.8: Rolling Disk schematic

3. C: a reference frame rotated by the tilt angle θ about $\hat{b}_x$, where $\hat{c}_y$ is aligned with the disk axle, whereas $\hat{c}_x$, $\hat{c}_z$ span the plane of the disk.
4. D: a reference frame associate to the disk, rotated by the pitch angle ϕ about $\hat{c}_y$.

We start by setting up the dynamical symbols:

```
psi, theta, phi, xc, yc = me.dynamicsymbols("psi, theta, phi, x_c, y_c")
omega1, omega2, omega3, uc, vc = me.dynamicsymbols(
    "omega1, omega2, omega3, u_c, v_c")
psid, thetad, phid, xcd, ycd = me.dynamicsymbols(
    "psi, theta, phi, x_c, y_c", 1)
m, r, g = symbols("m, r, g")
```

Here, `uc` and `vc` are the velocities of the contact point along the x and y directions in the ground frame. Then, we create the reference frames and compute the angular velocity of the disk with respect to the ground frame, expressed in the disk frame:

```
A, B, C, D = symbols("A, B, C, D", cls=me.ReferenceFrame)
B.orient_axis(A, A.z, psi)
C.orient_axis(B, B.x, theta)
D.orient_axis(C, C.y, phi)

A_w_D = D.ang_vel_in(A).express(D)
A_w_D
```

$$(-\sin(\phi)\cos(\theta)\dot{\psi} + \cos(\phi)\dot{\theta})\hat{\mathbf{d}}_{\mathbf{x}} + (\sin(\theta)\dot{\psi} + \dot{\phi})\hat{\mathbf{d}}_{\mathbf{y}} + \\ (\sin(\phi)\dot{\theta} + \cos(\phi)\cos(\theta)\dot{\psi})\hat{\mathbf{d}}_{\mathbf{z}} \tag{24.39}$$

This angular velocity can be used to formulate the rotational kinematic differential equations. Translational kinematic differential equations will be hard coded:

```
rot_kin = A_w_D - (omega1 * D.x + omega2 * D.y + omega3 * D.z)
rot_kin = rot_kin.to_matrix(D)
transl_kin = [xcd - uc, ycd - vc]
kdes = transl_kin + list(rot_kin)
display(*kdes)
```

$$\begin{gathered} -u_c + \dot{x}_c \\ -v_c + \dot{y}_c \\ -\omega_1 - \sin(\phi)\cos(\theta)\dot{\psi} + \cos(\phi)\dot{\theta} \\ -\omega_2 + \sin(\theta)\dot{\psi} + \dot{\phi} \\ -\omega_3 + \sin(\phi)\dot{\theta} + \cos(\phi)\cos(\theta)\dot{\psi} \end{gathered} \tag{24.40}$$

Let's setup the points and their velocities in the ground frame (where the EoM will be formulated):

```
O = me.Point("O")
Cp = O.locatenew("Cp", xc * A.x + yc * A.y)
G = Cp.locatenew("G", r * C.z)

O.set_vel(A, 0)
Cp.set_vel(A, uc * A.x + vc * A.y)
D.set_ang_vel(A, omega1 * D.x + omega2 * D.y + omega3 * D.z)
G.set_vel(A, D.ang_vel_in(A) ^ G.pos_from(Cp))
```

In the above code block, even though we already know the angular velocity of D with respect to A, we must set it again using the generalized speeds. By omitting that step, Kane's EoM will contain $\dot{\psi}$, $\dot{\theta}$, $\dot{\phi}$, which forces the numerical integrator to raise very hard to debug errors.

Now we have to define the nonholonomic constraints. The rolling without slipping condition is $\vec{v}_{C_p} = \vec{0}$, which gives two velocity constraints (the equal to zero part is implied):

```
vCO = Cp.vel(A) + (D.ang_vel_in(C) ^ Cp.pos_from(G))
fv = [vCO.dot(A.x), vCO.dot(A.y)]
display(*fv)
```

$$\begin{gathered} -r\cos(\psi)\dot{\phi} + u_c \\ -r\sin(\psi)\dot{\phi} + v_c \end{gathered} \tag{24.41}$$

Finally, we set up the rigid bodies (in this case, only the disk), the loads and construct the Kane's EoM. Because we have two nonholonomic constraints, we have three independent and two dependent generalized speeds: