Lowering barriers to using computational notebooks

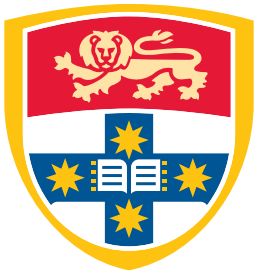
Tanaka Chitete
sid: 530222136

Supervisor: Dr. Rahul Gopinath

This thesis is submitted in partial fulfillment of
the requirements for the degree of
Bachelor of Science (Honours), Computer Science

School of Computer Science
The University of Sydney
Australia

6 November 2023



THE UNIVERSITY OF
SYDNEY

Compliance statement

I certify that:

I have read and understood the University of Sydney Student Plagiarism: Coursework Policy and Procedure;

I understand that failure to comply with the Student Plagiarism: Coursework Policy and Procedure can lead to the University commencing proceedings against me for potential student misconduct under Chapter 8 of the University of Sydney By-Law 1999 (as amended);

This Work is substantially my own, and to the extent that any part of this Work is not my own I have indicated that it is not my own by Acknowledging the Source of that part or those parts of the Work.

Name: Tanaka Chitete

Signature: Tanaka Chitete **Date**: 06/11/2023

Abstract

Computational notebooks have revolutionised how data scientists, researchers, and educators engage with scientific computing—combining executable code, rich-text documentation, and visualisations. As their popularity has surged, so too has the realisation that the state-of-the-art solutions cause numerous frustrations, pain points, affecting a myriad of workflows. Chattopadhyay et al. (2020) taxonimised nine general workflows, indicating that users face numerous pain points across all nine, from setup to deployment. Grounded upon this work, we present *Yuna*, a novel self-contained computational notebook *and* platform that aims to sooth multiple identified pain points. We discuss *Yuna*'s implementation details, ranging from the interface implementation to session management. We thereafter evaluate Yuna with respect to the state-of-the-art, illuminating *Yuna*'s potential to address *code management*, *reliability assurance*, *archival*, *dissemination*, and *reproducibility*. Finally, we gather insights into current limitations and explore future directions. *Yuna* posits a step forward in the evolution of computational notebook platforms, democratising access and lowering barriers to their use. Try *Yuna* at `https://tanaka-chitete.github.io/yuna/`.

Acknowledgements

To my supervisor, Dr. Rahul Gopinath, for plentiful contributions, chief of which being continually taking time out of your busy schedule to discuss my progress, keeping me on track.

To my friends, for providing solace in between days cooped up at home writing line after line of code—either to talk about life's problems or simply go for a walk.

To my parents, for continuing to put a roof over my head, even from the other side of the country.

Contents

List of figures

List of tables

Chapter 1

Introduction

1.1 Problems

In a comprehensive study, combining semi-structured interviews ($n = 20$) and confirmatory surveys ($n = 156$), Chattopadhyay et al. (2020) identified nine major pain points associated with using computational notebooks. Their findings indicate that users encounter numerous challenges at every stage, from setup to deployment, across several platforms. Table 1.1 personifies these frustrations (Chattopadhyay et al., 2020).

Setting up. Computational notebook users face problems as soon as they try to create a notebook. The process of importing and scrubbing data obtained from a variety of different sources often proves to be a clerical nightmare (Chattopadhyay et al., 2020).

Exploring and analysing data. In many platforms, exploration and analysis feels incomplete. Users often find themselves stuck in an infinite loop of copying and pasting code or attempting to tweak pre-existing code (Chattopadhyay et al., 2020). Stability issues such as kernel crashes and feedback latency complicate the matter further (Chattopadhyay et al., 2020).

Managing code. Despite representing the core workflow, writing and managing code is an ongoing frustration. Effective dependency management presents numerous issues for users without software engineering backgrounds. These users are plunged into a "dependency hell", relying on strewn-together workarounds (Chattopadhyay et al., 2020).

Ensuring reliability. Reliability problems are two-fold: notebook execution can be quite unreliable and scaling to big data is generally unsupported (Chattopadhyay et al., 2020). Scalability concerns cause endless kernel crashes which themselves cause irreparable data inconsistencies.

Loading data	Pulling data requires "going to seperate cloud instances to bring down the data locally, taking that to a file, and uploading it...".
Cleaning data	Efforts to scrub data are mostly clerical. There's "no mystery—it's just time consuming".
Modeling	"I need immediate feedback, like when I am testing slight changes in the model. I don't want to execute everything again".
Visualising	"Everything is in a cell, and the chart is limited by the boundaries and real estate of the notebook".
Iterating	The lack of code assistance is frustrating. You have to "go through the same ceremony to do even the most basic modelling task".
Writing code	"Anyone who tries to use notebooks has to start off with an IDE and then graduate into a notebook".
Managing dependencies	Managing packages and library dependencies within the notebook is, to put it bluntly, a "dependency hell".
Debugging code	It's a "horrible experience". "The only way to debug in most notebooks is through the use of `print` statements".
Testing	"There is no standard way to test notebooks". Different users "end up following different approaches".
Executing	Sometimes "it's easiest to just restart and run the whole notebook again".
Scaling	"The data is way too huge for notebooks to handle".
Versioning	There is "a lot of room for improvement when we want to check notebooks into source control".
Searching	Folders and files "become disorganised fast", making "finding and navigating to the intended file difficult".
Handling sensitive data	Adding security is time-consuming but handling desensitised data is dangerous—"if [you] screw up something, it can be leaked".
Controlling access	"The only way to share notebooks securely is to upload notebooks to secure team drives".
Sharing the notebook	It is common to present notebooks to other people. Despite this, sharing notebooks is "messy".
Sharing supporting artefacts	"Sharing...is kind of useless without the underlying data". Users also need to "match the environment settings of the notebook".
Editing collaboratively	"Collaborating in real-time with multiple users on a single notebook is super powerful". However, it is "not straightforward".
Reproducing	"The only way another person can run the notebook is if they're able to match all the environment settings".
Reusing and adapting	"...reuse can become more complicated than expected", like when "there are complex dependencies to bring into the new notebook".
Scheduling long-running computations	There's no feedback. "When the process is done" the notebook should "automatically create a notification".
Packaging as a standalone artefact	"It's very difficult and painful to transition notebooks to production code without doing a full rewrite".

Table 1.1: Pain points associated with computational notebook platforms.

The main culprit, a lack of processing power, induces failures when processing big data (Chattopadhyay et al., 2020).

Archiving. While many notebooks are typically used for quick-and-dirty work, others are authored with long-term usage in mind (Chattopadhyay et al., 2020). However, most platforms don't offer integrated versioning or search mechanisms (Chattopadhyay et al., 2020).

Securing. Mitigating leaks of sensitive data and controlling notebook access are core concerns (Chattopadhyay et al., 2020). With a widespread lack in native security measures, many users are relegated to implementing custom processes. However, custom implementations are often prone to errors, risking private data leaks and unauthorised access (Chattopadhyay et al., 2020).

Sharing and collaborating. Users seek to share notebooks at three different levels: *interactively*, *statically*, and as *read-only* files (Chattopadhyay et al., 2020). Interactive sharing entails the author executing their notebook in real time, static sharing involves the author disseminating a previously-executed notebook, and read-only documents merely allow recipients to leave comments. Notebook users also wish to collaborate remotely with real-time editing (Chattopadhyay et al., 2020). However, sharing at any of these three levels is generally unsupported (Chattopadhyay et al., 2020) and synced notebooks present coordination challenges (Wang et al., 2019).

Reproducing and reusing. Fundamental tasks such as marking student work or extending upon existing research hinge upon a notebook's reproducibility. Furthermore, users desire to adapt and reuse code from past notebooks in order to save time. Yet, reproducing computation or code is often unfeasible (Chattopadhyay et al., 2020). Notebook and machine dependencies are the main culprits (Chattopadhyay et al., 2020).

Deploying to production. Although notebooks are suited to quick exploration and analysis, they are difficult to deploy. Many features which enable rapid work, namely cell-based programming, represent an affront to software engineering best practices (Chattopadhyay et al., 2020). Deployment often requires significant code refactoring and dependency packaging—DevOps skills well-beyond the typical notebook user (Chattopadhyay et al., 2020).

1.2 Contributions

Our contributions are multi-fold:

(1) We author a compendium of knowledge not only pertinent to this dissertation, but the wider landscape of computational notebook platforms.

(2) We compile a 19-point chronology of the most significant computation platforms, from early progenitors to modern powerhouses, charting the evolution of this arena.
(3) We present *Yuna*, a fully self-contained computational notebook platform, with the ultimate goal of lowering barriers to entry. *Yuna* soothes the following pain points:
    - Managing code
    - Ensuring reliability
    - Archiving
    - Sharing and collaborating
    - Reproducing and reusing

    Try Yuna at `https://tanaka-chitete.github.io/yuna/`.
(4) We critically evaluate 13 state-of-the-art platforms, referencing all 9 pain points and their underlying activities, totalling 22.

1.3 Outline

We compose this dissertation as eight chapters:

- Chapter 2: We prescribe the background knowledge unfamiliar readers can call upon to comprehend this work.
- Chapter 3: We chronologise significant platforms, encompassing both historical and contemporary solutions.
- Chapter 4: We delve into *Yuna*'s implementation details.
- Chapter 5: We evaluate *Yuna*'s significance through comparison to the state-of-the-art.
- Chapter 6: We discuss the present limitations of our work and suggest avenues for further research.
- Chapter 7: We conclude this dissertation with a reflection upon our research.

Chapter 2

Background

A *computational notebook* is a document used to encapsulate code, narrative text, visualisations, and other rich media. This file is typically organised as a sequence of cells which contain either code or text. The code cells, when executed, display results in adjacent output cells, while text cells allow for the prescription of explanatory documentation. A *computational platform*, on the other hand, is the underlying system that runs the notebook, providing the computational engines and resources necessary to execute code and parse documentation. Figure 2.1 depicts the interface of such a platform, with an opened notebook.

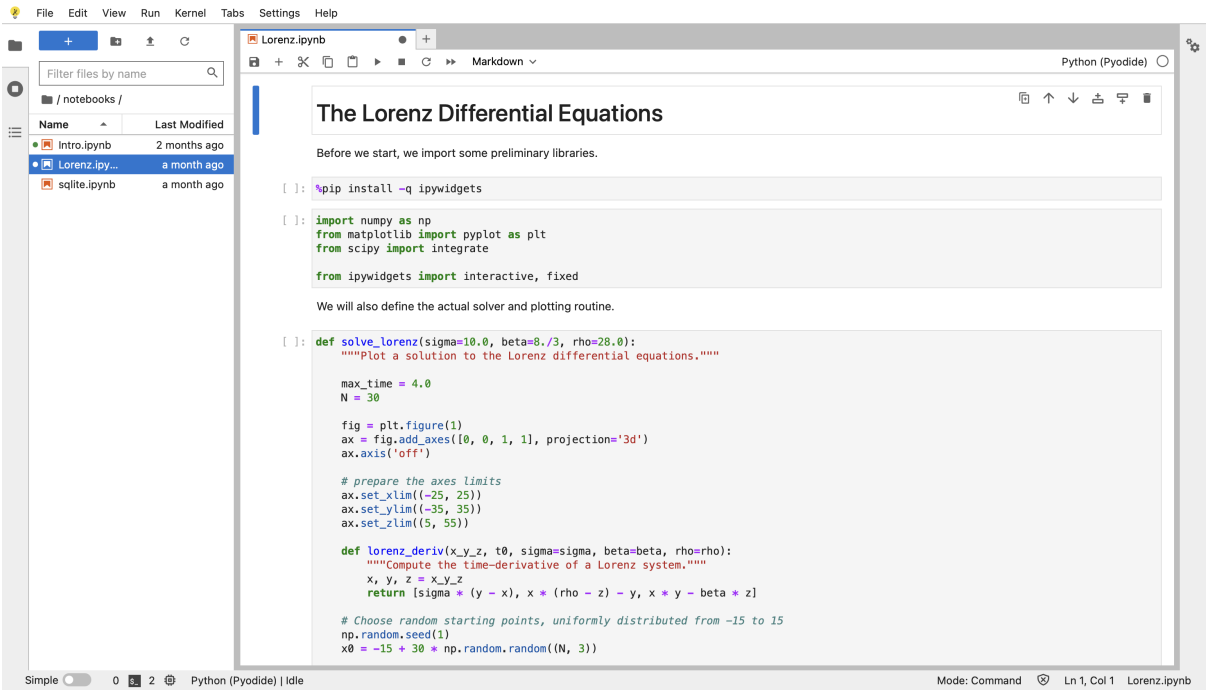


Figure 2.1: JupyterLab's interface.

Literate programming, conceptualised by Knuth (1984), lies at the foundation of the computational notebook. It is a paradigm which seeks to improve the readability of programs by considering them to be "works of literature" (Knuth, 1984). It authors a program as documentation of how it works in a human language, interspersed with executable snippets of code (Knuth, 1984). Consequently, the author's main task is not to instruct the computer, but rather explain to other people what they want the computer to do (Knuth, 1984).

The *interpreter* powers code execution, representing the main computational engine. It operates as follows:

(1) Receive code input from the user.
(2) Evaluate the commands contained within, storing the result.
(3) Return the result, for subsequent display in the interface.

While executing code, the interpreter might interact with additional artefacts: *data*, *packages*, and the *session*. Data pertains to the raw information to be explored or analysed, typically presented in tables, plain text, or images. Packages are bundles of related modules organised under a single name, offering highly-specialised functionalities which can be imported by the user to augment data processing efforts. The session is an abstract representation of the notebook's computational environment, encompassing all variables, functions, and imports.

The *parser* provisions rich-text documentation. Unlike plain text, rich text offers enhanced text formatting capabilities, including fonts, colours, and sizes. It also allows for the embedding of objects, such as images, tables, and files, allowing for greater flexibility in documentation. The parser operates as follows:

(1) Receive text input from the user.
(2) Translate the text into a displayable format, storing the translation.
(3) Return the translation, for subsequent display in the interface.

Synthesising this background, Figure 2.2 illustrates the components associated with a typical computational platform.

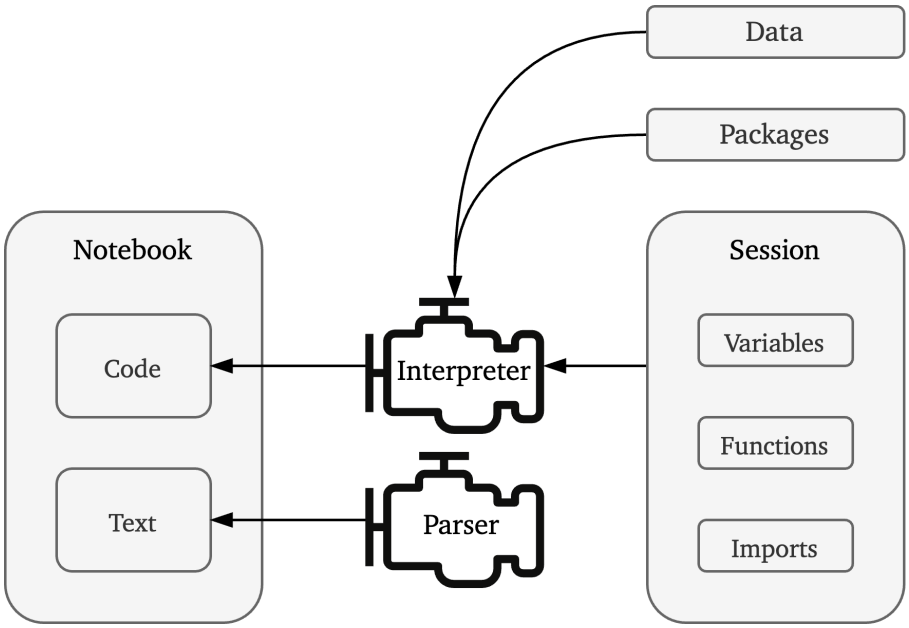


Figure 2.2: Components associated with a computational notebook platform.

Chapter 3

Chronology

3.1 VisiCalc

VisiCalc, an abbreviation of "Visible Calculator" (Byte, 1989), was the first spreadsheet program, pioneering the concept of virtual tables where users could input and manipulate data. Released in 1979 (Byte, 1989), it was designed to replicate traditional paper spreadsheets while providing the advantages of computational data manipulation.

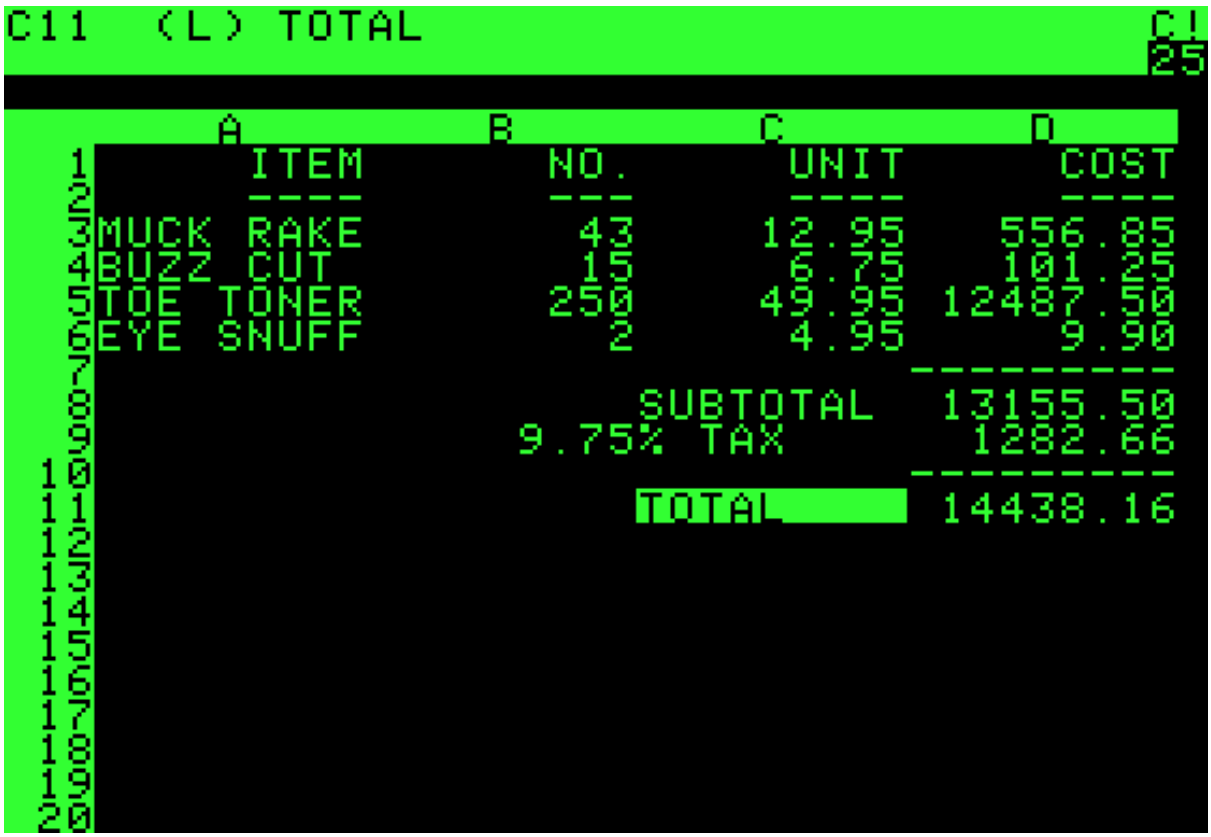


Figure 3.1: VisiCalc's interface.

Emerging during the nascent stages of personal computing in the late 1970s, VisiCalc was the brainchild of Dan Bricklin (Hormby, 2006). Developed with Bob Frankston under the banner of Software Arts (Gilpin and Purdum, 1985), the software initially graced the Apple II (Harford, 2019). As it ascended in significance, other platforms embraced it (Byte, 1989), solidifying its foundational status in the computational realm.

VisiCalc introduced a dynamic electronic grid where cells contained values, labels, or formulas. These formulas, recalculating cell values upon request, were revolutionary. The program's interface, depicted in Figure 3.1 (Gortu, 2005), is rudimentary by today's standards but was groundbreaking in its time. The tool's simplicity and specialisation contrasted it from subsequent, multi-functional computational notebooks.

After its release, VisiCalc quickly became the Apple II's "killer app" (Harford, 2019), with many purchasing the $2,000 USD computer, equivalent to $13,200 AUD in 2023, just to use the $100 USD software, equivalent to $700 AUD today (PC Mag, 1982). In 1996, Steve Jobs lauded VisiCalc, stating that it "propelled the Apple II to the success it achieved" (Bricklin, n.d.). While VisiCalc's innovative essence is undeniable, it struggled to remain competitive as computational needs diversified, culminating in its eventual eclipse in the mid-1980s by more sophisticated solutions (Microsoft, 2009). Despite this, VisiCalc cemented its legacy as the progenitor of modern spreadsheet software and laid the foundations for the landscape.

3.2 Macsyma

Macsyma, an acronym for Project MAC's "Symbolic Manipulator", stands as perhaps the earliest computer algebra system (CAS). Granted to the United States Department of Energy (DoE) in the early 1980s (Moses, 2012), the system performed symbolic manipulation of mathematical equations, augmenting the department's research efforts.

Rooted in the MIT Project MAC (Multiple Access Computer and Machine-Aided Cognition) initiative of the late 1960s, Macsyma emerged from the intersection of artificial intelligence and advanced mathematical research (Moses, 2012). Creators Carl Engelman, William A. Martin, and Joel Moses, alongside other key contributors, fashioned algorithms capable of symbolic differentiation and integration (Moses, 2012), filling a gap left by then-prevailing numerical solutions. While the program's initial focus was on symbolic computation, over time, it underwent

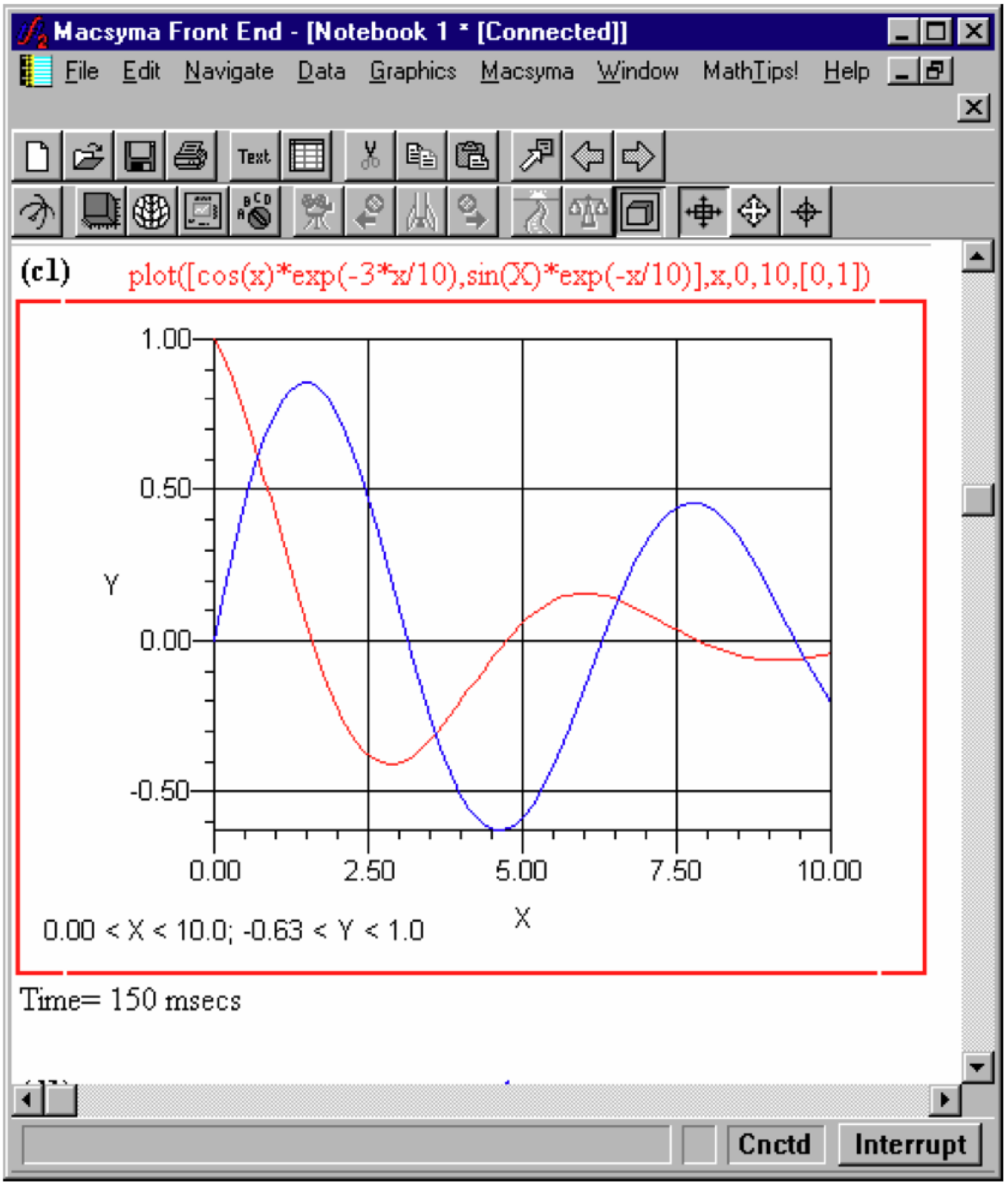


FIGURE 3.2: Macsyma's interface.

various adaptations and spawned several successors, such as Maple (Moses, 2012), ensuring its continuing influence in the computational arena.

Macsyma introduced groundbreaking features, such as exact arithmetic capabilities, allowing it to handle rational numbers without approximation (University of California, Berkley, 1996). It implemented Gröbner basis algorithms, pivotal for polynomial ideal operations (Moses, 2012). Its interactive command-line interface, depicted in Figure 3.2 (Macsyma, 1998), managed both algebraic and calculus tasks, using a unique syntax tailored for mathematical clarity. Macsyma also had extensive libraries dedicated to polynomial factorisation, pattern matching, and combinatorics (Maccone, 2012). The system's precise computation capabilities, coupled with its specialised library suite, distinguished it from counterparts.

Upon release, Macsyma's applicability quickly became apparent, with the system enjoying widespread use in across numerous research fields. By 1985, the system had been leveraged to explore problems in nuclear magnetic resonance, spectral analysis, emulsion chemistry, and close to 40 other areas (Pavelle and Wang, 1985). Though it faced competition from evolving computational tools, Macsyma's legacy is unassailable, influencing the next generation of systems.

3.3 Maple

Maple is a symbolic and numeric computational system, in addition to a multi-paradigm programming language. Debuted in 1982 (Walz, 1998), it was conceived to simplify complex symbolic and numeric computations.

Maple originated at the University of Waterloo in 1980 where, professors Keith Geddes and Gaston Gonnet, bonded over similar research interests and decided to design a CAS for research and education (Walz, 1998). Waterloo Maple, or Maplesoft, incorporated by the pair in 1988 (Symbolic Computation Group, 2012), continues to maintain the software, expanding Maple's purview. Through consistent development, the software matured, integrating functionalities like physics, statistics, and data analysis (Maplesoft, n.d.f,n).

Maple distinguishes itself through its advanced symbolic algebra capabilities. Its "2-D Math" mode replicates mathematical notation with accuracy (Maplesoft, n.d.a), allowing users to input equations as they appear on paper. It's interface is depicted in Figure 3.3 (Maplesoft, 2010). While its core strength lies in the Maple language, integration with MATLAB (Maplesoft, n.d.d) and connectivity to Python (Maplesoft, n.d.e) has been achieved, broadening its application horizons. Moreover, its extensive library, boasting hundreds of functions across topics ranging

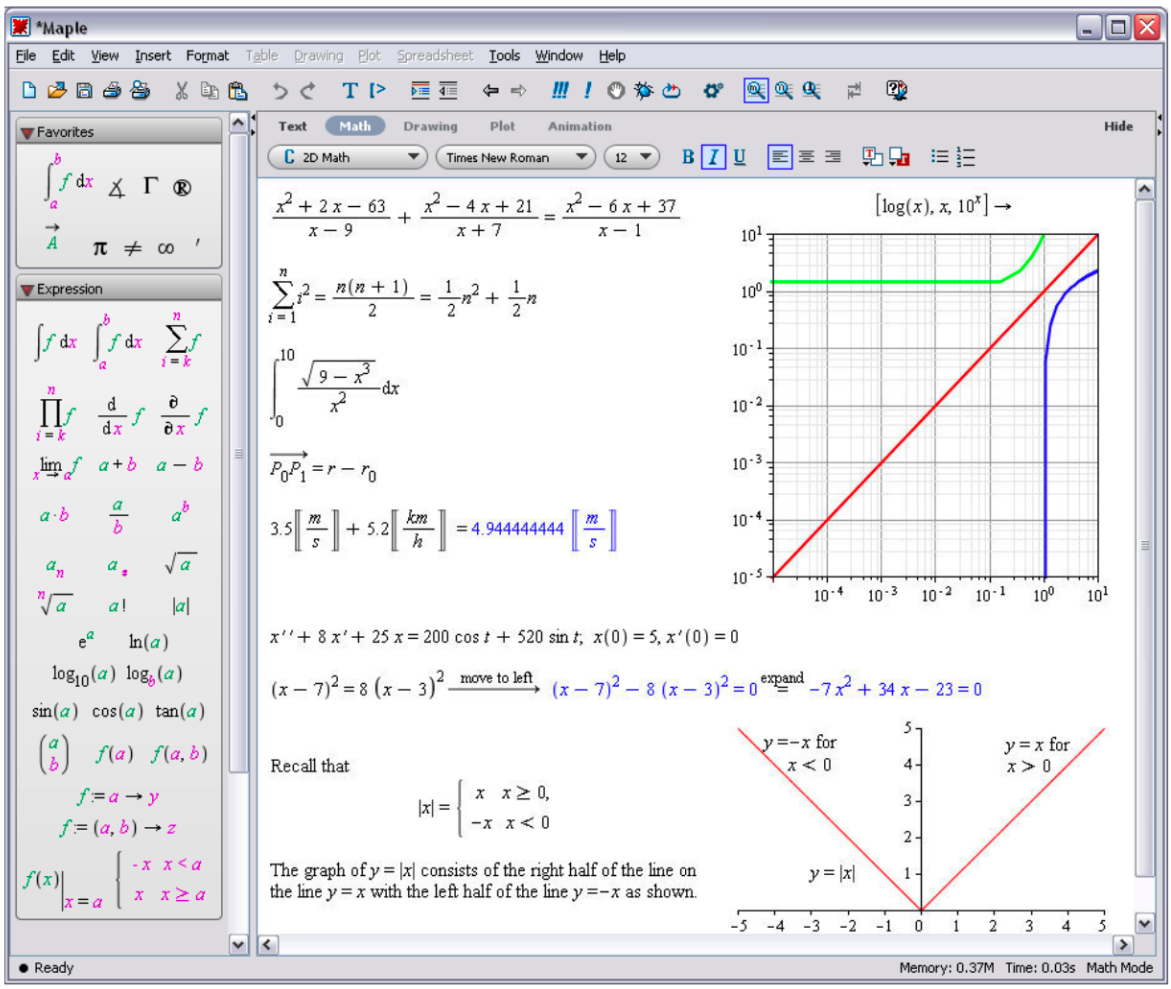


FIGURE 3.3: Maple's interface.

from signal processing (Maplesoft, n.d.g) to animation (Maplesoft, n.d.c), underscores its depth and comprehensive nature.

Commercially, Maple has enjoyed significant adoption by multiple high-profile organisations. Canon, renowned for its revolutionary cameras and lenses, used Maple and the broader Maplesoft platform in the production of its products (Maplesoft, n.d.b). Maple's contribution to computational mathematics is underlined by its enduring presence and continued adoption in numerous areas.

3.4 MATLAB

MATLAB, representing "Matrix Laboratory" (MathWorks, 2023g), is a numeric computation platform and multi-paradigm programming language (MathWorks, n.d.a). Introduced in 1984, it initially provided a specialised environment for manipulating matrices (Haigh, 2008), lowering barriers to executing complex computations.

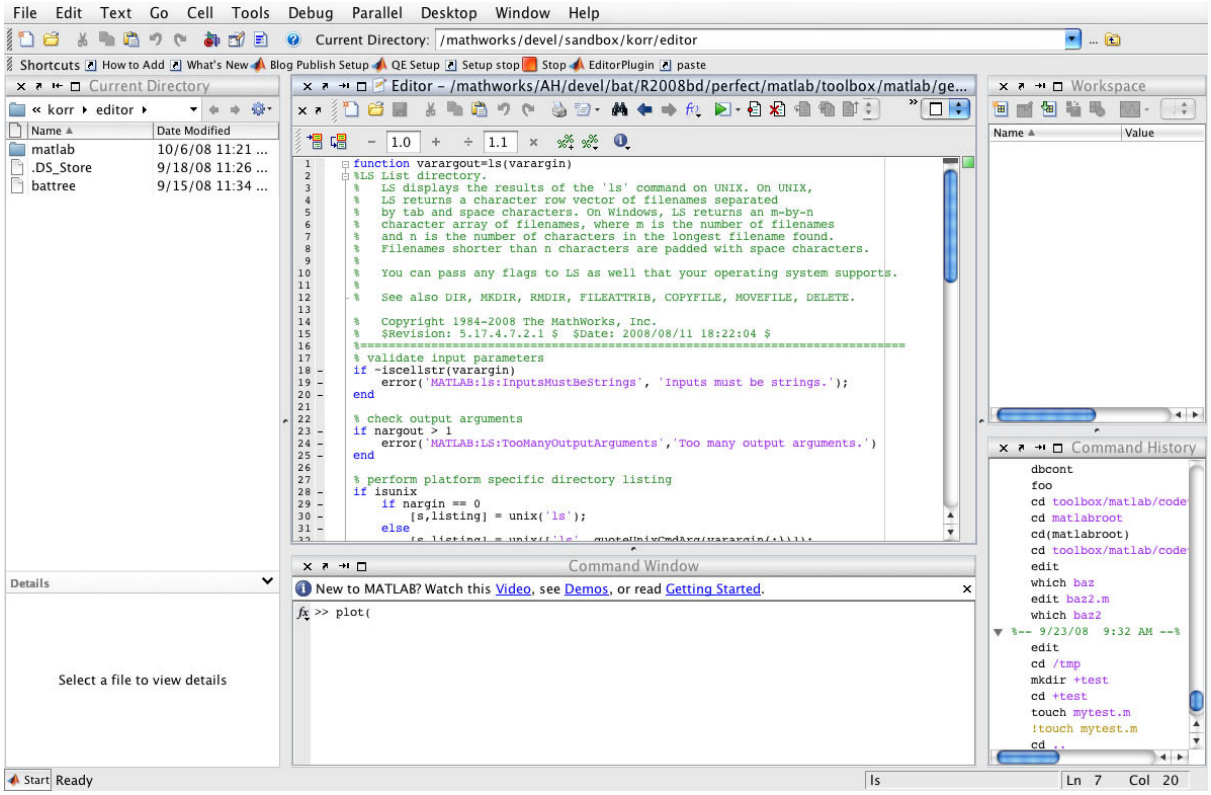


FIGURE 3.4: MATLAB's interface.

Conceptualised in creator Clive Moler's 1965 PhD thesis (Moler, 1965), MATLAB sought to simplify matrix operations by eliminating the need to learn FORTRAN directly (Chonacky and Winch, 2005). The subsequent establishment of MathWorks in 1984, co-founded by Clive Moler, Jack Little, and Steve Bangert (Chonacky and Winch, 2005; Haigh, 2008), marked a significant turning point, facilitating MATLAB's commercial expansion. With time, MATLAB evolved from its roots as a simple matrix calculator to become a comprehensive platform, serving diverse applications.

The user interface comprises a command window, workspace, and live editor, all of which are shown in Figure 3.4 (MATLAB, 2008). MATLAB supports a plethora of toolboxes, catering to

varied disciplines including machine learning (MathWorks, 2023e), signal processing (MathWorks, 2023d), and control systems (MathWorks, 2023c). Additionally, its ability to interface with other languages, notably Java (MathWorks, 2023f), C/C++ (MathWorks, 2023a), and FORTRAN (MathWorks, 2023b), accentuates its adaptability and cross-platform utility.

MATLAB's versatile toolboxes have been deployed to address complex domain-specific challenges. Toyota employed MATLAB and the broader MathWorks platform for "simulation, data processing, and controls design" (MathWorks, n.d.b). Its profound impact transcends mere computation, MATLAB has fostered a large community, expediting discovery and innovation. Consequently, MATLAB's enduring influence lies not just in its computational prowess, but in uniting diverse disciplines under a common computational umbrella.

3.5 Mathcad

Mathcad is an interactive platform for engineering calculation, documentation, and visualisation (PTC, n.d.f). Brought to market in 1987 (PC Mag, 1987), its primary aim was to offer an integrated platform for mathematical notation, text, and graphics (PC Mag, 1987). It was the first commercial system to introduce live editing (PC Mag, 1987).

Mathcad's creation was instigated by the concept of a tool that would bring "what you see is what you get" (WYSIWYG) capabilities to mathematical computations. In 2016, to further develop Mathcad, the system was brought under the umbrella of Parametric Technology Corporation (Digital Engineering, 2006). Since its inception, the platform has undergone several updates, incorporating data analysis (PTC, n.d.b), image processing (PTC, n.d.e), and finance (PTC, n.d.c), preserving its relevance in a rapidly evolving landscape.

Mathcad distinguished itself with its pioneering WYSIWYG interface. Inline with this essence, Mathcad's "natural math notation" allows users to input equations just as they would write them by hand (PTC, n.d.h), enhancing clarity and usability. Mathcad's interface is pictured in Figure 3.5 (Abelniak, 2011). Integrated symbolic computation allows users to perform algebraic simplifications and manipulations seamlessly (PTC, n.d.g). Moreover, Mathcad integrates with C++, C#, VB, VBScript, and JavaScript (PTC, n.d.a), offering enhanced computational capabilities. Mathcad's rich library, coupled with its numerous language integrations, bolsters its adaptability across disciplines.

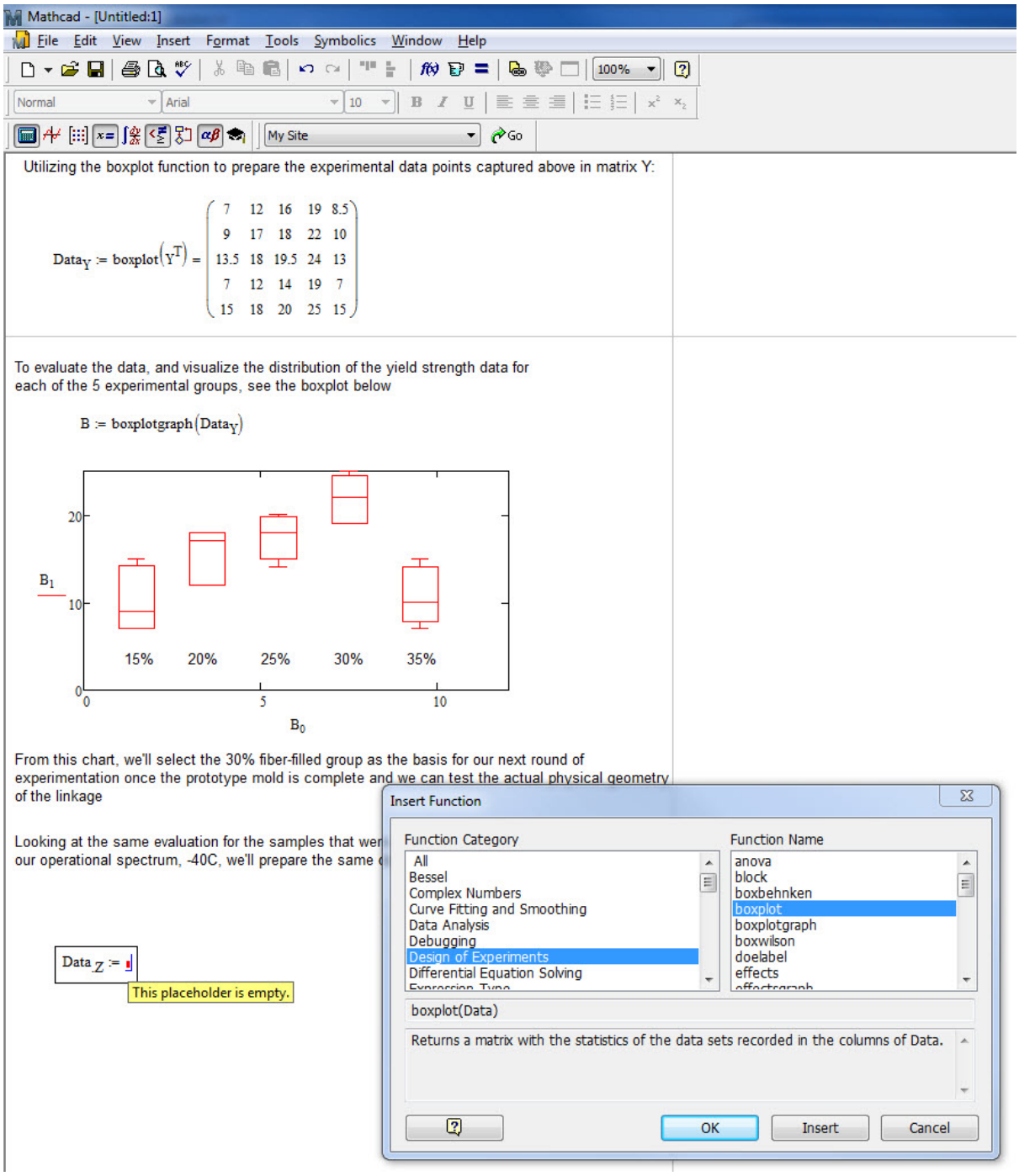


FIGURE 3.5: Mathcad's interface.

Honeywell, a multinational conglomerate operating in sectors such as commercial real estate and aerospace (Honeywell, n.d.), used Mathcad in the training and mentoring of newly-hired engineers (PTC, n.d.d). Honeywell left a lasting mark, transforming human interaction with

computing systems, ultimately influencing how subsequent computational notebooks married documentation and live computation.

3.6 Mathematica

Mathematica is a computational platform that integrates a symbolic mathematics engine and high-level programming language with a far-reaching native library (Wolfram, n.d.h). Unveiled in 1988 (Wolfram, 2008), it was developed to serve as a comprehensive system for various technical fields spanning science, engineering, and mathematics (Wolfram, 2008).

The system emerged from the personal computing revolution of the 1980s, where diminishing production costs meant that computers could find their way into peoples' homes (Computer History Museum, n.d.). Creator, Stephen Wolfram, sought to capitalise. With the foundation Wolfram Research, Mathematica 1.0 was rolled out to various Macintosh and Unix computers (Wolfram, 2013). Over the decades, Mathematica evolved, incorporating advancements like image processing (Wolfram, n.d.c) and machine learning (Wolfram, n.d.e).

Through `Manipulate` (Wolfram, n.d.f) and other dynamic constructs, users can create interactive interfaces in the notebook. Figure 3.6 (HolyCookie, 2011) depicts the standard user interface. While built primarily for the Wolfram Language, Mathematica is interoperable with variety of other programming languages, including R (Wolfram, n.d.g), Java (Wolfram, n.d.d), and C/C++ (Wolfram, n.d.b). A rich collection of built-in packages, unique user interface, and multi-language support position Mathematica a versatile tool in computational research and analysis.

EQA Partners, a global hedge fund, utilised Mathematica and the wider Wolfram ecosystem to support an enterprise-wide business logic, from market data ingestion to investor report preparation (Wolfram, n.d.a). Mathematica's legacy is dual-faceted: not only did the platform enhance symbolic and numeric computation, it also redefined the landscape of computational exploration, influencing successive solutions.

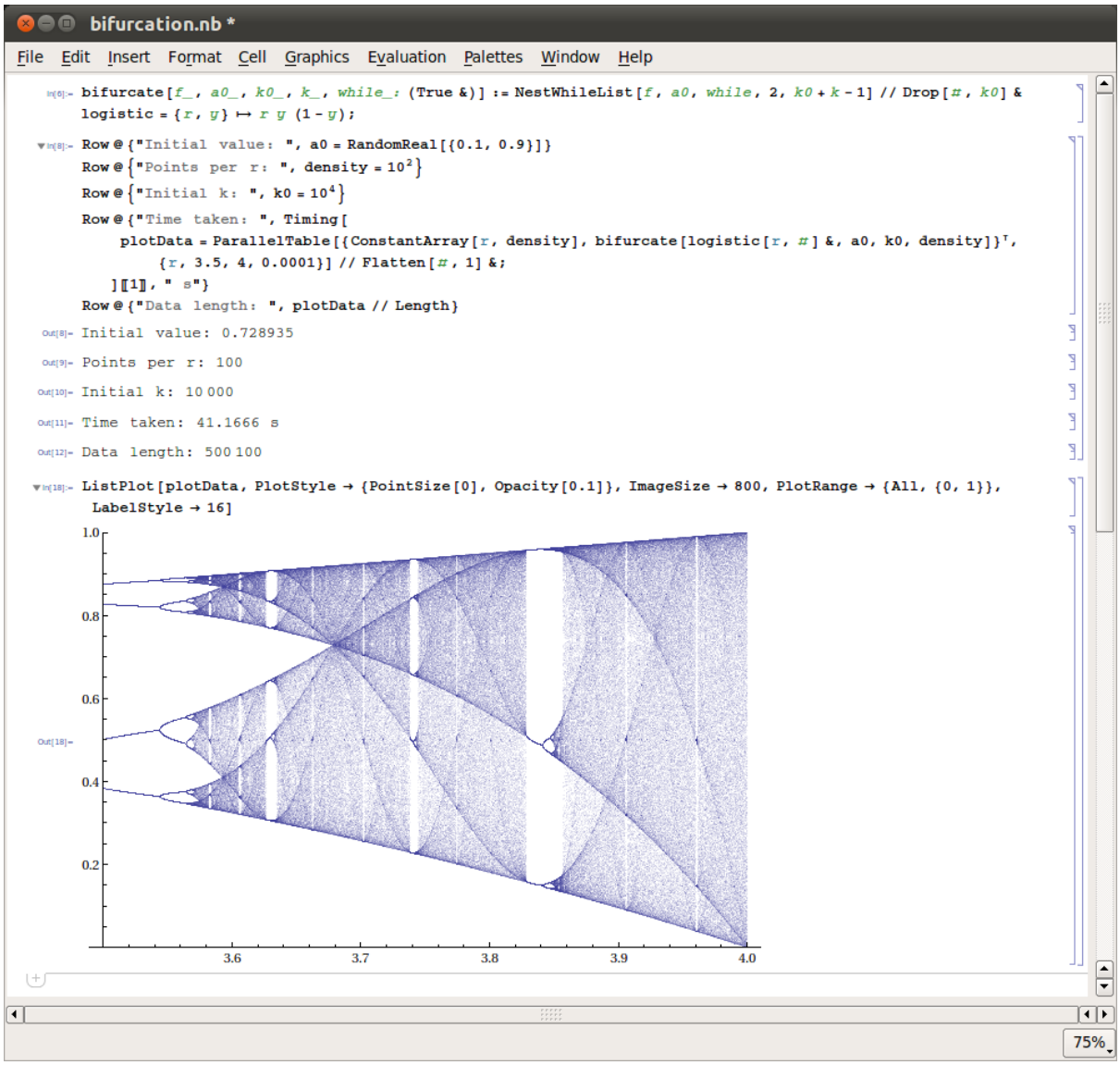


FIGURE 3.6: Mathematica's interface.

3.7 Maxima

Maxima, emerging as a derivative of Macsyma, is an open-source CAS (Maxima, n.d.a). Launched in 1998, it primarily aimed to extend symbolic computation to a wider audience, breaking the commercial constraints associated with its predecessor (Maxima, n.d.a).

With permission from the DoE, William Schelter, long-standing maintainer of the software, released Macsyma source code under the GNU public license in 1998 (Maxima, n.d.a). Prior to his death in 2001, he founded the Maxima project to ensure Maxima's survival (Maxima, n.d.a).

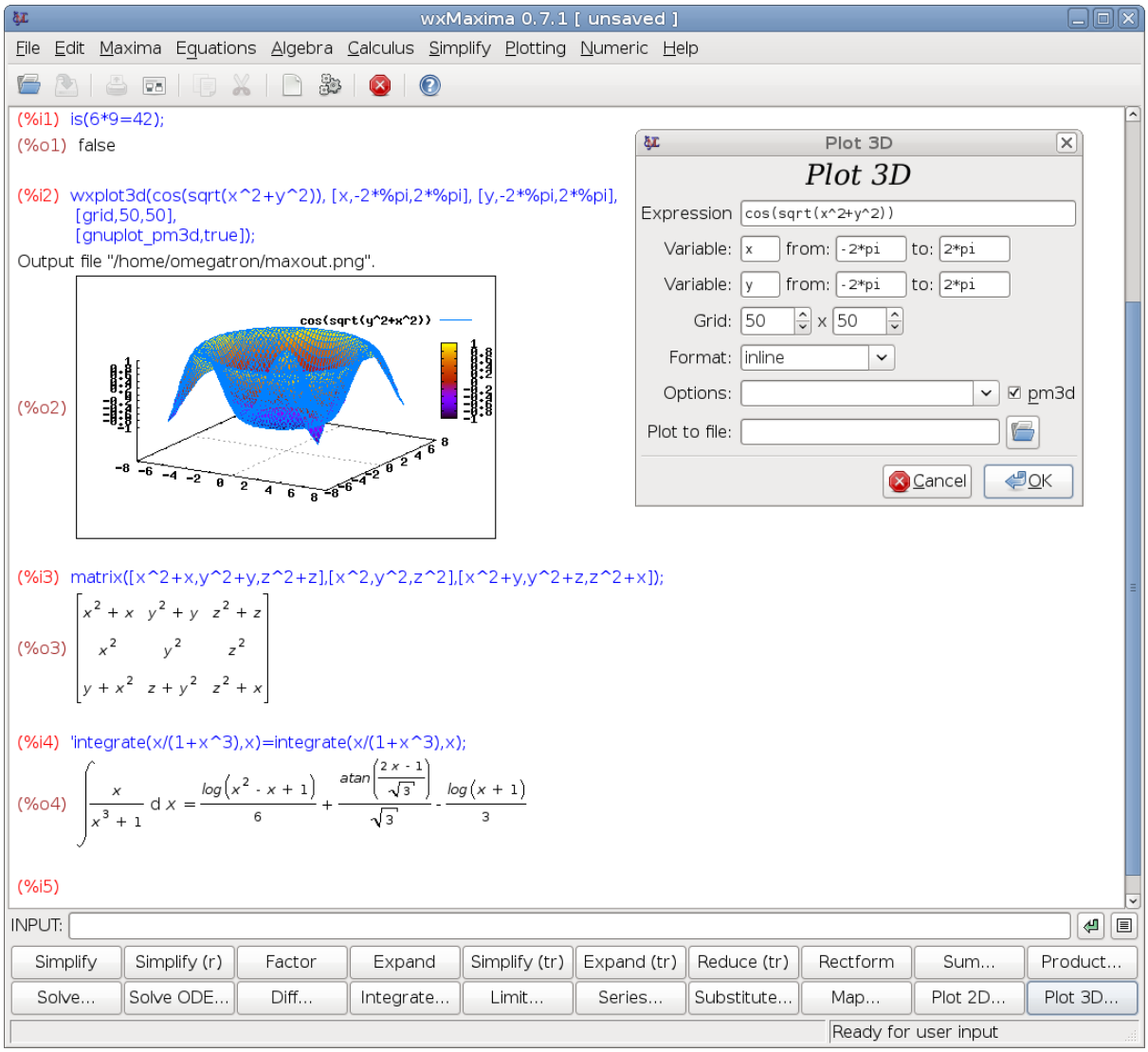


FIGURE 3.7: Maxima's interface.

Over the years, the platform has been enhanced with modern packages supporting numerous advanced functionalities.

wxMaxima, a GUI for Maxima (andrejv and peterpall, 2019), shown in Figure 3.7 (Omegatron, 2008), supports visualisation. The platform also offers 3D plotting (Maxima, n.d.c), finance (Maxima, n.d.d), probability (Maxima, n.d.b), and Maxima to Python translation (Maxima, n.d.e). Furthermore, since Maxima is implemented using Lisp, it provides users with an easy means to interface with the language, simply prefacing commands with `lisp` (Maxima, n.d.f). Maxima's unique blend of ancient Macsyma origins with contemporary functionalities differentiates it in the sphere of computational platforms.

The International Neuroinformatics Coordinating Facility (INCF) (INCF, n.d.), an international non-profit organisation, utilised Maxima to advance their neuroscientific research efforts during the 2019 Google Summer of Code (Belgian Neuroinformatics, 2019). Maxima's lasting impact lies in democratising access to symbolic mathematics, expanding Macsyma's purview while continually adapting to evolving demands, 20 years since its inception.

3.8 IPython Notebook

IPython Notebook, precursor to Jupyter Notebook (Perez, 2014), was a pivotal scientific computing platform. Launched in 2011, it was fashioned to elevate the IPython shell experience by bringing live code execution, narrative text, and visualisations to the web (Vu, 2021).

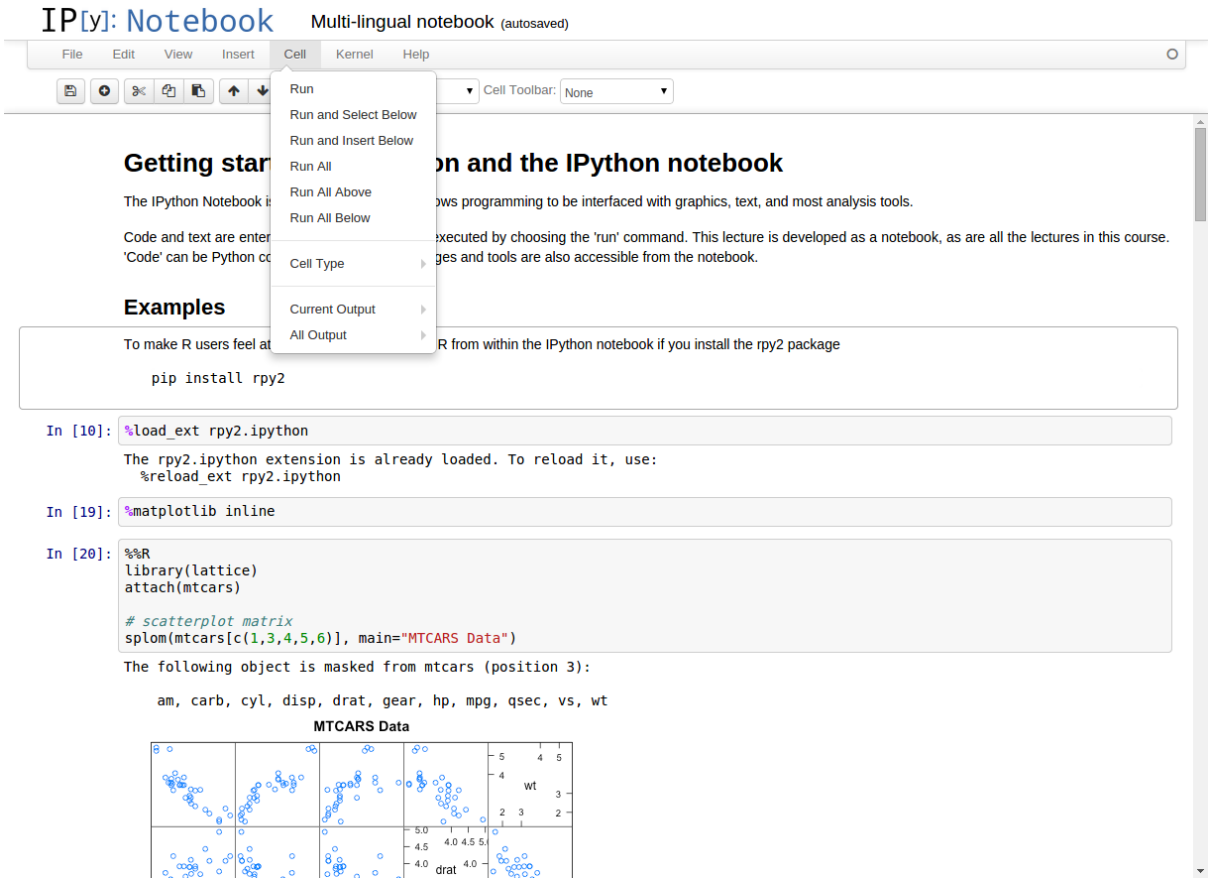


FIGURE 3.8: IPython Notebook's interface.

IPython, the interactive shell backing IPython Notebook, was manifested in 2001 by Fernando Perez' desire to use Python code in data-based workflows, iterating between writing code and

exploring data (Perez, 2012; Vu, 2021). Through collaboration with Brian Granger and Min Ragan-Kelley, numerous notebook prototypes would be conceived, leading with the release of IPython Notebook in 2011 (Vu, 2021). Over time, the platform would undergo significant refinements, culminating with the metamorphosis into Jupyter Notebook in 2014 (Perez, 2012).

Its user interface, shown in Figure 3.8 (Duke University, n.d.), was browser-based, ensuring platform independence and accessibility. While solely designed for Python, IPython Notebook maintained flexibility through integration with popular Python libraries such as pandas (pandas, n.d.), NumPy (NumPy, n.d.), and Matplotlib (Matplotlib, n.d.). Furthermore, "magic" commands could be employed to augment kernel functionality (IPython, 2023). The totality of these capabilities made IPython Notebook a powerful tool for data analysis and visualisation.

Recognising IPython Notebook's prowess for data analysis and pedagogical endeavours, renowned institutions, like the University of California, Berkeley, provisioned IPython Notebook to support the learning of its students (Manning, 2016). IPython Notebook not only further democratised scientific computing, it also paved the way for the evolution of the Jupyter ecosystem (Perez, 2014), cementing its position as a cornerstone in the modern landscape of computational notebook platforms.

3.9 RStudio

RStudio is an open-source integrated development environment (IDE) for the R programming language (Posit Software, n.d.). Announced in 2011, the IDE was fashioned to provide a streamlined environment to support R-based workflows (RStudio Team, 2011).

The genesis of RStudio can be attributed to a desire for a more comprehensive and user-friendly interface for R, as no native solution existed at the time. RStudio Inc (now Posit), founded in 2009 by Joseph J. Allaire, was instrumental in addressing this need (Posit, 2023a). Over the years, RStudio underwent continuous refinement, incorporating features like support for Markdown (RStudio, n.d.c), Shiny applications (RStudio, n.d.a), and integration with version control systems (Posit, 2023b). The software, originally a singular desktop application, expanded to encompass a hosted counterpart, Posit Cloud (Posit, 2023a), bolstering accessibility and collaborative capabilities.

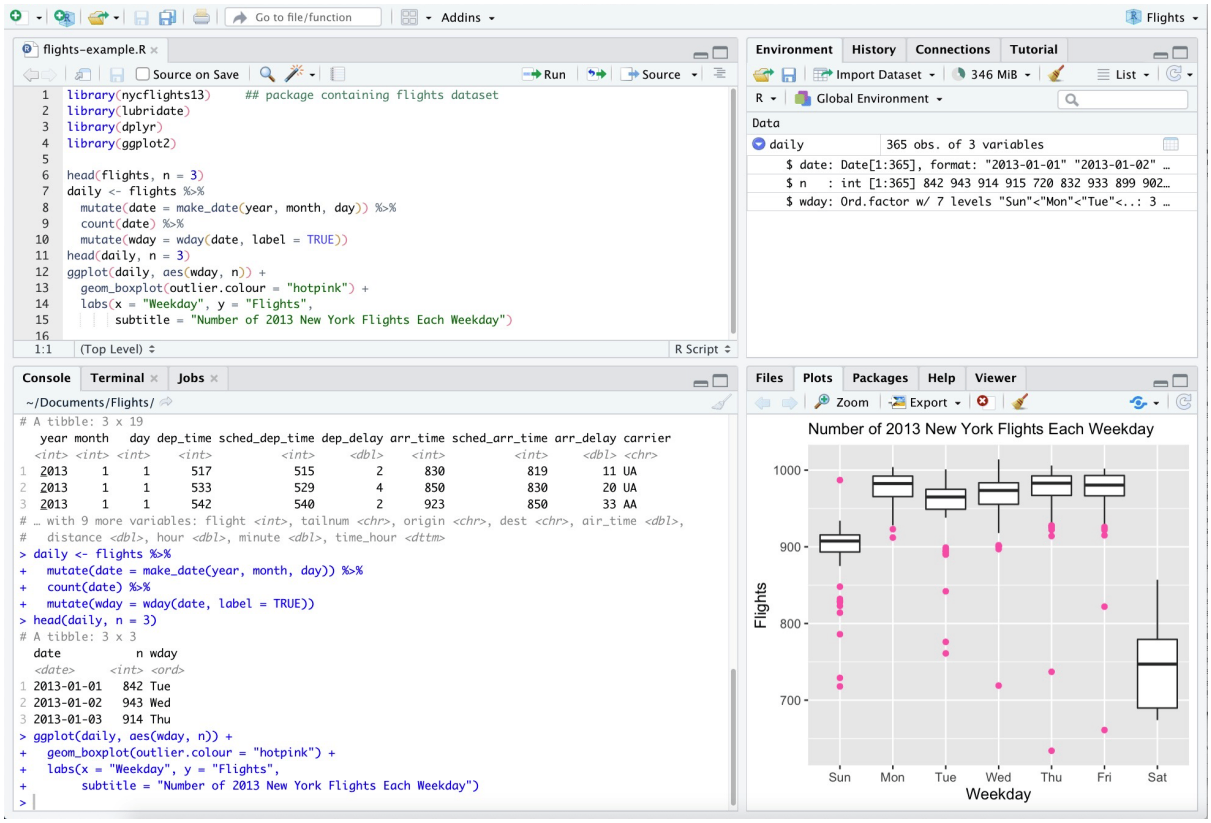


FIGURE 3.9: RStudio's interface.

RStudio stands out with its seamless integration of R tools in one platform, including a console, script editor, plot viewer, and R package manager, depicted in Figure 3.9 (cdhowe, 2021). The interface eases the R programming workflow with features like code-completion and syntax highlighting (Ushey, 2023). Additionally, RStudio supports multiple libraries within the R ecosystem (RStudio, n.d.d) and even offers an interface for Python (RStudio, n.d.b), highlighting its adaptability to diverse computational needs.

Attesting to the platform's impact, Centers of Disease Control and Prevention recently leveraged R to analyse trends in COVID-19 deaths (Centers for Disease Control and Prevention, 2020). RStudio's lasting impact rests in its pivotal role in making R more accessible, fostering a broader adoption of the language and underpinning critical applications with its numerous advancements.

3.10 Jupyter Notebook

Jupyter Notebook, the successor to IPython Notebook, supports various programming languages, namely Julia, Python, and R (hence the name "Ju-Pyt-er") (Perez, 2014). Introduced in early 2015, its vision expanded to offer a versatile platform for reproducible science across numerous languages (Perez, 2014).

FIGURE 3.10: Jupyter Notebook's interface.

Jupyter Notebook's inception was driven by the aspiration to provision a language-agnostic platform for scientific computing. It was the brainchild of the Project Jupyter team, an initiative led by key figures including Fernando Pérez and Brian Granger (Perez, 2014). Over the years, Project Jupyter had undergone myriad enhancements, evolving into an ecosystem encompassing various platforms, like JupyterHub and JupyterLab (Project Jupyter, n.d.b).

Jupyter Notebook distinguishes itself with its kernel-based approach, facilitating simultaneous execution environments for multiple programming languages. The browser-based user interface, illustrated in Figure 3.10 (Makeability Lab, n.d.) merges code execution, rich text rendering, and multimedia illustration. Central to its prowess is its support for over 40 programming languages

(Project Jupyter, n.d.b). Considering Python alone, Jupyter integrates with prominent packages, including Matplotlib (Matplotlib, n.d.), NumPy (NumPy, n.d.), and pandas (pandas, n.d.), reinforcing its position as a versatile platform in scientific and data-driven arenas.

538 (538, n.d.), a data-centric news outlet, continues to employ Jupyter Notebooks for investigative journalism (538, 2023), communicating complex data stories with great efficacy. Jupyter's unparalleled success is a testament to its ability to bridge the gap between theory and practice. Moreover, it propelled the computational notebook paradigm forward, shaping the future trajectory of reproducible research and collaborative computing.

3.11 Databricks

Databricks is a cloud-based data platform that streamlines big data processing and machine learning (Databricks, n.d.f). Founded in 2013 (Databricks, n.d.d), it was conceived to simplify big data analytics through Apache Spark with an emphasis on assisting data teams in collaborative endeavours (Databricks, 2023a).

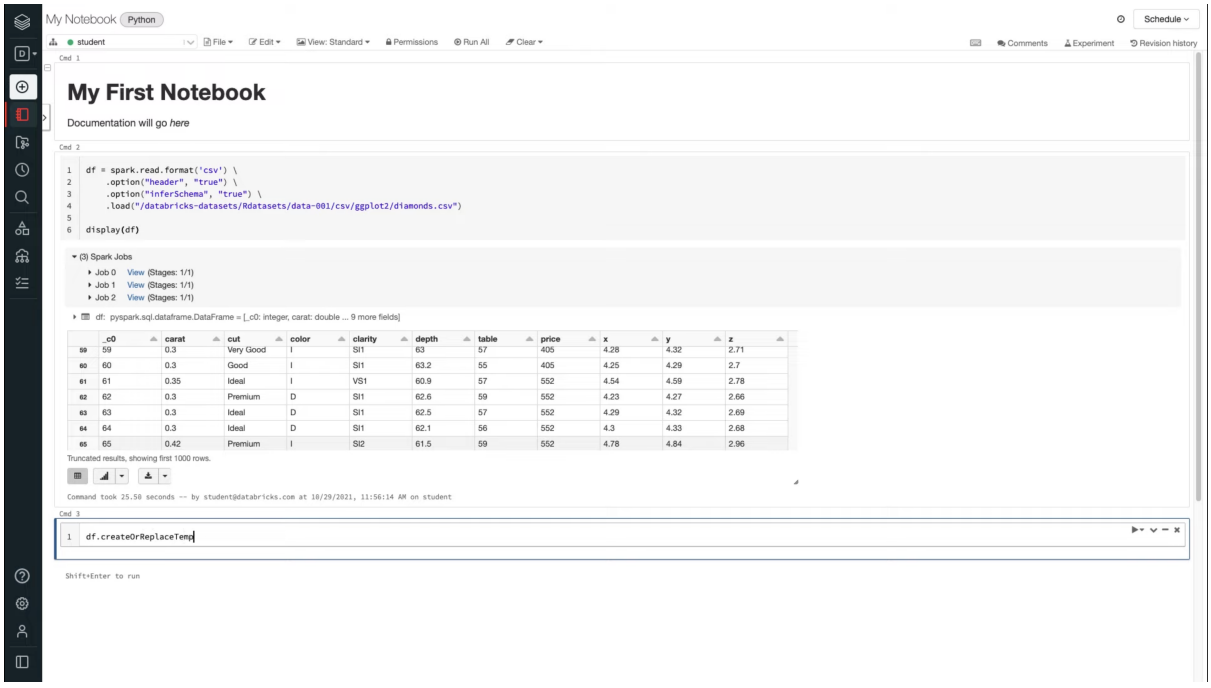


Figure 3.11: Databricks' interface.

Spawned from the pioneering work on Apache Spark at University of California, Berkeley's AMPLab (Apache Spark, n.d.), Databricks was co-founded by the creators of Spark (Databricks,

n.d.d). As the landscape of data analytics matured, Databricks continually adapted, enhancing its core Spark capabilities and integrating solutions for data streaming (Databricks, n.d.c), and AI/ML (Databricks, n.d.a). Its evolution, driven by both market demands and the vision of its founders, has solidified its position as a key player in big data analytics.

Databricks distinguishes itself by offering a unified analytics platform, integrating batch and streaming workloads seamlessly (Aashita, 2023). The user interface, shown in Figure 3.11 (Databricks, 2022), supports multi-language notebooks, catering to Python (Databricks, 2023b), Scala (Databricks, 2023d), R (Databricks, 2023c), and SQL (Databricks, 2023g). Delta Lake, a foundational feature, offers ACID transactions, versioning, and schema enforcement, ensuring data reliability (Databricks, 2023h). Moreover, Databricks boasts built-in integrations with MLflow (Databricks, 2023e) and TensorBoard (Databricks, 2-23), addressing the full machine learning lifecycle from experimentation to deployment.

Databricks has been instrumental in expediting data-driven decisions for myriad enterprises. For instance, Comcast harnessed Databricks for real-time analytics, efficiently processing billions of events per day to enhance user experiences (Databricks, n.d.b). Databricks' impact is manifold: it has not only furthered cloud-based big data analytics but has also fostered collaboration among data scientists, engineers, and their collaborators. The integration of advanced functionalities within a unified platform showcases its visionary approach.

3.12 Kaggle

Kaggle, through its notebooks feature, (Kaggle, n.d.a) provides a cloud-based Python environment to write, modify, and execute code directly on the platform. Rolled out in the mid-2010s, Kaggle implemented notebooks to bolster data science and machine learning competitions.

Emerging from Kaggle's burgeoning ecosystem, Notebooks was a strategic response to the perceived need for a tool that enabled seamless exploration of datasets within the platform. Driven by Kaggle's dedicated team under the umbrella of Google Cloud (Li, 2017), the notebooks capability experienced refinements over time, integrating competitions, datasets, and discussions (Kaggle, n.d.a).

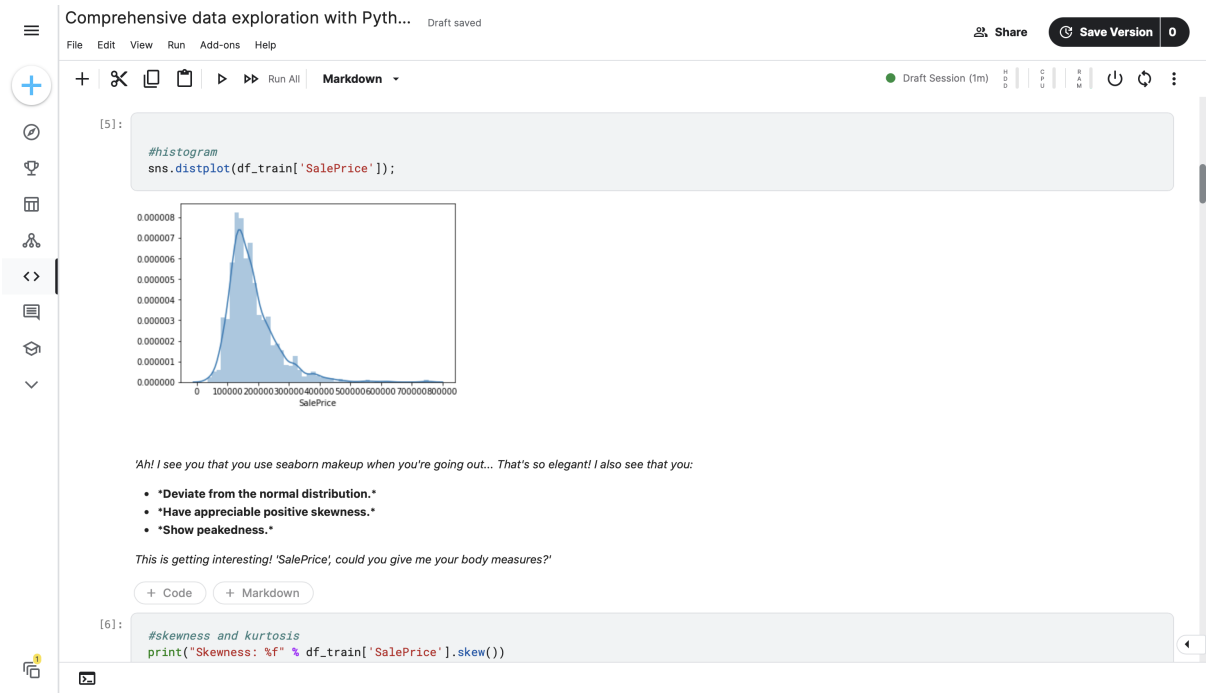


Figure 3.12: Kaggle's interface.

The environment is versatile, providing out-of-the-box support for Python and R (Kaggle, n.d.a), in addition to allowing augmentation with custom libraries (Kaggle, n.d.a). A sample notebook is depicted in Figure 3.12 (Marcelino, 2022). The environment is outfitted with prominent data science tools, such as TensorFlow and Keras (Kaggle, n.d.c). GPUs and TPUs, integral for deep learning projects, are available at users' disposal (Kaggle, n.d.b), supporting highly-performant code execution.

Notebooks on Kaggle have been involved in numerous high-stakes data science competitions, enabling participants to swiftly prototype and iterate on models. The European Space Agency utilised the platform in 2017 for their Mars Express Power Challenge (European Space Agency, 2017). The platform has democratised access to advanced computational resources and cultivated an ethos of open collaboration, reshaping how enthusiasts learn, compete, and innovate.

3.13 Azure Machine Learning

Azure Machine Learning is a cloud-based service that offers a suite of tools and frameworks for the entire ML lifecycle (Microsoft, n.d.a). The platform was launched in 2015 to provide a

comprehensive platform for building, training, and deploying machine learning models (Miller, 2015).

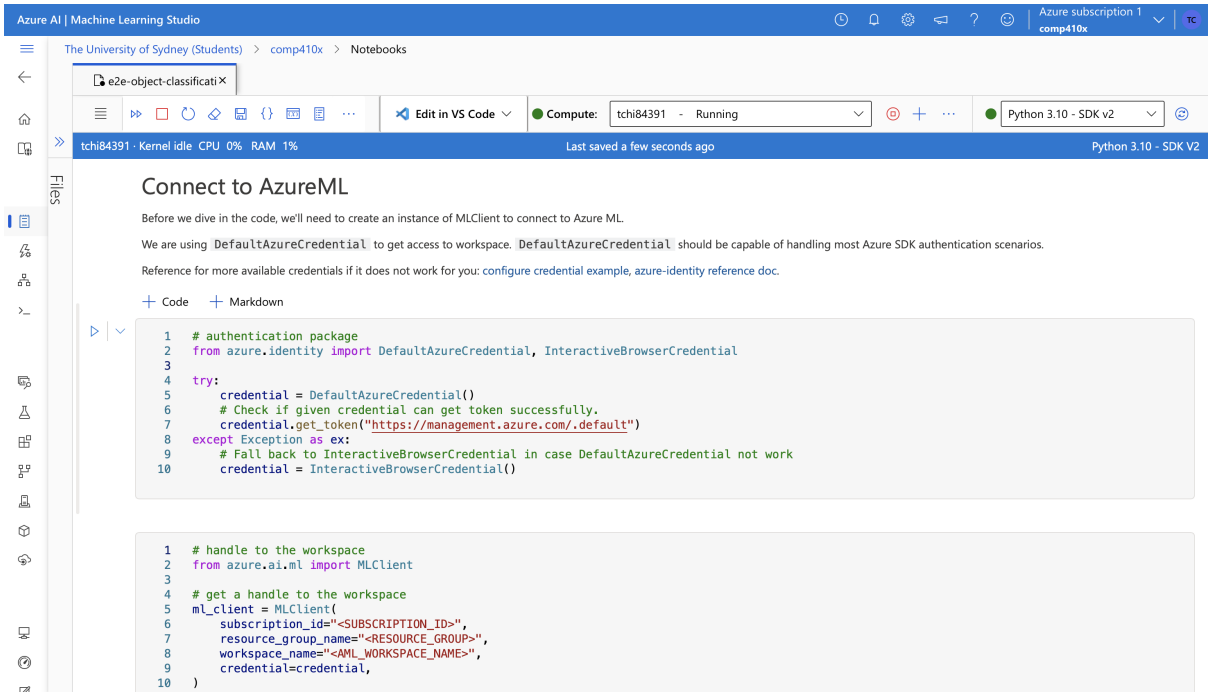


Figure 3.13: Azure Machine Learning's interface.

Azure Machine Learning's conception was embedded in Microsoft's vision of democratising artificial intelligence and making machine learning accessible to a broader audience. It emerged from Microsoft's research in cloud-based data analysis (Microsoft, 2013). Led by Microsoft's Azure division, the service has seen significant contributions since its inception, including real-time collaboration, data preparation, and automated machine learning (Microsoft, n.d.a). Such features have ensured the platform's relevancy in the ever-changing landscape of machine learning platforms.

The service boasts unique functionalities such "drag-and-drop" designing (Microsoft, n.d.a), enabling swift model development without necessitating deep programming expertise. The user interface, depicted in Figure 3.13, natively integrates with Azure's broader cloud ecosystem. With Python at the centre of the platform, it fosters a diverse set of libraries and frameworks, including TensorFlow (Microsoft, 2023d), scikit-learn (Microsoft, 2023c), and PyTorch (Microsoft, 2023b). This broad integration ensures practitioners can employ a wide array of tools while benefiting from Azure's scalable infrastructure and deployment capabilities.

Azure Machine Learning and the wider Azure platform and has found adoption by more than 95% of Fortune 500 companies (Microsoft Azure, 2018). For example, Progressive Insurance leveraged Azure Machine Learning in 2017 to implement Flo, an AI-powered customer service chatbot (Microsoft, n.d.c). Azure Machine Learning's footprint is substantial, driving enterprise towards cloud-based machine learning solutions while democratising access for smaller entities. The platform's commitment to seamless integration and scalability has left an indelible mark on the landscape of AI-driven applications in industry.

3.14 Nextjournal

Nextjournal is a computing platform for reproducible research (Nextjournal, n.d.b). It emerged in early 2017 with the aim of bringing reproducibility and collaboration to the forefront of research (Sierles, 2017), ensuring consistency in studies and simplifying the dissemination efforts.

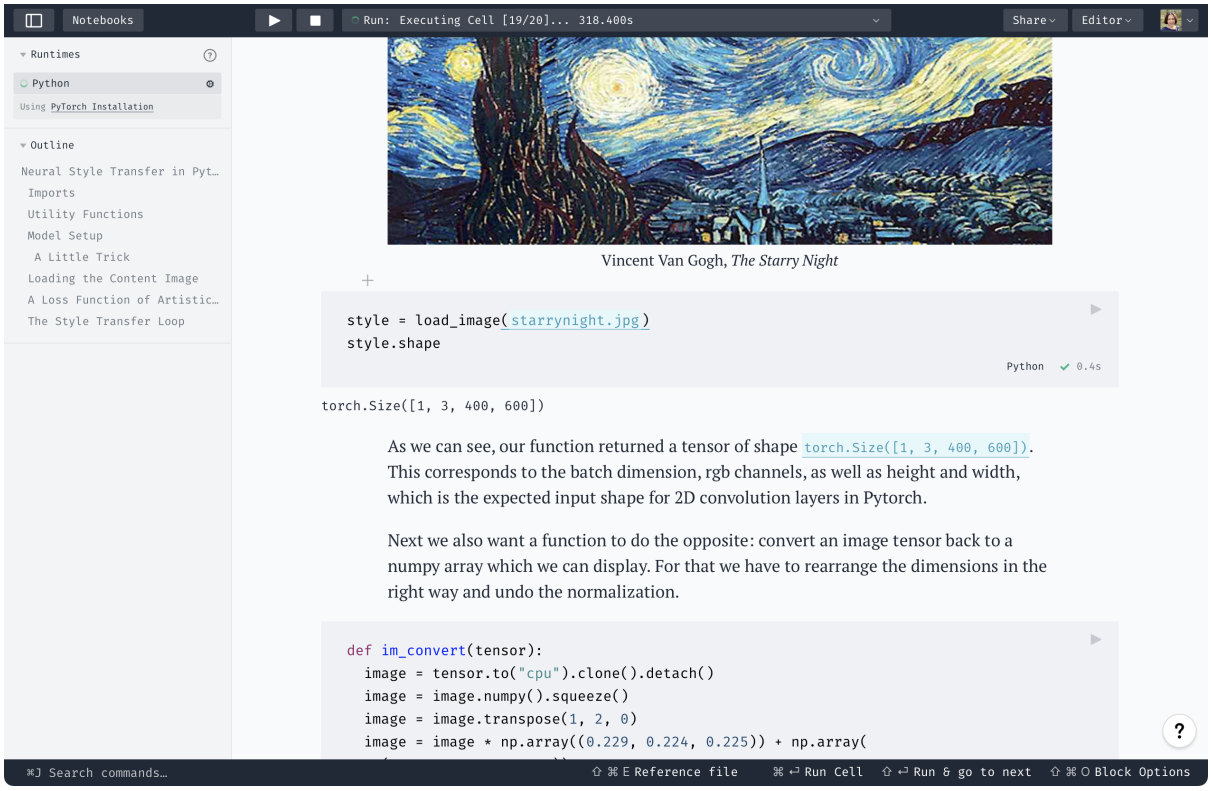


FIGURE 3.14: Nextjournal's interface.

Nextjournal's conception was sparked by the increasing complexity and fragmentation in research and data analysis (Sierles, 2017). The founders recognised the challenges practitioners

faced in ensuring reproducibility. Through development by a dedicated team at Nextjournal GmbH (Nextjournal, n.d.a), Nextjournal embraced advancements such as polyglot language support (Nextjournal, n.d.b), allowing users to use multiple languages in the same notebook.

Nextjournal sets itself apart through the use of Docker containers, furthering reproducibility (Nextjournal, n.d.b). The exact computational environment, down to specific library versions, can be consistently replicated by those with Docker configured on their machine. Its browser-based interface is depicted in Figure 3.14 (Nextjournal, n.d.b). While Python remains a primary language, Nextjournal also supports R, Julia, Clojure, Bash, and Agda (Markovics et al., 2021). Notably, it promotes collaboration through features like real-time co-editing (Nextjournal, n.d.b), broadening its appeal to team-oriented research environments.

Numerous institutions, including NASA, the University of Michigan, and the Swiss Institute of Bioinformatics, have harnessed Nextjournal to further research efforts (Nextjournal, n.d.b). The platform's enduring legacy is its robust stance on reproducibility and collaboration, accentuating the importance of consistent, shared research environments in the computational sciences.

3.15 Colab

Colab is a hosted low-setup Jupyter-based platform which provides free access to accelerated computing hardware (Google, n.d.c). Google introduced Colab in 2017 to promote machine learning education (Gershgorn, 2017) , offering an accessible platform for experimentation and sharing.

The platform's genesis can be attributed to Google's aspiration to further its foothold in the cloud-based computational ecosystem. The platform, a brainchild of Google Research (Gershgorn, 2017), represented a continuation of Google's efforts to support machine learning pedagogy. The platform has undergone several enhancements since its inception, notably introducing subscription tiers for extended resources (Google, n.d.a).

Colab's foremost feature is its native GPU and TPU support (Google, n.d.c), allowing expedited machine learning model training. The user interface, shown in Figure 3.15, offers an environment that should be familiar to longstanding Jupyter users. Despite Python remaining the solely-supported language, the platform also integrates with major ML libraries such as TensorFlow and

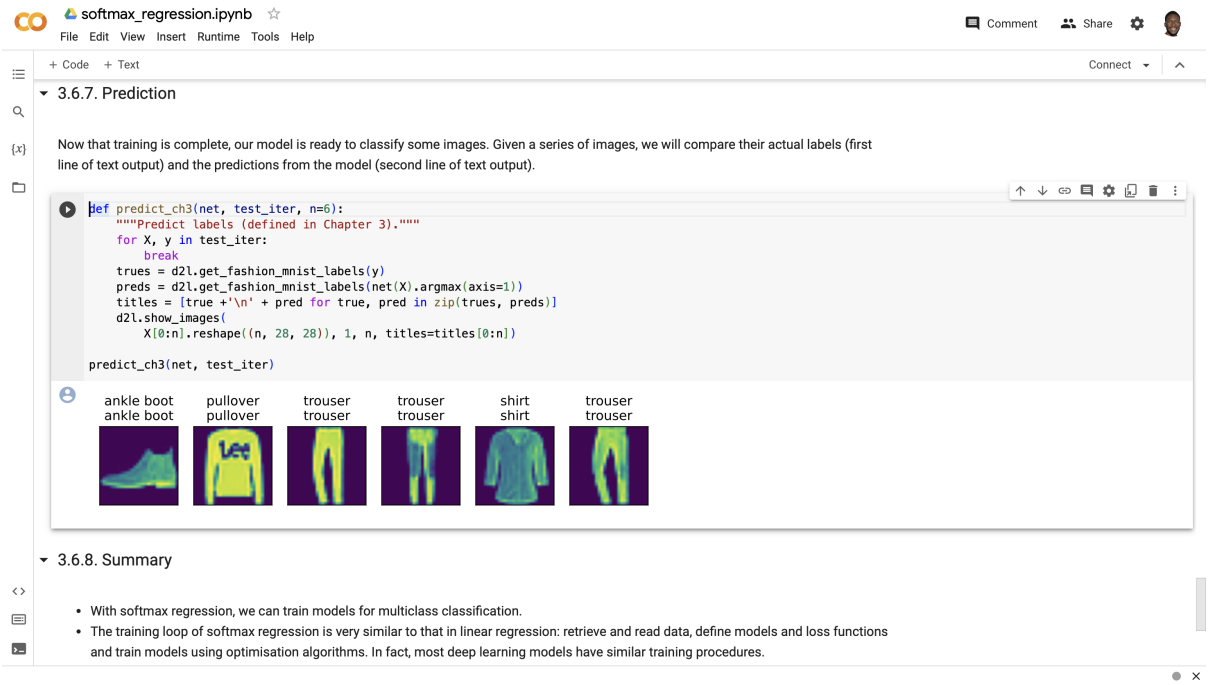


FIGURE 3.15: Colab's interface.

Keras (Google, n.d.c). Further integration with Google Drive streamlines data sourcing (Google, n.d.c). Ultimately, such integrations cement Colab's position as a comprehensive platform for collaborative AI/ML research and education efforts.

Colab has seen widespread adoption by educational institutions, facilitating hands-on learning in machine learning courses and workshops. Renowned organisations like Princeton University have since integrated Colab into their curricula (Johnston, 2020), bolstering pedagogy in scientific computing. Colab's overarching influence has been its democratisation of machine learning, offering unparalleled access to high-end computational resources.

3.16 SageMaker

SageMaker is a hosted ML platform which enables practitioners to build, train, and deploy models on the cloud (Amazon, n.d.b). Debuted in 2017, its goal was to streamline the ML lifecycle with a particular emphasis on scalability (Miller, 2017).

SageMaker's inception was rooted in Amazon Web Services' ambition to make machine learning workflows on the cloud more accessible. Under this banner, SageMaker developers consistently

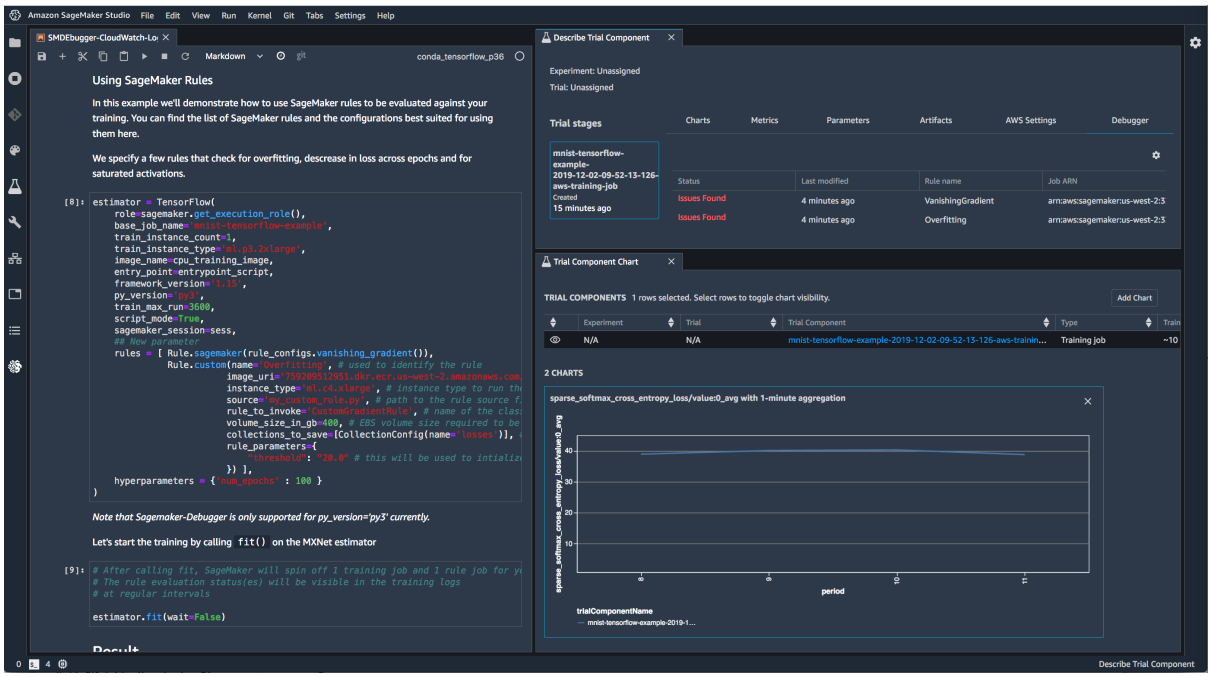


FIGURE 3.16: SageMaker's interface.

integrated cutting-edge tools and paradigms, like reinforcement learning (Amazon, n.d.l) and automatic model tuning (Amazon, n.d.g).

SageMaker has numerous built-in offerings, such as a K-Means algorithm (Amazon, n.d.e) and Linear Learner (Amazon, n.d.f), enabling users to bypass the intricacies of algorithm design. The interface of SageMaker Studio, its accompanying user interface, is shown in Figure 3.16 (Amazon, n.d.d). Through its Python-supporting environment, SageMaker supports prevalent ML packages, including MXNet (Amazon, n.d.j), TensorFlow (Amazon, n.d.m), and Hugging Face (Amazon, n.d.k). Furthermore, SageMaker integrates with Amazon S3 (Amazon, n.d.i) to support data sourcing, expediting the data loading process.

SageMaker has been instrumental to Airbnb, utilising the platform to classify text from customer service logs in order to improve overall customer satisfaction (Amazon, n.d.c). Akin to similar platforms, SageMaker's impact resonates in its democratisation of machine learning—enabling organisations, irrespective of size, to optimise ML workflows.

3.17 Observable

Observable is a hosted data visualisation platform that offers a reactive notebook for data exploration (Observable, n.d.b). Announced in 2018, its vision was to offer a more dynamic approach to data visualisation using web technologies (Observable, n.d.a).

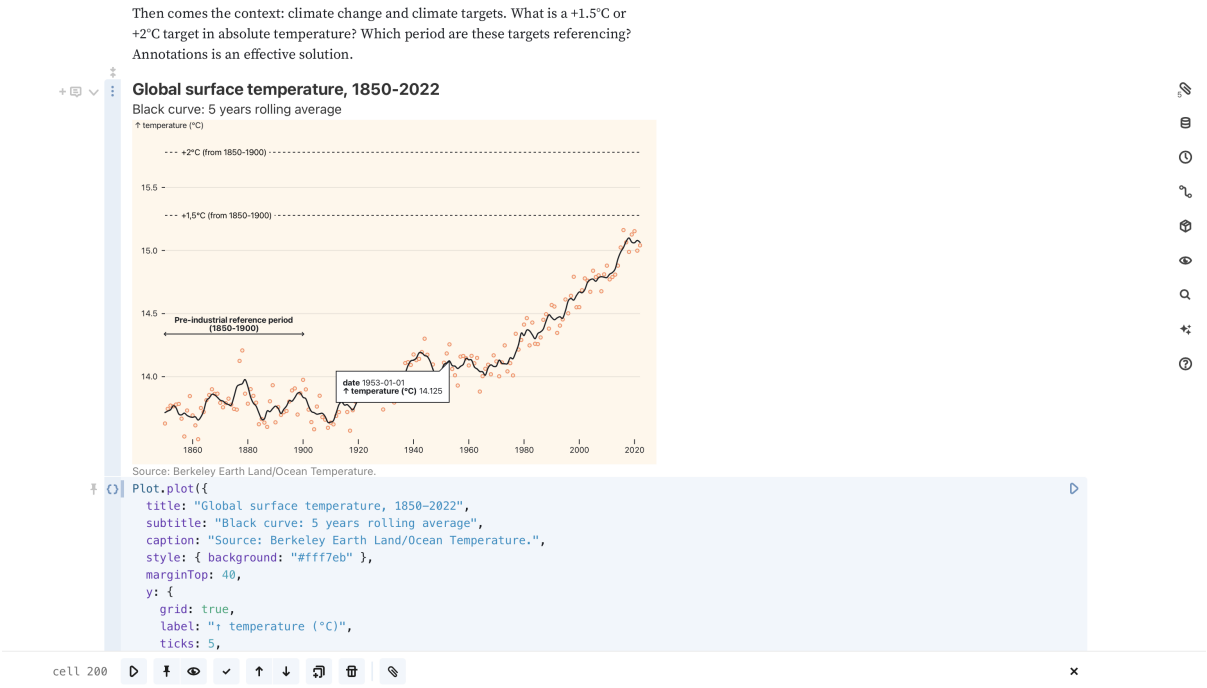


FIGURE 3.17: Observable's interface.

Founded by Mike Bostock, the creator of D3.js, and Melody Meckfessel (Observable, n.d.a), Observable's inception was influenced by the quest for better data visualisation tools. Over time, Observable has released multiple versions, iterating on user feedback and web technology developments, embracing features such as real-time collaboration (Meckfessel, 2021).

Observable's distinctive feature is its reactive programming model, where cell changes automatically propagate to dependent cells (Kosara et al., 2018). The platform's notebook interface is shown in Figure 3.17 (Ansart, 2023). Predominantly supporting JavaScript, Observable has integrated with libraries such as D3.js (D3.js, n.d.), enabling greater flexibility in visualisation. Furthermore, the platform's modularity facilitates embedding of cells in external web pages (Observable, 2023), broadening its applicability beyond the confines of the notebook interface.

Observable has found particular relevance to news outlets, evidenced through routine use by The Economist, LA Times, and NBC News (Observable, n.d.b). In 2020, NBC News utilised the platform to explore historical unemployment rates in the United States (Barquinha, 2020). Such usage underscores the Observable's capacity to simplify complex data for vast readership, further epitomising its transformative impact on modern data journalism.

3.18 Datalore

Datalore is a cloud-based collaborative data analytics platform (JetBrains, n.d.c). Initially launched in 2018 (JetBrains, 2018), Datalore sought to streamline collaboration between data science, engineering, and business teams (JetBrains, n.d.c).

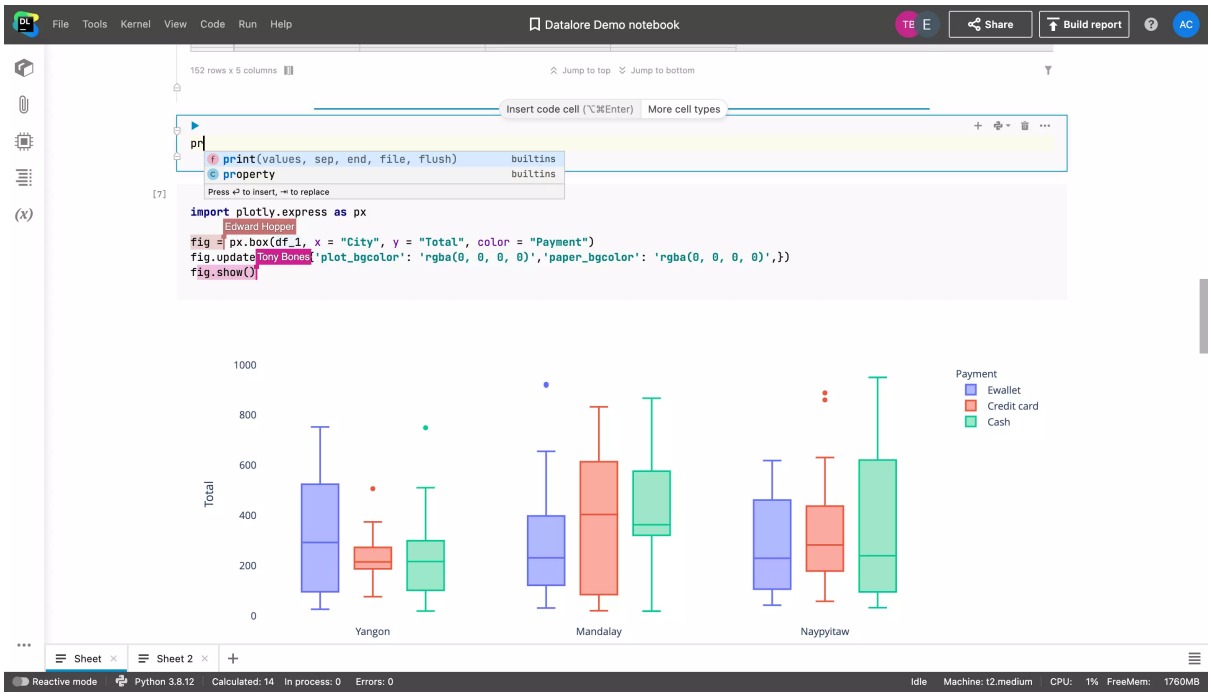


FIGURE 3.18: Datalore's interface.

The JetBrains team, renowned for their prowess in IDE development, saw an opportunity to capitalise on cloud-based data analysis. As Datalore matured, it embraced more sophisticated integrations. Its trajectory accentuated a commitment to efficient, collaborative data analysis, bridging the chasm between traditional IDEs and modern computational notebooks.

One of Datalore's hallmark features is intelligent code completion, capable of context-aware recommendations (JetBrains, 2023a). Its notebook interface, shown in Figure 3.18 (JetBrains, n.d.b), facilitates real-time collaboration (JetBrains, 2022), version control integration (Guzharina, 2023) and workbook sharing for team-based projects (JetBrains, 2022). The platform also integrates with IPython magics (JetBrains, n.d.e), broadening functionality.

Datalore's customer base encompasses numerous organisations in sectors ranging from blockchain analysis to telecommunications (JetBrains, n.d.a). LINE Corporation, a leading Japanese internet company, adopted Datalore to achieve existing business goals and satisfy data governance requirements (JetBrains, n.d.d). Datalore's report-sharing capability was a standout (JetBrains, n.d.d), promoting efficient communication across business lines. With such collaborative features, the platform continues to augment cloud-based teamwork, setting a precedence for future cloud-hosted solutions.

3.19 Deepnote

Deepnote is a hosted Python-supporting computational platform for data analytics (Deepnote, n.d.c). The platform was founded in 2019 (Crunchbase, n.d.) with the goal of providing teams with a more effective solution for working with data.

Deepnote originated in the backdrop of a flourishing landscape. Still, collaboration has remained an ongoing issue with numerous platforms. Jakub Jurových, the company founder (Jurovych, 2022), alongside a dedicated team, sought to sooth this pain point. The development trajectory of Deepnote has been marked by integrations such as interactive charts (Szalai, 2023), underlining the platform's commitment to redefining collaborative data analytics.

Other collaborative features include real-time co-editing (Deepnote, n.d.e) and in-notebook commenting (Deepnote, n.d.b). Its user interface is depicted in Figure 3.19. Deepnote natively supports Python and integrates with vital libraries like pandas (Benedict, n.d.), scikit-learn (Valdarrama, n.d.), and TensorFlow (kangar dev, n.d.). Moreover, direct connectivity to data sources such as PostgreSQL (Deepnote, n.d.d) and S3 (Deepnote, n.d.a) showcase its versatility, making it apt for contemporary data science tasks.

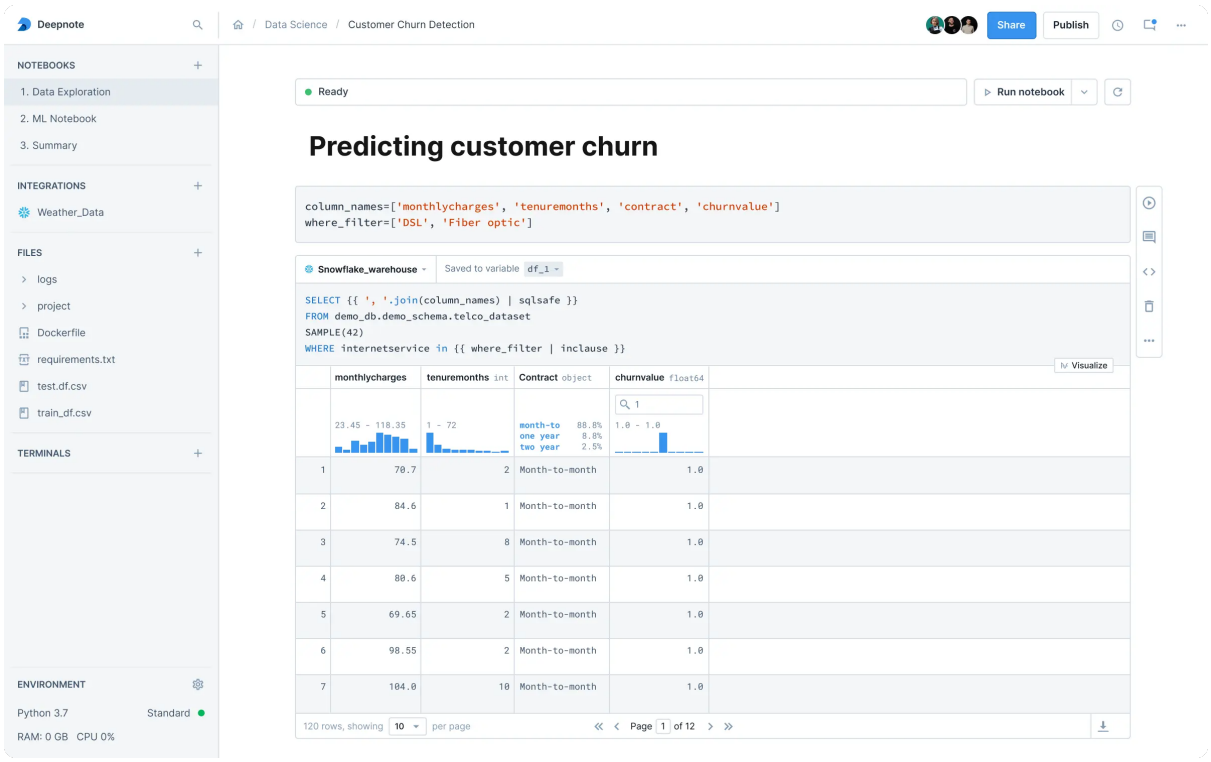


Figure 3.19: Deepnote's interface.

Numerous companies have adopted Deepnote to simplify their data science workflows. Webflow, a leading website building and hosting company, employed the platform to run statistics, prototype SEO solutions, and share metrics (Wendt, 2023). Deepnote's emergence has not only further moulded collaborative dynamics in data science but also accentuated the importance of real-time engagement on computational notebook platforms.

Chapter 4

Solution

4.1 Implementing the interface

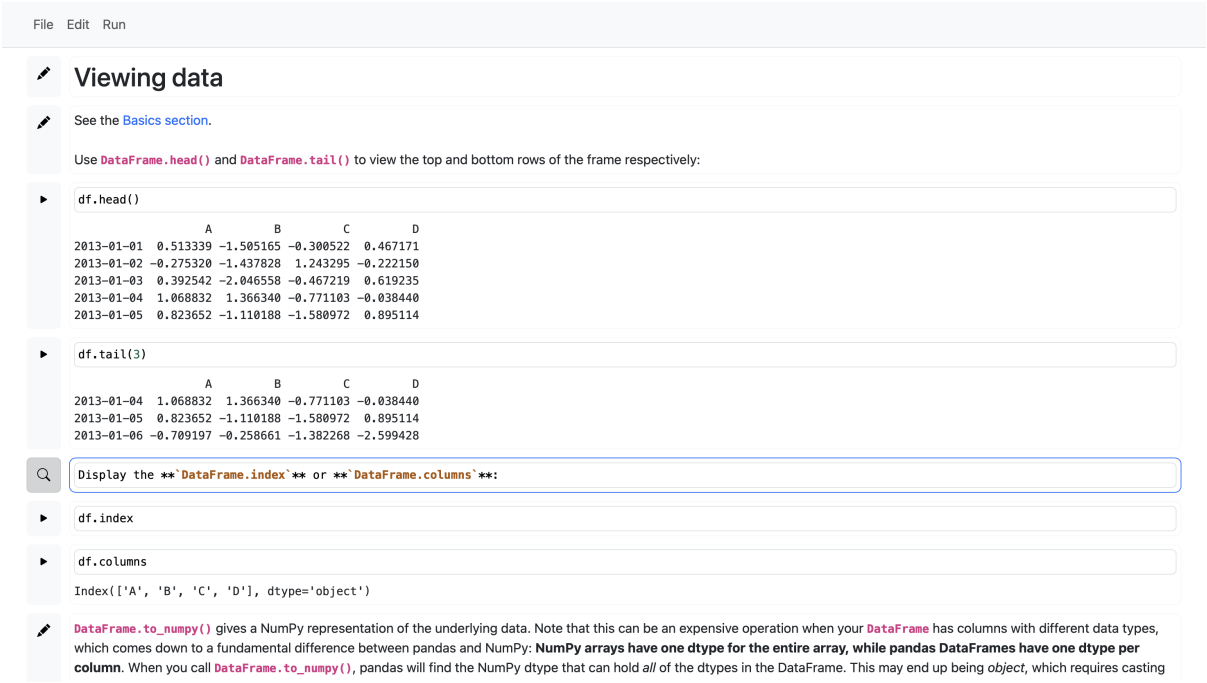


Figure 4.1: *Yuna*'s interface.

4.1.1 Supplying input cells

Yuna provides Python and Markdown cells, facilitating code execution and rich-text documentation, respectively. These cells are provisioned by CodeMirror 5, an in-browser text editor solution

(CodeMirror, n.d.). Version `5.65.13`, the most recent release at the time of development, was selected for integration (CodeMirror, 2023).

Core logic, found in `codemirror.js`, governs the behaviour and structure of both the Python and Markdown cells, provisioning functionalities from basic editing to advanced features such as line wrapping. Its companion, `codemirror.css`, ensures visual consistency. This CSS file defines the styling attributes of all cells, ensuring a pleasant user experience which upholds functionality while appealing to visual aesthetics.

For Python cells, syntax-highlighting is necessary in order to distinguish Python's rich vocabulary. This is achieved through `python.js`, a specialised script tailored for Python syntax-highlighting within CodeMirror 5. It accentuates Python-specific constructs, keywords, and syntax, ensuring readability and visual differentiation.

Markdown's unique set of notations and formatting rules also requires syntax-highlighting. `markdown.js` meets this end, ensuring clarity and distinction between standard text and Markdown-specific structures. It delineates structures such as headers, links, and lists, through italicising, colouring, and emboldening.

Found within `template.html`, self-implemented JavaScript functions, such as `configureCodeInputArea()` and `configureTextInputArea()` serve as bridges, connecting CodeMirror functionality to *Yuna*'s custom notebook interface. Naturally, these functions configure the input areas for Python and Markdown cells, respectively. They:

(1) Retrieve the cell's underlying `<textarea>`.
(2) Instantiate a CodeMirror instance, ensuring the correct input type, indentation units, and line-wrapping is set.
(3) Copy previously-saved user input from the cell's hidden `div#saved-code-area` or `div#saved-text-area` elements and paste it into the cell.

4.1.2 Provisioning interactivity and styling

Bootstrap, a prominent open-source frontend framework (Bootstrap, n.d.), is employed to grant further aesthetic and functional enhancements. Two main components, `bootstrap.bundle.min.js`

and `bootstrap.min.css`, extracted from version `5.3.0-alpha3` (XhmikosR, 2023), are embedded into the environment. The former provides Bootstrap's interactivity components, while the latter grants the essential styles that are ubiquitous in contemporary web interfaces.

`bootstrap.bundle.min.js` is responsible for endowing interactive behaviours to the notebook interface elements. For instance, the collapsible navigation menu is implemented by Bootstrap's built-in toggling logic, implemented by the `data-toggle` and `data-target` attributes used in the `div#tool-bar` element. The `navbar-toggler` class, paired with this logic, enables responsiveness, collapsing on smaller screens, and expanding on larger ones.

Simultaneously, `bootstrap.min.css` dictates the styling—from colours and typography to layouts and spacing—ensuring that these interactive elements are visually appealing and intuitive. The intrinsic elements of the notebook, namely the toolbar, Python cells, and Markdown cells, are styled with a consistent and familiar theme, owing to Bootstrap's CSS classes, coupled with self-implemented CSS.

The toolbar houses the primary computational notebook options, "File", "Edit", and "Run". Its fixed position at the top of the browser window ensures familiar access. Bootstrap's `navbar` class is employed to structure and style the toolbar, with dropdown menus facilitating a cleaner, organised display of options.

Each Python cell features a "Run code" button decorated with a play icon, allowing users to execute the code contained within. A container, implemented by the `div#code-input-output-container` element, wraps the code input and output areas, featuring a distinctive rounded border for separation. Bootstrap's `btn`, `border`, and `svg` classes play an integral in styling these sub-components.

Markdown cells also feature interactive buttons: one for previewing and another for editing documentation. The "Preview" button is adorned with a magnifying glass icon, while the "Edit" button is furnished with a pencil icon. Akin to each Python cell, the `div#text-input-out-container` element encapsulates text input and output areas. They also feature rounded borders—implemented by Bootstrap's `border`, `border-light`, and `rounded-3` CSS classes—ensuring consistency with the design of the Python cells.

4.2 Executing code

Code execution engine is driven by Pyodide (Pyodide, n.d.a), specifically `version 0.23.2` (pyodide-ci bot, 2023). `pyodide.asm.wasm` and `pyodide.asm.js` are of particular importance. `runCode()`, a custom JavaScript function, is employed to enable high-performance execution of Python code within the confines of each HTML file.

Upon interaction with the "Run code" button, `runCode()` first captures the code written in the cell as a string. Thereafter, the function employs `Pyodide.runCode()` to execute the code string, storing the result. Finally, the function clears the output area, inserts the result inside of it, and un-hides the area, immediately rendering the result in the browser.

4.2.1 Porting the runtime

To provision this execution environment, Pyodide uses WebAssembly (Wasm). Wasm is a binary instruction format for compiling programming languages for web-based contexts (WebAssembly, n.d.). Through Pyodide's Wasm support, *Yuna* is empowered with a highly optimised Python runtime, making real-time computations in a self-contained system not just feasible but efficient.

The Wasm module itself, `pyodide.asm.wasm`, represents this port. Housed within the designated `div#pyodide-web-assembly-module` element, the original file is encoded using Base64, a binary-to-text encoding schema (Mozilla, 2023a). The result is an associated string representation of the module, suitable for embedding within the singular-file HTML environment. This encoding is then concealed using the `display: none` attribute, ensuring that it remains invisible to end-users but accessible programmatically.

Contained within the custom `RuntimeLoader` class, `pyodide.asm.js` represents the foundational logic which prepares the Python Wasm module for execution, loading `pyodide.asm.wasm` asynchronously. `RuntimeLoader.pyodideAsmJS()`, the getter which houses `pyodide.asm.js`, employs a modified `readAsync()` function. This function retrieves the Base64-encoded Wasm module, decodes the module into a byte array, and subsequently feeds it into the Wasm loading mechanism, thus enabling Python's runtime environment.

4.2.2 Interfacing with the runtime

At the heart of `pyodide.js` is the `loadPyodide()` asynchronous function. It is responsible for configuring and initialising the Pyodide environment. The function begins by detecting the runtime environment and setting up the necessary paths and configurations. `loadPyodide()` then creates a `Module` object, which encapsulates configurations, input/output streams, and other utilities required for the Python Wasm runtime. It verifies if the correct Pyodide version is being used, bootstraps the necessary packages, and initializes I/O streams.

Integration with the Wasm context is achieved by linking with `pyodide.asm.js`. Typically, if the necessary module for Pyodide isn't already available in the environment, `loadPyodide()` fetches the script. However, given *Yuna*'s self-contained architecture, the source code was modified so that `loadPyodide()` would serve the necessary modules from within the file itself.

The self-implemented `EnvironmentManager` class leverages these functionalities to bridge the gap between the interface and execution engine. `EnvironmentManager.initialisePyodide()` asynchronously initialises Pyodide and redirects its console output for in-notebook display. The method also loads micropip and cloudpickle to facilitate the installation of user-specified packages and restoration of session data, respectively. Upon the user interacting with a Python cell's "Run" code button, `EnvironmentManager.runCode()` calls `Pyodide.runPython()` to execute its code, returning the result from Python to JavaScript for subsequent display in the browser.

4.2.3 Incorporating the standard library

At the core of Python's wide-ranging versatility lies its far-reaching standard library, providing numerous tools and functionalities to support a multitude of programming tasks. In order to provision a fully-featured Python experience, its incorporation thus becomes essential. Doing so ensures that the user, while conducting work in a *Yuna* environment, can harness the full spectrum of Python's native capabilities.

Python's entire standard library, provisioned by Pyodide through `python_stdlib.zip`, is encoded into Base64 string format. Once again, this binary-to-text protocol transforms the archive into

a standard string, embeddable within the HTML file. The encoded data is housed in the hidden `div#python-standard-library` element. This approach hides the voluminous data of the standard library while allowing for its complete containment.

For the purpose of library retrieval, custom JavaScript logic decodes the Base64 string, reinstating its binary avatar, and then converts the string into a `Uint8Array`. Subsequent to this transformation, the standard library finds its rightful place in the Python runtime, elevating its functionality. Again, the employment of the highly-performant Wasm format ensures use of the standard library is bereft of any performance-related compromises.

4.3 Loading packages

4.3.1 Embedding packages

In the over-archiving *Yuna* ecosystem, `yuna.py` is the primary utility used construct *Yuna* environments. It provides the mechanisms for embedding user-specified Python packages and their associated dependencies into the HTML file.

The primary function, `__main__()`, parses command-line arguments to determine the environment builder's intent. Two usages are accommodated: one in which an environment with specific package requirements is desired, and another where an environment is created without explicit requirements. As such, the script can take two arguments, a filename for the environment and another for the list of requirements. Alternatively, solely an environment filename can be provided. The syntax is as follows:

```
yuna.py <environment_filename>.html [requirements_filename].txt
```

`buildEnvironment()` orchestrates the process of environment construction. The function initiates by creating a copy of `template.html`, the foundational template which serves as the baseline of each *Yuna* environment. It then reads an optional user-specified requirements file and has the associated packages embedded within the copy.

`readRequirements()` initially receives these requirements, reading the requirements file to create an associated Python list. `embedRequirements()` is then invoked to facilitate the embedding

process. It begins by validating the specified requirements against the repository of supported Python packages, represented by the `REPODATA` dictionary.

If validation is successful, `embedRequirements()` then employs Beautiful Soup 4, a Python package for parsing HTML (leornard, 2023), to traverse the HTML structure and write the encoded packages inside their respective `div` elements. A breadth-first search algorithm implemented by `bfs()` is used to ensure that all dependencies are recursively embedded.

4.3.2 Installing packages

Package installation is achieved through the strategic amalgamation of micropip, a lightweight Python package manager for Pyodide (Pyodide, n.d.b), and packaging, a standardised module for package verification (brettcannon et al., 2023). These utilities are encoded as Base64 entities, contained within respective hidden `div` elements. Versions `0.3.0` and `23.0`, respectively, are used to implement *Yuna*'s current release.

The core mechanics of *Yuna*'s package loading system can be gleaned from `EnvironmentManager.initialisePyodide()`, operating as follows:

(1) Initialise Pyodide asynchronously by invoking `Pyodide.loadPyodide()`.
(2) Redirect standard output to a string buffer, for capturing and displaying `print()` outputs in the browser.
(3) Install micropip into the runtime with `Pyodide.loadPackage()`.
(4) Install cloudpickle with `micropip.install()`.
(5) After `Pyodide.loadPyodide()` returns, invoke `EnvironmentManager.loadRequirements()`, installing embedded packages using `micropip.install()`.

4.4 Restoring the session

cloudpickle, a Python module for enhanced pickling support (antoine et al., 2023), alongside custom Python and JavaScript logic, is harnessed to ensure the continuity of the user's computational work. The session—encompassing variables, function, and imports—associated with a particular environment is embedded in the file, ready for subsequent restoration.

4.4.1 Serialising objects

`pySerialiseObjects()` and `jsSerialiseObjects()` embody the serialisation process. When the user interacts with the "Save environment" toolbar option or triggers the "Save" command on their machine, object serialisation begins with an invocation of `saveEnvironment()`. The function then calls `jsSerialiseObjects()`, which ultimately invokes `Pyodide.runPython()` to trigger `pySerialiseObjects()`.

Iterating through all user-defined objects, `pySerialiseObjects()` operates as follows:

(1) Pickle the object.
(2) Encode the pickled object.
(3) Add the object's name and encoding to a dictionary.

This fully populated dictionary is then returned to `jsSerialiseObjects()` for subsequent embedding.

`jsSerialiseObjects()` first facilitates its translation from a Python `dict` to a JavaScript `Map`. For each name–encoding entry, `jsSerialiseObjects()`:

(1) Creates a `div` element with the element's `id` associated with the object's name.
(2) Writes the encoding inside the element.
(3) Appends the element to the `div#objects` container.

Execution is then passed back to `saveEnvironment()`, where the object-embedded environment is saved locally to the browser's "Downloads" directory.

4.4.2 Deserialising objects

The subsequent deserialisation process is conducted by `jsDeserialiseObjects()` and `pySerialiseObjects()`. Upon the user opening a previously saved notebook, `restoreNotebook()` commences deserialisation. This function triggers `initialisePyodide()`, which ultimately invokes `jsDeserialiseObjects()`.

For all embedded objects, represented as `div` elements inside the `div#objects container`, the method:

(1) Retrieves the element's `id` attribute.
(2) Retrieves the element's `innerHTML`.
(3) Places the element's `id` and into a map, reinstating the previously-established name–encoding association.

`jsDeserialiseObjects()` finally adds the map to the `window` object for access in the Python runtime and harnesses `Pyodide.runPython()` to call `pyDeserialiseObjects()`.

This function first converts the JavaScript `Map` to a Python `dict`. Then, iterating through each name–encoding pair:

(1) Decodes the encoded object.
(2) Unpickles the decoded object.
(3) Places the restored Python object back in the `globals()` dictionary.

Once this process is complete, the user can immediately resume computation.

4.5 Parsing text

Marked, a JavaScript compiler for parsing markdown (Marked, n.d.), powers rich-text documentation. In particular, version `5.0.2` is utilised in *Yuna*'s current release (github-actions, 2023). Core logic, implemented in `marked.min.js`, is completely embedded within the file. Custom JavaScript, namely the `previewText()` and `editText()` functions, bind the text cells with their supporting logic, achieving seamless translation between Markdown and HTML.

When the user interacts with a cell's "Preview text" button, `previewText()` initially retrieves the user input enclosed within. After validating the input isn't purely whitespace, the function then calls `Marked.parse()` to transform the Markdown syntax into HTML syntax. Finally, `previewText()` hides the input area and inserts the HTML syntax into the cell's output area, immediately rendering the user's narrative.

Conversely, the `editText()` function reinstates the user's raw input. When the user interacts with the cell's "Edit text" button, the function is triggered. `editText()` promptly hides the output area and reinstates the original Markdown input, allowing the user to further write their documentation.

Chapter 5

Evaluation

To shed light on the rapidly-evolving domain of computational notebook platforms, Lau et al. (2020) undertook an extensive analysis of such platforms. They presented their findings in tabular-format, plotting each platform against workflows similar to those found by Chattopadhyay et al. (2020), the workflows discussed in this dissertation. The evaluation presented here embodies a similar format, but instead considers the workflows from Chattopadhyay et al. (2020). Alike Lau et al. (2020), each platform's features were identified by consulting official guides, documentation, and videos, and trying platforms where possible. Table 5.1 presents this evaluation.

5.1 Setting up

5.1.1 Loading data

Loading data refers to the process of importing raw data into the environment for subsequent analysis or exploration. This data can be sourced from the local file system or an external storage solution such as Amazon S3 (Amazon, n.d.a) or Google Drive (Google, n.d.b). At present, importing data can quickly balloon into a complex, multi-step chore, as data often needs to move across multiple platforms (Chattopadhyay et al., 2020).

psycopg (piro, 2023) and similar data libraries promise solutions, yet they are third-party solutions, and thus, might not holistically integrate with the user's dataset. Faced with these challenges, many users resort to others for assistance. One user had a developer write magic commands in their notebook to simplify data connection to commonly-used sources (Chattopadhyay et al., 2020). However, not all notebook users have others to rely on for help.

	Azure Machine Learning	Colab	Databricks	Datalore	Deepnote	Jupyter Notebook	Kaggle	Mathematica	MATLAB	Nextjournal	Observable	RStudio	SageMaker	Yuna
Loading data	•	•	•	•	•		•			•	•	•	•	
Cleaning data	•		•	•					•				•	
Modelling														
Visualising	•	•	•	•	•	•	•	•	•	•	•	•	•	
Iterating	•		•	•	•			•	•	•	•	•	•	
Writing code	•		•	•	•			•	•	•	•	•	•	
Managing dependencies				•										•
Debugging code	•		•	•	•			•	•			•		
Testing									•			•	•	
Executing						•		•	•			•		•
Scaling	•	•	•	•	•		•			•			•	
Versioning										•				•
Searching			•	•										
Handling sensitive data														
Controlling access	•	•	•	•	•		•			•	•		•	
Sharing the notebook	•	•	•	•	•	•	•	•	•	•	•	•	•	•
Sharing supporting artefacts														•
Editing collaboratively		•	•	•	•					•	•		•	
Reproducing														•
Reusing and adapting				•										•
Scheduling long-running computations	•		•	•	•					•			•	
Packaging as a standalone artefact														

Table 5.1: Evaluation of computational notebook platforms.

To provision an effective solution to loading data, any given computational notebook platform must offer an integrated interface for data transfer. This interface must natively provision connections between the platform and relevant data sources, such as cloud storage providers or the local file system.

5.1.2 Cleaning data

Data cleaning involves fixing or removing erroneous, corrupted, improperly-formatted, redundant, or incomplete data from a dataset. Models benefit significantly from the presence of clean data, but data is more often dirty than clean (Chattopadhyay et al., 2020). Presently, efforts to scrub data are mostly clerical—the diversity in datasets means that it's virtually impossible to prescribe a one-size-fits-all cleaning procedure (Chattopadhyay et al., 2020).

Despite the prevalence of data cleaning platforms like OpenRefine (OpenRefine, n.d.), the inherently repetitive nature of data scrubbing is often left unaddressed. To mitigate the repetition of cleaning data, many users are forced to implement their own data cleaning solutions (Chattopadhyay et al., 2020). However, these implementations often need tweaks to tailor each dataset, and transferring code between notebooks via copy–paste methods is highly-error-prone (Chattopadhyay et al., 2020).

To temper frustrations associated with scrubbing data, any given platform should integrate a suite of automated data cleaning tools. These tools should allow for repeatable, customisable workflows, minimising manual interventions and inevitable user errors.

5.2 Exploring and analysing data

5.2.1 Modelling

Modelling entails the intricate process of constructing and refining algorithms to represent and analyse data patterns. Not only does it take a long time to have the platfrom learn the model, often without feedback, the intricacies of each model mandate considerable time in the design and implementation phases.

GPU-provisioned cloud-based platforms, such as Colab (Google, n.d.c), offer a solution to the modelling problem as users can utilise high power GPUs through their Colab Pro and Colab Pro+ tiers (Google, n.d.a). This, however, would require users to migrate the entirety of their work to an alternative platform, significantly impacting productivity. Consequently, users are forced to wait extended periods of time to verify execution (Chattopadhyay et al., 2020). Worse, the lack of feedback disallows for evaluating alternatives in the meantime (Chattopadhyay et al., 2020).

Any given solution must offer numerous enhancements to provision satisfactory modelling. The platform should natively support features such as real-time feedback, iterative adjustments, mid-process evaluations, and error-handling without full re-execution.

5.2.2 Visualising

Visualising refers to the practice of using graphical representations, like plots, to gain insights from data. Whilst crafting a computational narrative, users make use of such visualisations to rapidly observe how data is altered by their code tweaks. Yet, they experience great difficulty customising plots, as they cannot be directly interacted with (Chattopadhyay et al., 2020). Users fall into the endless cycle of tweaking or copying and pasting code to modify visualisations (Chattopadhyay et al., 2020).

Data analysis and visualisation platforms, such as Tableau (Tableau, n.d.), provide interactive visualisation. Its nature as a third-party solution, however, means integration would likely be non-trivial. With many notebooks failing to provide such rich visualisation, users are forced to work within the confines of their notebook environment. Unsatisfied, many users manually export data and conduct visualisation on such platforms, regardless (Chattopadhyay et al., 2020).

Any given solution must offer interactive visualisation customisation, native integration with a third-party visualisation platform, or adaptive cell resizing for unrestricted graphical representation.

5.2.3 Iterating

Iterating is the recurrent refinement of methods, typically between modelling and visualisation, aiming for configurations which yield optimal model performance. However, the lack of code assistance often means that iteration is torturous (Chattopadhyay et al., 2020). To clean up the notebook for subsequent iterations, the user must painstakingly comb through the environment to delete now-unused code, find required packages, and install these packages into the runtime (Chattopadhyay et al., 2020).

Numerous IDEs, like Visual Studio Code (Microsoft, n.d.e), offer a direct solution. They provision numerous code assistance features, including as auto-complete (Microsoft, n.d.b). The general purpose nature of typical IDEs, however, means they are unsuitable for many data-centric workflows. Regrettably, users can switch between a computational notebook and an IDE. However, the associated overheads render this approach entirely unfeasible.

Computational platforms must offer robust code assistance, straightforward package installation, or efficient code management tools to facilitate streamlined iteration between modelling and visualisation, nullifying all dependence on IDEs.

5.3 Managing code

5.3.1 Writing code

Writing code involves the manual process of composing source code that an interpreter can execute to produce some desired output. It entails implementing algorithms, well-defined step-by-step sequences of instructions, in one or more programming languages. With the absence of code assistance, many users find writing code to be a major source of friction in the narrative-writing process (Chattopadhyay et al., 2020).

Again, Visual Studio Code (Microsoft, n.d.e) and similar IDEs offer a multitude of code assistance features (Microsoft, n.d.b). However, since they are tailored for general software development, and not data-centric programming, they don't provision feature sets wholly-applicable to data-based computation. As such, some users will open a browser to search for help and consult documentation, in addition to their notebook (Chattopadhyay et al., 2020).

To represent a solution to writing code, computational notebook platforms should offer integrated code assistance, supporting capabilities such as real-time API exploration, function/class auto-completion, and embedded documentation retrieval, eliminating the need for external resources.

5.3.2 Managing dependencies

Dependency management is the process of handling, updating, and resolving software packages a project relies on, ensuring ongoing functionality and compatibility across platforms and machines. Extant solutions offer piecemeal assistance in managing packages (Chattopadhyay et al., 2020). Resultantly, many users are plunged into "dependency hell" (Chattopadhyay et al., 2020), left to engage in the pain-staking task of managing dependencies manually.

Once more, Visual Studio Code offers a solution, suggesting "quick fixes" to patch smaller-scale issues (Microsoft, n.d.d). But the same issue outlined in *Sub-section 5.2.3: Iterating* and *Sub-section 5.3.1: Writing code* apply here—Visual Studio Code is a general-purpose, not data-centric, platform. Users are then forced to drudge through terminal outputs and execute commands such as `conda` and `pip` to manage their environment (Chattopadhyay et al., 2020).

A computational platform should offer integrated package management. It should offer capabilities such as seamless discovery, installation, updating, and deprecation handling directly within through the interface, obviating command-line interventions.

5.3.3 Debugging

Debugging refers to the systematic process of identifying, isolating, and resolving errors or anomalies in code, often involving supplementary tools. As users run their code, they often encounter inexplicable failures that their platform's debugging tools are insufficient to diagnose (Chattopadhyay et al., 2020). Non-sequential execution can complicate matters further.

Some environments, like JupyterLab (Project Jupyter, n.d.b), enable users to inspect variables, set execution breakpoints, and analyse the call stack at any given point (Project Jupyter, n.d.a). However, each solution is tailored for use in particular domains, and thus, might not be the most

appropriate for any given task. Most users have to scatter print calls all over their notebook to gain further insights into resultant failures.

Features such as integrated variable inspection, data introspection tools, or cell-level execution history should be provisioned, facilitating clearer code flow understanding and enabling more efficient debugging.

5.3.4 Testing

Testing entails the systematic evaluation and verification of code through various approaches, ensuring correctness, reliability, and desired functionality. Most platforms have little to no native testing functionalities available (Chattopadhyay et al., 2020). Similar to data scrubbing, there are no standard frameworks to which users can adhere in testing their notebooks (Chattopadhyay et al., 2020).

While testing packages such as `pytest` (Krekel and pytest, n.d.) exist, users are still faced with the same issue as data libraries—they are external solutions which often require significant time and effort in order to configure and integrate into existing workflows. These packages are also language-specific. As such, users might not stand to benefit if their workflows coincide with an alternate language. Consequently, testing procedures are often ad-hoc—certain users write test cases in the notebook itself, while others author new notebooks solely for testing (Chattopadhyay et al., 2020).

Any given platform should offer an integrated testing solution, allowing users to write, execute, and evaluate test cases, ensuring code reliability and functionality consistency.

5.4 Ensuring reliability

5.4.1 Executing

Executing refers to running code within a computational notebook, either within specific cells or the entire notebook. Disconnections can place the session or data into an inconsistent state (Chattopadhyay et al., 2020). Worse, the lack of transparency over state changes during execution means it's virtually impossible to profile and rectify inconsistencies (Chattopadhyay et al.,

2020). Cloud-based platforms—with their reliance on constant internet access—are further impacted, due to inherent network unreliability.

Qubole, a cloud-based data platform (Qubole, n.d.b), allows users of Jupyter Notebooks to access kernel logs (Qubole, n.d.a), potentially aiding in the resolution of state-based issues. Unfortunately, users of other environments are still left to find their own workarounds. Worse, Qubole is a paid platform (Qubole, n.d.c), meaning even Jupyter users might be prohibited from accessing its benefits.

For satisfactory ruling, a computational platform must offer persistent kernel connectivity. Real-time execution monitoring, automatic state recovery, and transparency in kernel activities are further enhancements. These features will ensure consistent state management, helping mitigate reliability concerns.

5.4.2 Scaling

Scaling refers to the ability to efficiently handle and process increasing volumes of data without compromising performance. One of the ever-present limitations with many interpreters is a lack of processing power (Chattopadhyay et al., 2020). Most solutions run on a single node, and are thus, unable to distribute tasks across multiple machines (Chattopadhyay et al., 2020). While notebooks are preferable for lightweight extracting, transforming, and loading (ETL), migration to more powerful platforms, like Databricks (Databricks, n.d.f), is necessary to handle big data (Chattopadhyay et al., 2020).

Platforms such as Apache Hadoop (Apache, n.d.) offer distributed storage and processing for petabytes of data. However, from conducted reviews, only Databricks offers native integration with Hadoop (Databricks, n.d.e). Alternatively, users can migrate to a solution like Colab in order to harness more powerful GPUs (Google, n.d.a). However, doing so would require additional financial expenditure. Furthermore, feature-sets would like differ, too.

To address scaling issues, a computational platform must provide seamless integration with external data processing pipelines and offer accelerated hardware access, enabling more efficient data handling and offloading of heavy ETL tasks without compromising capabilities.

5.5 Archiving

5.5.1 Versioning

Typical version control involves recording changes to source code files over time, enabling users to recall specific versions, facilitate collaboration, and track modifications. Computational platforms, however, differ from traditional scripts as they rely on a wider range of artefacts: data, plots, tables, and text outputs, all in addition to code. Importantly, the session data associated with a notebook further influences versioning. There's "a lot of room for improvement when we want to check notebooks into source control" (Chattopadhyay et al., 2020).

A version control system, like Git (Git, n.d.), allows for the versioning of notebook files. However, as previously-established, the broad spectrum of computational notebook artefacts means versioning notebooks alone is inadequate. As such, users find themselves creating many different notebooks and attempt to version these notebooks manually (Chattopadhyay et al., 2020).

To address versioning, the essential capability is an integrated version control mechanism. This mechanism must capture notebook contents *and* the session data—variables, functions, and imports—associated with the notebook.

5.5.2 Searching

Searching refers to the process of locating a specific computational notebook among numerous files and directories. Typical search mechanisms search implement title-only search algorithms. Over the course of a single computational process, a user might create numerous notebooks, each representing particular trials or approaches. This rapid accumulation means that notebook names "become disorganised very fast" (Chattopadhyay et al., 2020).

Very few environments allow for searching notebooks by content—Databricks and Datalore are among the few (Databricks, 2023f; JetBrains, 2023b). Many more platforms, such as JupyterLab, display an auto-generated table-of-contents in each interface (Project Jupyter, n.d.c). Fatally, this feature still requires users to open any notebook in order to conduct further searching, doing nothing to resolve the extant issue. Unsurprisingly, many users would find themselves overwhelmed by the multitude of indistinguishable files resulting from their work.

To address searching, computational platforms need to provision an integrated search functionality. This functionality must support both title-based and content-based indexing, streamlining file retrieval irrespective of naming conventions.

5.6 Securing

5.6.1 Protecting sensitive data

Securing data entails the process of protecting confidential information in notebooks from unauthorised access and inadvertent leaks. The process commonly involves the implementation of data transformation and suppression algorithms. Implementation, however, necessitates additional time and effort on behalf of the user (Chattopadhyay et al., 2020). Computational resources are also strained, slowing down the ultimate performance of the notebook or requiring investment in more performant hardware to restore performance (Chattopadhyay et al., 2020).

Once more, Colab and similar hosted platforms allow users to access greater GPU performance (Google, n.d.a), for an additional financial investment. Unfortunately, resource-constrained users, like students and hobbyists, simply cannot not afford the expenditure. Akin to *Sub-section 5.2.1: Modelling*, such users must then wait extended periods of time while their datasets undergo securing processes (Chattopadhyay et al., 2020).

Any given platform should provide cell-level masking, allowing users to mask content while retaining executability. Integrated data transformation tools and robust export safeguards can further prevent inadvertent data leaks and streamline the data science process.

5.6.2 Controlling access

Generally-speaking, access control refers to the act of granting or denying specific permissions to individuals or groups, ensuring appropriate rights to use, modify, or execute a system's resources. For many users, it's not possible to finely control notebook access—it's not possible to make a notebook read-only or only allow comments (Chattopadhyay et al., 2020). They also cannot restrict viewers to "only run but not modify cells" or "modify cells but not run" (Chattopadhyay et al., 2020). Such users are stuck in a dilemma—they need to collaborate to progress forwards, but doing can jeopardise data confidentiality (Chattopadhyay et al., 2020).

In a multi-user setup, some environments, such as Azure Machine Learning, allow administrators to provision and control workspace access (Microsoft, 2023a). This, however, doesn't address the underlying issue—a lack of notebook-level control. Unfortunately, granular in-notebook permissions are seemingly non-existent. Many users are left without reprieve—the "only way to share notebooks securely is to upload notebooks to secure team drives" (Chattopadhyay et al., 2020).

A computational platform must provide granular permission settings, enabling users to designate access levels on a per-notebook basis, such as read-only, comment-only, execute without modification, and full edit permissions.

5.7 Sharing and collaborating

5.7.1 Sharing the notebook

Sharing a notebook refers to the process of making the notebook accessible to others, either for collaborative or presentative purposes. In practically all contexts, from academia to industry, notebooks must be shared with a variety of stakeholders, many with limited domain knowledge. This is achieved in one of three ways: *interactively*, *statically*, or *read-only* (Chattopadhyay et al., 2020).

When sharing notebooks interactively, users run the notebook in real-time and explain the results to the audience (Chattopadhyay et al., 2020). Since viewing code isn't always meaningful, users want the ability to "collapse code". When sharing notebooks statically, authors share a notebook complete with analysis and results that recipients can interact with later (Chattopadhyay et al., 2020). When sharing notebooks as read-only artefacts, authors allow the recipient leave comments to which they can respond later (Chattopadhyay et al., 2020).

Interactive sharing promotes the highest level of engagement, and thus, allows for the most effective means of dissemination. However, actually delivering a presentation would require users to allocate time which could have been spent on learning or further research. As a result, many users would likely resort to exporting select notebook contents to other formats, like scripts and rich-text languages like Markdown.

To mitigate sharing grievances with sharing notebooks, computational platforms should offer features such as:

- Code-collapsing features, for visualisation emphasis
- Exporting to a diverse set of formats, such as presentations, if sharing the notebook itself is infeasible.
- Comment-only modes for feedback without cell execution.

5.7.2 Sharing supporting artefacts

Sharing supporting artefacts refers to the dissemination of the data and environment which influence the execution of a notebook, ensuring reproducibility and functionality. The data includes the raw data used for analysis and exploration, *and* the runtime session—variables, functions, and imports. The environment, on the other hand, refers to the versions of libraries, packages, and potentially the operating system, with which the notebook was run.

Sharing raw data would require the user to have database access. For databases which contain sensitive data, confidentiality would naturally become a concern. In addition, configuring an environment would require DevOps skills well-outside the aptitudes of most users (Chattopadhyay et al., 2020).

Platforms like Binder allow users to create custom computing environments using a GitHub repository (Binder, n.d.). Once provisioned, others can access the environment via the web, nullifying the need for any local software installations. Such a solution, however, is not natively-supported through many solutions. Consequently, considerable setup would be necessary for operation.

Alternatively, the sender can provide various auxiliary resources, such as requirements files, which programmatically assist the receiver by installing notebook-specific dependencies on their machine (Chattopadhyay et al., 2020). This approach, unfortunately, breaks the "single file" metaphor of notebooks (Chattopadhyay et al., 2020).

Computational platforms should provide integrated data- and environment-packaging, ensuring recipients can reproduce computation without manual setup. Importantly, the platform must do so while preserving the "single file" paradigm.

5.7.3 Editing collaboratively

Collaborative editing refers to multiple users working simultaneously on a single notebook, allowing real-time development and interaction, especially in remote settings. Despite being a powerful functionality, real-time collaboration isn't provisioned in many notebook environments (Chattopadhyay et al., 2020). Real-time editing requires non-trivial software engineering—complex synchronisation mechanisms that ensure changes from multiple users are propagated in real-time with great accuracy (Chattopadhyay et al., 2020).

Colab was among the first platforms to incorporate synchronous editing (Google, n.d.b). Still, it is important to note that synced notebooks introduce interference—users need to be more strategic when coordinating work (Wang et al., 2019). To avoid these issues, users just resort to completing work themselves and later incorporate this work into the notebook (Wang et al., 2019), ultimately defeating the purpose of synchronisation.

To ensure seamless remote development, a computational platform should provide real-time synced multi-user editing. A means to resolve merge conflicts would further enhance functionality.

5.8 Reproducing and reusing

Reproducibility refers to the ability to consistently recreate computational results, ensuring the software and underlying environment used remain identical in subsequent executions. Reproducing notebooks is pivotal to verifying how a result was computed or obtained. Nonetheless, architectural features, such as non-linear execution, and notebook–machine coupling, make achieving reproducibility non-trivial (Chattopadhyay et al., 2020).

Nextjournal provides the most direct solution to reproducibility. Using Docker containers—abstract boxes which bundle both code and dependencies—complete reproducibility can be achieved (Nextjournal, n.d.b). However, it is important to note that Docker itself still needs to be installed and configured on the recipient's machine. Depending on the specifics, this may be non-trivial. Users without the aptitudes needed to navigate such complexity are left unsupported.

To address reproducibility issues with notebooks, the baseline capability is an integrated environment management system, capturing and auto-configuring dependencies, software versions,

extensions, and other influential artefacts, ensuring consistent execution across different machines.

5.8.1 Reproducing

5.8.2 Reusing and adapting

The process of reusing and adapting pertains to leveraging existing notebook components in varied contexts. Yet, handling logic subtleties can prove frustrating, like when an "earlier notebook uses absolute paths" (Chattopadhyay et al., 2020). The lack of modularity associated with cells further harms reuse, "cells have no designated format or function" (Chattopadhyay et al., 2020). Finally, migrating intricate notebook and machine dependencies represents yet another hurdle (Chattopadhyay et al., 2020).

Cloud-based solutions, namely Azure Machine, Colab, and Sagemaker, use remote servers to provide virtual execution environments, decoupling the notebook from any underlying machine (Microsoft, n.d.a; Google, n.d.c; Amazon, n.d.b). Still, logic minor differences and modularity are still present. As such, users simply copy and paste code between notebooks (Chattopadhyay et al., 2020). Considering the intricacies of both notebook and environment dependencies, notebooks can easily end up rife with errors which need to resolved users can event execute their own code (Chattopadhyay et al., 2020).

To address challenges relating to reuse and adaptation, computational platforms should offer standardised modular cells, resolve small-scale logic discrepancies, or provide automated dependency management for seamless transfer and adaptation across various platforms and machines.

5.9 Deploying to production

5.9.1 Scheduling long-running computations

A long-running computation is simply a task which takes an extended period of time to execute, whether it be hours, days, or weeks. Typically, such computations halt all interactivity, providing no feedback on progress (Chattopadhyay et al., 2020). Recall that most platforms adopt a single-node architecture. In addition to hampering big data capabilities, as discussed in *Sub-section*

5.4.2: Scaling, this architecture hinders long-running computations. Running heavy tasks on a single node locks up considerable computational resources, preventing users from engaging in other tasks (Chattopadhyay et al., 2020).

Hosted platforms, like Azure Machine Learning and SageMaker, allow for notebook scheduling (mssaperla et al., 2023; Amazon, n.d.h). They can notify relevant users of task completions, failures, and interruptions. As such, users are unburdened with having to constantly monitor progress themselves. That said, leveraging such functionalities typically involves financial investment, as expected from such high-powered platforms. For students and hobbyists, these investments might quickly balloon, prohibiting continued adoption.

Platforms should introduce asynchronous execution for long-running computations, real-time progress feedback, and completion notifications, ensuring uninterrupted interactivity.

5.9.2 Packaging as a standalone artefact

Packaging refers to the process of converting notebook-sourced code into an executable unit, suitable for immediate deployment. In practice, this is wholly-infeasible—the exploratory nature of notebooks doesn't align with software engineering best practices (Chattopadhyay et al., 2020). Cell-based code authorship, while suitable for exploration, doesn't coincide with the use of functions and classes, hindering maintainability (Chattopadhyay et al., 2020). To make matters worse, while traditional scripts execute linearly, execution order for notebooks is typically unenforced, obscuring how data is transformed (Chattopadhyay et al., 2020).

Most platforms allow for the export of notebook code to script formats. A user can export their Jupyter Notebook to a Python script, translating its rich-text documentation to Python-compliant comments. While useful for trivial computations, linear-execution is still unenforced, hampering the reliability of the resultant script. As such, users in professional contexts often have to rewrite their code for integration into production pipelines (Chattopadhyay et al., 2020).

To provision a production-compliant packaging solution, a computational notebook should offer integrated tools for linearising cell execution, automatically refactoring code for production readiness, or exporting to modular, deployable formats without manual intervention.

Chapter 6

Discussion

6.1 Limitations

In this section, constraints and limitations encountered in *Yuna*'s development are discussed. These limitations, while posing challenges, also offer insights into potential areas for improvement and future introductions.

Loading data. *Yuna* currently does not facilitate the direct loading of local data. This limitation hinders seamless data integration, a critical feature for extensive data analysis.

Usability testing. *Yuna*'s development phase lacked structured usability testing. This absence of empirical user experience evaluation restricts our understanding of *Yuna*'s performance in-situ and how to best augment the user experience.

Serialising. *Yuna*'s object serialisation, reliant on cloudpickle, is more effective than an implementation using the standard pickle package (antoine et al., 2023). However, dill is the state-of-the-art—offering advanced functionalities such as saving an entire session (McKerns, 2023). Unfortunately, since dill is not supported by Pyodide, its advanced serialisation capabilities will remain untapped until it is officially-supported.

Visualising. The current version of *Yuna* does not support data visualisation. Although visualisation packages can be installed into the runtime, front-end development is still necessary in order to render them.

Minimising environment sizes. Remarkably, a standard environment has a file size of only 28 MB. Environments embedded with multiple packages, however, can be excessively large, easily exceeding 100 MB. However, if users can be saved from "dependency hell" (Chattopadhyay et al.,

2020), a larger file size—a natural byproduct of self-containment—is certainly justifiable. With version control and cloud-based storage solutions, navigating this limitation becomes trivial.

Maintaining software. The manual embedding of core software—Pyodide (Pyodide, n.d.a), Marked (Marked, n.d.), CodeMirror (CodeMirror, n.d.), and Bootstrap (Bootstrap, n.d.)—has introduced significant complexity to the build process, complicating updates and maintenance.

Exporting. *Yuna* currently lacks the ability to export notebook contents into varied formats, namely Python scripts or Markdown documents. As such, versatility in varied academic and professional contexts is currently restricted.

6.2 Future work

Loading data. Future versions of *Yuna* will integrate with the File System API (Mozilla, 2023b) to enable direct loading of local data. Furthermore, special attention will be given to embedding small local files to facilitate seamless data integration while managing environment sizes.

Usability testing. Comprehensive usability studies will be conducted to assess *Yuna*'s user experience. These studies will aim to gather user feedback in real-world scenarios, enabling the development of more intuitive and effective features.

Serialising. Upon availability of dill support, the package will be integrated into *Yuna*, broadening serialisation capabilities and allowing for the more straightforward saving and restoration of session data.

Visualising. Efforts will be directed towards the development of a robust visualisation component, enabling users to render and interact with visualisations directly within the interface.

Minimising environment sizes. Implementing either Base85 or ASCII85 encodings will be explored in an attempt to reduce file sizes (Python, 2023), particularly for environments with multiple embedded packages. This will help in maintaining the balance between functionality and storage efficiency.

Maintaining software. Automating the embedding process of core software components is a key focus. This will involve maintaining source files in a centralised repository and employing automated scripts for embedding, ensuring straightforward updates and maintenance.

Exporting. Enhancing *Yuna* with the ability to export notebooks to Python scripts or Markdown documents is planned. This feature will include enforcing linear execution of code cells when exporting, aiding in the transition of notebook code to robust production-ready scripts.

Chapter 7

Conclusion

This dissertation introduced *Yuna*, an innovative, self-contained computational platform, crafted to address multiple pain points identified in current solutions. *Yuna*'s detailed development process, covering aspects ranging from user interface design to session management, has been meticulously chronicled. Through comparison to leading platforms, *Yuna* demonstrates its proficiency in enhancing key operational areas—namely code management, reliability, archiving, collaborative work, and reproducibility. This dissertation also candidly addressed *Yuna*'s current limitations, shedding light on future development trajectories. In conclusion, *Yuna* epitomises a pivotal advancement in the realm of computational platforms, standing as a beacon for democratising access and simplifying the user experience, lowering barriers to entry.

Bibliography

538. 2023. data. *538*.

538. n.d. Home. *538*.

Aashita. 2023. Beginners Guide to Databricks: Batch Processing and Streaming. *Databricks*.

Abelniak. 2011. Mathcad 15.0. *Wikimedia Commons*.

Amazon. n.d.a. Amazon S3. *Amazon*.

Amazon. n.d.b. Amazon SageMaker. *Amazon*.

Amazon. n.d.c. Amazon SageMaker Ground Truth Customers. *Amazon*.

Amazon. n.d.d. Analyze and debug anomalies. For example, training a neural network will cease if gradients are determined to be vanishing. SageMaker Debugger identifies vanishing gradients so you can remediate before training is impacted. *Amazon*.

Amazon. n.d.e. K-Means Algorithm. *Amazon*.

Amazon. n.d.f. Linear Learner Algorithm. *Amazon*.

Amazon. n.d.g. Perform Automatic Model Tuning with SageMaker. *Amazon*.

Amazon. n.d.h. Schedule a notebook job. *Amazon*.

Amazon. n.d.i. Use an Amazon S3 bucket for input and output. *Amazon*.

Amazon. n.d.j. Use Apache MXNet with Amazon SageMaker. *Amazon*.

Amazon. n.d.k. Use Hugging Face with Amazon SageMaker. *Amazon*.

Amazon. n.d.l. Use Reinforcement Learning with Amazon SageMaker. *Amazon*.

Amazon. n.d.m. Use TensorFlow with Amazon SageMaker. *Amazon*.

andrejv and peterpall. 2019. wxMaxima: Cross platform GUI for Maxima. *SourceForge*.

Thomas Ansart. 2023. Visualising absolute and anomaly temperature. *Observable*.

antoine, holdenk, hyukjinkwon, Kyle.Kelley, lllllllll, mrocklin, ogrisel, pierreglaser, and ssanderson. 2023. cloudpickle. *Python Package Index*.

Apache. n.d. Apache Hadoop. *Apache*.

Apache Spark. n.d. Spark Research. *Apache Spark*.

Pedro Barquinha. 2020. Unemployment Exploration. *Observable*.

Belgian Neuroinformatics. 2019. GSoC 2019 completed successfully. *Belgian Neuroinformatics*.

Iheagwara Ifeanyi Benedict. n.d. Data Analytics with Pandas. *Deepnote*.

Binder. n.d. Home. *Binder*.

Bootstrap. n.d. Home. *Bootstrap*.

brettcannon, dstufft, pf_moore, and pradyunsg. 2023. packaging. *Python Package Index*.

Dan Bricklin. n.d. Steve Jobs talking about VisiCalc in 1996. *Bricklin*.

Byte. 1989. Ten Years of Rows and Columns. *Byte*.

cdhowe. 2021. The RStudio IDE is a free and open source integrated development environment (IDE) for R. It helps programmers develop software written in R, Python, and several other languages. *Wikimedia Commons*.

Centers for Disease Control and Prevention. 2020. Race, Ethnicity, and Age Trends in Persons Who Died from COVID-19—United States, May-August 2020. *Morbidity and Mortality Weekly Report*.

Souti Chattopadhyay, Ishita Prasad, Austin Z. Henley, Anita Sarma, and Titus Barik. 2020. What's wrong with computational notebooks? Pain Points, Needs, and Design Opportunities. In *Proceedings: 2020 Conference on Human Factors in Computing Systems*.

Norman Chonacky and David Winch. 2005. Reviews of Maple, Mathematica, and MATLAB: Coming Soon to a Publication Near You. *Computing in Science and Engineering*.

CodeMirror. 2023. Release notes and version history. *CodeMirror*.

CodeMirror. n.d. CodeMirror 5. *CodeMirror*.

Computer History Museum. n.d. 1970s-early 1980s. *Computer History Museum*.

Crunchbase. n.d. Deepnote. *Crunchbase*.

D3.js. n.d. The JavaScript library for bespoke data visualization. *D3.js*.

Databricks. 2-23. TensorBoard. *Databricks*.

Databricks. 2022. Databricks | Notebook Development Overview. *YouTube*.

Databricks. 2023a. Collaborate using Databricks notebooks. *Databricks*.

Databricks. 2023b. Databricks for Python developers. *Databricks*.

Databricks. 2023c. Databricks for R developers. *Databricks*.

Databricks. 2023d. Databricks for Scala developers. *Databricks*.

Databricks. 2023e. MLflow guide. *Databricks*.

Databricks. 2023f. Search for workspace objects. *Databricks*.

Databricks. 2023g. SQL language reference. *Databricks*.

Databricks. 2023h. What is Delta Lake? *Databricks*.

Databricks. n.d.a. AI and Machine Learning. *Databricks*.

Databricks. n.d.b. Customer Story: Comcast. *Databricks*.

Databricks. n.d.c. Data Streaming. *Databricks*.

Databricks. n.d.d. Databricks is the data and AI company. *Databricks*.

Databricks. n.d.e. Migrate from Hadoop. *Databricks*.

Databricks. n.d.f. The best data platform is a lakehouse. *Databricks*.

Deepnote. n.d.a. Amazon S3. *Deepnote*.

Deepnote. n.d.b. Comments. *Deepnote*.

Deepnote. n.d.c. Deepnote: Analytics and data science notebook for teams. *Deepnote*.

Deepnote. n.d.d. PostgreSQL. *Deepnote*.

Deepnote. n.d.e. Real-time collaboration. *Deepnote*.

Digital Engineering. 2006. PTC to Acquire Microsoft. *Digital Engineering*.

Duke University. n.d. Getting started with Python and the IPython notebook. *Duke University*.

European Space Agency. 2017. ESA's Mars Express Operations Dataset. *Kaggle*.

Dave Gershgorn. 2017. Nerds rejoice: Google just released its internal tool to collaborate on AI. *Quartz*.

Kenneth N. Gilpin and Todd S. Purdum. 1985. Former Friendly Rivals Joining Forces at Lotus. *New York Times*.

Git. n.d. Home. *Git*.

github-actions. 2023. v5.0.2. *GitHub*.

Google. n.d.a. Choose the Colab plan that's right for you. *Google*.

Google. n.d.b. Google Drive. *Google*.

Google. n.d.c. Welcome to Colab! *Google*.

Gortu. 2005. Screenshot of VisiCalc running on an Apple II computer. *Wikimedia Commons*.

Alena Guzharina. 2023. How to Work With Git in Datalore. *The JetBrains Datalore Blog*.

Thomas Haigh. 2008. Clive Moler: Mathematical Software Pioneer and Creator of MATLAB. *IEEE Annals of the History of Computing*.

Tim Harford. 2019. How computing's first 'killer app' changed everything. *BBC*.

HolyCookie. 2011. The image shows the Mathematica 8.0.0 (GNU/Linux) frontend, in which the bifurcation diagram of the logistic map is calculated. *Wikimedia Commons*.

Honeywell. n.d. Home. *Honeywell*.

Tom Hormby. 2006. VisiCalc and the Rise of the Apple II. *LowEndMac*.

INCF. n.d. INCF: Standards and Best Practices Organisation for open and FAIR neuroscience. *International Neuroinformatics Coordinating Facility*.

IPython. 2023. Built-in magics. *IPython*.

JetBrains. 2018. Datalore 1.0: Intelligent Web Application for Data Analysis. *JetBrains*.

JetBrains. 2022. Sharing and collaboration. *JetBrains*.

JetBrains. 2023a. Coding assistance. *JetBrains*.

JetBrains. 2023b. Notebook search. *JetBrains*.

JetBrains. n.d.a. Customer Stories. *JetBrains*.

JetBrains. n.d.b. Datalore Demo notebook. *JetBrains*.

JetBrains. n.d.c. Datalore for Teams. *JetBrains*.

JetBrains. n.d.d. How LINE adopted Datalore as a new data analysis platform. *JetBrains*.

JetBrains. n.d.e. Notebooks. *JetBrains*.

Ben Johnston. 2020. Google Colab. *Princeton University*.

Jakub Jurovych. 2022. Deepnote raises a $20M series A led by Accel & Index. *Deepnote*.

Kaggle. n.d.a. How to Use Kaggle: Notebooks. *Kaggle*.

Kaggle. n.d.b. How to Use Kaggle: Tensor Processing Units (TPUs. *Kaggle*.

Kaggle. n.d.c. TensorFlow Guide. *Kaggle*.

kangar dev. n.d. Creating Tensors. *Deepnote*.

Donald Knuth. 1984. Literate Programming. *The Computer Journal*.

Robert Kosara, Allison Horst, and Paul Buffa. 2018. Learning Observable: Reactive dataflow. *Observable*.

Holger Krekel and pytest. n.d. pytest. *pytest*.

Sam Lau, Ian Drosos, Julia M. Markel, and Philip J. Guo. 2020. The design space of computational notebooks: An analysis of 60 systems in academia and industry. In *Proceedings: 2020 IEEE Symposium on Visual Languages and Human-Centric Computing*.

leornard. 2023. beautifulsoup4. *Python Package Index*.

Fei-Fei Li. 2017. Welcome Kaggle to Google Cloud. *Google Cloud*.

Claudio Maccone. 2012. *Mathematical SETI: Statistics, Signal Processing, Space Missions*. Springer Berlin Heidelberg.

Macsyma. 1998. *Introduction to Macsyma*. Macsyma.

Makeability Lab. n.d. Jupyter Notebook screenshot showing an analysis and visualization of a 3-axis accelerometer to infer step counts. *Makeability Lab*.

Maurice M. Manning. 2016. IPython notebook available to ease data transfer between Savio and Box. *University of California, Berkeley*.

Maplesoft. 2010. *Maple User Manual*. Maplesoft.

Maplesoft. n.d.a. 2-D Math. *Maplesoft*.

Maplesoft. n.d.b. About. *Maplesoft*.

Maplesoft. n.d.c. Animation. *Maplesoft*.

Maplesoft. n.d.d. MATLAB Connectivity. *Maplesoft*.

Maplesoft. n.d.e. Overview of the Python Package. *Maplesoft*.

Maplesoft. n.d.f. Physics. *Maplesoft*.

Maplesoft. n.d.g. Signal Processing. *Maplesoft*.

Maplesoft. n.d.h. Statistics and Data Analysis. *Maplesoft*.

Pedro Marcelino. 2022. Comprehensive data exploration with Python. *Kaggle*.

Marked. n.d. Home. *Marked*.

Philippa Markovics, Micah P. Dombrowski, Kleopatra Pirpinia, David Schmudde, and Andrea Amantini. 2021. Working with Code. *Nextjournal*.

MathWorks. 2023a. C++ with MATLAB. *MathWorks*.

MathWorks. 2023b. FORTRAN with MATLAB. *MathWorks*.

MathWorks. 2023c. Get Started with Control System Toolbox. *MathWorks*.

MathWorks. 2023d. Get Started with Signal Processing Toolbox. *MathWorks*.

MathWorks. 2023e. Get Started with Statistics and Machine Learning Toolbox. *MathWorks*.

MathWorks. 2023f. Java with MATLAB. *MathWorks*.

MathWorks. 2023g. Matrices and Arrays. *MathWorks*.

MathWorks. n.d.a. Math. Graphics. Programming. *MathWorks*.

MathWorks. n.d.b. MATLAB and Simulink Help Toyota Design for the Future. *MathWorks*.

MATLAB. 2008. New and Updated Desktop Features in R2008b. *MATLAB*.

Matplotlib. n.d. Image tutorial. *Matplotlib*.

Maxima. n.d.a. Home. *Maxima*.
Maxima. n.d.b. Introduction to distrib. *Maxima*.
Maxima. n.d.c. Introduction to draw. *Maxima*.
Maxima. n.d.d. Introduction to Finance. *Maxima*.
Maxima. n.d.e. Introduction to pytranslate. *Maxima*.
Maxima. n.d.f. Maxima and Lisp. *Maxima*.
Michael McKerns. 2023. dill. *Python Package Index*.
Melody Meckfessel. 2021. Introducing Observable Collaboration. *Observable*.
Microsoft. 2009. The History of Microsoft – 1985. *Microsoft*.
Microsoft. 2013. Tempe. *Microsoft*.
Microsoft. 2023a. Manage access to an Azure Machine Learning workspace. *Microsoft*.
Microsoft. 2023b. Train PyTorch models at scale with Azure Machine Learning. *Microsoft*.
Microsoft. 2023c. Train scikit-learn models at scale with Azure Machine Learning. *Microsoft*.
Microsoft. 2023d. Train TensorFlow models at scale with Azure Machine Learning. *Microsoft*.
Microsoft. n.d.a. Azure Machine Learning. *Microsoft*.
Microsoft. n.d.b. IntelliSense. *Microsoft*.
Microsoft. n.d.c. Progressive gives voice to Flo's chatbot, and it's as no-nonsense and reassuring as she is. *Microsoft*.
Microsoft. n.d.d. Refactoring. *Microsoft*.
Microsoft. n.d.e. Visual Studio Code. *Microsoft*.
Microsoft Azure. 2018. Microsoft Azure: The only consistent, comprehensive hybrid cloud. *Microsoft*.
Ron Miller. 2015. Microsoft Officially Launches Azure Machine Learning Platform. *TechCrunch*.
Ron Miller. 2017. AWS releases SageMaker to make it easier to build and deploy machine learning models. *TechCrunch*.
Clive Moler. 1965. *Finite difference methods for the eigenvalues of Laplace's operator*. Ph.D. thesis, Stanford University.
Joel Moses. 2012. Macsyma: A personal history. *Journal of Symbolic Computation*.
Mozilla. 2023a. Base64. *Mozilla*.
Mozilla. 2023b. File System API. *Mozilla*.
mssaperla, docs-preview bot, irinaskaya, jaseidman, and mariesalet. 2023. Create and manage scheduled notebook jobs. *Microsoft*.
Nextjournal. n.d.a. Abour nextjournal. *Nextjournal*.
Nextjournal. n.d.b. The notebook for reproducible research. *Nextjournal*.
NumPy. n.d. NumPy: The Absolute Basics for Beginners. *NumPy*.
Observable. 2023. Embedding. *Observable*.
Observable. n.d.a. Our mission and our values. *Observable*.
Observable. n.d.b. The future of data visualisation starts here. *Observable*.

Omegatron. 2008. A screenshot of WxMaxima 0.7.1 running in Ubuntu Linux (Clearlooks theme with Tango icons) with some of the standard screenshot equations and a plotting window open to demonstrate the newer features and look of Maxima. *Wikimedia Commons*.

OpenRefine. n.d. Home. *OpenRefine*.

pandas. n.d. Intro to Data Structures. *pandas*.

Richard Pavelle and Paul S. Wang. 1985. MACSYMA from F to Gt. *Journal of Symbolic Computation*.

PC Mag. 1982. Apple Charts the Course for IBM. *PC Mag*.

PC Mag. 1987. Now you can calculate on your PC with the same freedom you have on paper. *PC Mag*.

Fernando Perez. 2012. IPython Notebook: A Historical Restrospective. *Fernando Perez Blog*.

Fernando Perez. 2014. Project Jupyter. *Speaker Deck*.

piro. 2023. psycopg. *Python Package Index*.

Posit. 2023a. Friction free data science. *Posit*.

Posit. 2023b. Version Control. *Posit*.

Posit Software. n.d. RStudio Desktop. *Posit Software*.

Project Jupyter. n.d.a. Debugger. *Project Jupyter*.

Project Jupyter. n.d.b. Jupyter: Free software, open standards, and web services for interactive computing across all programming languages. *Project Jupyter*.

Project Jupyter. n.d.c. Table of Contents. *Project Jupyter*.

PTC. n.d.a. Component Object Model. *PTC*.

PTC. n.d.b. Data Analysis Functions. *PTC*.

PTC. n.d.c. Finance Functions. *PTC*.

PTC. n.d.d. Five Companies Crushing it with PTC Mathcad. *PTC*.

PTC. n.d.e. Image Processing Functions. *PTC*.

PTC. n.d.f. Mathcad: Math Software for Engineering Calculations. *PTC*.

PTC. n.d.g. The Symbolic Engine. *PTC*.

PTC. n.d.h. To Enter Math Expressions. *PTC*.

Pyodide. n.d.a. Home. *Pyodide*.

Pyodide. n.d.b. micropip. *Pyodide*.

pyodide-ci bot. 2023. 0.23.2. *GitHub*.

Python. 2023. base64—Base16, Base32, Base64, Base85 Data Encodings. *Python*.

Qubole. n.d.a. Accessing the logs for jupyter notebooks. *Qubole*.

Qubole. n.d.b. Home. *Qubole*.

Qubole. n.d.c. Pricing. *Qubole*.

RStudio. n.d.a. Easy web applications in R. *RStudio*.

RStudio. n.d.b. Python. *RStudio*.

RStudio. n.d.c. R Markdown. *RStudio*.

RStudio. n.d.d. RStudio Open-Source Packages. *RStudio*.

RStudio Team. 2011. RStudio, a new open-source IDE for R. *Posit Software*.

Joshua Sierles. 2017. Launch: Nextjournal Public Beta for Open Research. *Medium*.

Symbolic Computation Group. 2012. History of Maple Computer Algebra Software. *Symbolic Computation Group*.

Gabor Szalai. 2023. Smarter, faster, better: introducing interactive exploration & other power-ups to deepnote charts. *Deepnote*.

Tableau. n.d. Home. *Tableau*.

University of California, Berkley. 1996. *Maxima User's Guide*. University of California, Berkley.

Kevin Ushey. 2023. Code Completion in the RStudio IDE. *Posit Support*.

Santiago Valdarrama. n.d. A Scikit-Learn Pipeline. *Deepnote*.

Linda Vu. 2021. Project Jupyter: A Computer Code that Transformed Science. *Berkeley Lab Computing Sciences*.

Alexander F. Walz. 1998. History of Maple. *University of Texas, San Antonio*.

April Yi Wang, Anant Mittal, Christopher Brooks, and Steve Oney. 2019. How Data Scientists Use Computational Notebooks for Real-time Collaboration. *Proceedings of the ACM on Human-Computer Interaction*.

WebAssembly. n.d. Home. *WebAssembly*.

Eric Wendt. 2023. Explore, collaborate, share: how Webflow optimizes data workflows. *Deepnote*.

Wolfram. n.d.a. A Comprehensive Solution: Developing an Enterprise-Wide Platform with Wolfram Technologies. *Wolfram*.

Wolfram. n.d.b. C/C++ Interface. *Wolfram*.

Wolfram. n.d.c. Image Processing. *Wolfram*.

Wolfram. n.d.d. Java Interface. *Wolfram*.

Wolfram. n.d.e. Machine Learning. *Wolfram*.

Wolfram. n.d.f. Manipulate. *Wolfram*.

Wolfram. n.d.g. R Interface. *Wolfram*.

Wolfram. n.d.h. Wolfram Mathematica: The world's definitive system for modern technical computing. *Wolfram*.

Stephen Wolfram. 2008. Mathematica Turns 20 Today. *Wolfram*.

Stephen Wolfram. 2013. There was a Time before Mathematica... *Stephen Wolfram – Writings*.

XhmikosR. 2023. v5.3.0-alpha3. *GitHub*.