Bachelor's Thesis Computing Science

Radboud University Nijmegen

A SaC plotting visualisation integration in Jupyter

Exploring plotting of the compiled language SaC in the Jupyter interactive environment

Author:
Luc Schrauwen
s1036102

First supervisor/assessor:
Prof. Sven-Bodo Scholz

Second assessor:
dr. Peter Achten

January 26, 2024

Abstract

The goal of this research is a case study of implementing plotting visualisations for the compiled array programming language SaC in the Jupyter interactive environment. The present insight into SaC arrays provides only text-based output that provides little insight into data. The goal is reached by building on top of an existing SaC kernel written in Python that is in between the users Jupyter notebook files and the SaC compiler. To get the most versatile plotting solution, the Python part of the kernel is exploited by using the existing Matplotlib Python library. This library is interfaced to work with SaC variables and to be included in the Jupyter notebook syntax. The resulting lightweight implementation in the kernel has the same possibilities and look as if it was used on the Python programming language being able to plot a wealth of 3D and 2D plots.

Contents

1 Introduction **2**

2 Background **4**
2.1 The importance of plotting 4
2.2 Single assignment C 5
2.3 Jupyter project 6
2.4 Jupyter project architecture 9
2.5 Jupyter extension possibilities 10
2.5.1 SaC Jupyter kernel 11
2.5.2 Jupyter extensions 14
2.6 Matplotlib Python plotting library 15

3 Plotting SaC in Jupyter **17**
3.1 Approach 18
3.2 SaC plotting execution cycle 21
3.3 Visualisation example 22

4 Discussion **24**
4.1 Scope 24
4.2 Future work 24

5 Conclusion **26**

6 Acknowledgements **27**

A Appendix **31**

1. Introduction

As time goes on, more and more convenient tools are created to aid software development; to simplify and to speed up the development process by adding abstraction and visualisations. Countless libraries exist showing this goal of allowing programmers to focus more on program structure instead of small details and keep up with the rising complexity of programs. Originally, languages used the edit-compile-execute cycle, but with the introduction of interpreted languages such as Python, the benefit of rapid algorithmic exploration, prototyping, and visualisations became part of daily scientific work. Interpreted languages can be used in interactive environment that let scientists investigate, test new ideas, combine algorithmic approaches, and evaluate the outcome of expressions directly without the compilation of a whole program [18]. Well-known interactive environments of this kind are Matlab [10], Mathematica [11] and Jupyter [26]. Jupyter got especially popular for data scientists because of its laboratory simple notebook style, a plethora of possibilities, accessible to multiple systems and programming languages and good for easier data exploration [4]. Jupyter supports the interpreted language Python by default and is used for interactive development by splitting code into cells with each having its own output. These cells can also hold interpreted text and images, creating a very accessible learning and representing tool. This way of development is not readily available to compiled languages because of the full program compilation that is needed to run code. This usually gets solved by creating an interpreter for the compiled language such as Cling [29], developed for C.

This research is a case study of implementing plotting visualisations for a compiled language in an interactive environment. Jupyter is used for this because of its extendable nature that allows its framework to be used by other programming languages via programs called kernels (explained in section 2.5.1). The language this research focuses on is Single Assignment C (SaC) [21], which is a functional array programming language suitable for data, image and signal processing. This language already has support for creating PDF images but no support for plotting of any kind. This language

is chosen because it already has a bare kernel integration into Jupyter [27]. Plotting would be great for SaC because of the limited current ASCII output the language has and the plotting will be a great tool to easily visualise and subsequently better understand numerical data [30]. This is especially true for the abstract concepts and operations in the field of computer science [16]. The current kernel is not an interpreter but mimics to be one while still using the SaC compiler and its compilation cycle. The kernel currently is useful for quick prototyping, experimentation and demonstrations with the language. The plotting functionality will only enhance these benefits by giving even more insight into SaCs behaviour. SaC is also taught at the Radboud University to Master students and plots can aid in learning SaC and the array programming paradigm in the same way it aids experimentation and demonstration. The problem with the SaC kernel in the context of plotting is that it only uses communication holding ASCII streams to connect with SaC and Jupyter. In this research the following questions are answered; How can plots be created for SaC in a way that is convenient to program and use? How can a plot be transported from the plotting library to output in Jupyter? And how can the user have as much options while plotting?

In this paper the important components and systems used in this research are explained in the background (section 2), then the projects structure and final implementation will be explained, including an example (section 3 and appendix A). Finally some remarks about future work and improvements are given in section 4.

2. Background

This chapter provides background information for the libraries and software used in this research to further understand the research and its design. It starts by explaining the importance of plotting in the context of programming using an example (section 2.1) followed by the explanation of the programming language Single assignment C (SaC) for which this research provides plotting functionalities (section 2.2). Here the motivation of using SaC will also be clarified. After this the Jupyter project is explained (section 2.3) with a top down view of its architecture (section 2.4) and the two ways Jupyter can be extended or used by other programming languages (section 2.5). Lastly the Python Matplotlib plotting library used in this research is explained in section 2.6.

2.1 The importance of plotting

As mentioned in the introduction, plotting is one of the best ways to easily visualise and inspect numerical data for patterns, outliers or structure, especially for large collections of data [30]. To further illustrate the importance of plotting, the simple example of the convex hull algorithm is used. This algorithm tries, given a set of coordinates, to find the convex hull. The convex hull of a set of points in an Euclidean space is the smallest convex polygon that encloses all of the points. When implementing this algorithm in SaC, the only way of checking the correctness of this algorithm on a set of points is by looking at the print output of the array that SaC can provide. This would give almost no usable insight into the correctness of the computed solution, especially for large arrays of coordinates. Plotting these points into a scatter plot will instantly make it clear how the points in the set are distributed in 2D space. Even better, highlighting or even connecting the convex hull in this plot can instantly show whether this convex hull is valid. Both an ASCII way of printing arrays and a plot printing the same coordinates can be seen in figure 2.1. This example illustrates that the interpretation of data is helped by visualisations.

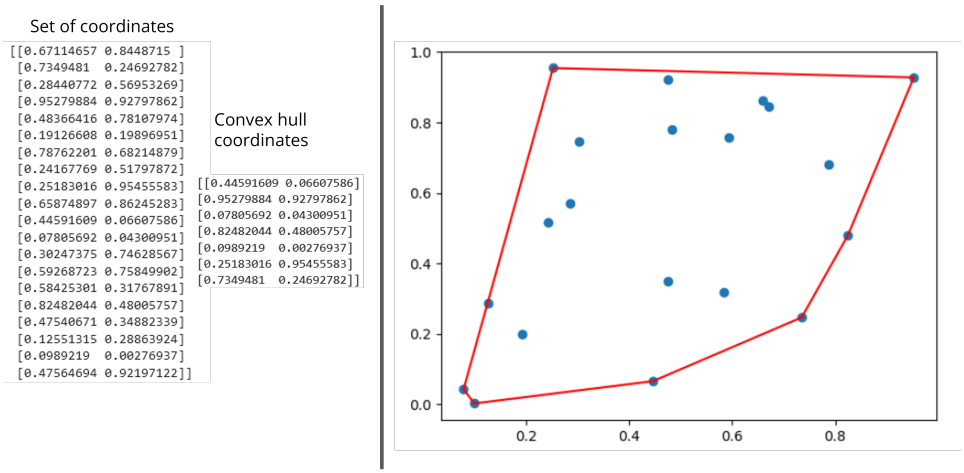

Figure 2.1: Showing text-based representation of an array with coordinates (right) and plot image of the same array (left) for the convex hull problem on an arbitrary list of 20 coordinates. (Example made with Matplotlib in Python)

2.2 Single assignment C

The programming language Single assignment C (SaC) is used in this research to add plotting capabilities to. It is a functional array programming language that prioritises speed and facilitates the compilation for non-sequential program execution in multiprocessor environments [21]. Instead of programmers manually having to program the low-level hardware instructions, it offers a high-level imperative C-like programming notation (seen in listing 1) that adapts to the underlying hardware to maximise efficiency and reduce execution time. An array programming language deals with generalised operations on sequences of values while functional programming languages map values to other values, rather than a sequence of imperative statements which update the running state of the program [8][13]. Most of the operations that are found as standard operations on scalars in other languages are applicable to entire arrays in SaC. Functional languages tend to be better at concurrency and parallel programming. SaCs syntax does not look like a functional programming language, but the for-loops in SaC are just syntactic and are actually translated to tail-end recursive calls. SaC also uses a dynamic type system that assigns data types to variables during compilation. SaC is originally meant for image and signal processing, but is also suitable for data mining and modelling or in applications that need to perform operations on large data structures [21]. This makes the plotting visualisations ideal for SaC because of the large amount of data it will handle and thus making visualisations of arrays useful in interpreting said

data.

```
use StdIO: all;
use Array: all;

int main()
{
    a = iota(40);
    b = modarray(a, [0], 9);
    print(b);

    return 0;
}
```

Listing 1: SaC syntax example that creates an array `a` of size 40 (counting up from 0 till 39) and then creates an array `b` that is the exact array of `a` with the value at index 0 be replaced by the value 9

2.3 Jupyter project

The interactive environment used to add plotting visualisations to SaC in this research is Jupyter. The Jupyter project (formerly called IPython) is a non-profit, open source project that aims to provide an enhanced interactive environment for data visualisation and parallel computation. IPython was launched in 2001 and solely focused on the Python programming language, extending its interactive capabilities [18].

In 2014 the project evolved to the Jupyter project to support all programming languages using the browser-based Jupyter notebook interface it is still today [26]. The name Jupyter originates from the three core supported languages in the project, which are Julia [5], Python [28] and R [19]. The Jupyter project uses an underlying file format called notebooks that use the JavaScript Object Notation (JSON) syntax with the `.ipynb` filename extension. These notebooks contain an ordered list of input/output cells that are not restricted to just code and can also contain Markdown interpreted text and images. Around 2015 the Jupyter team made the improved next generation notebook interface called Jupyterlab, that still uses the same notebook files but instead allows for multiple files to be open in a single browser tab and provides more support for interactive elements.

The Jupyter notebook files can be opened and edited in the Jupyter notebook environment that runs in a browser. Every cell in the notebook has, after writing an expression or piece of code in it, its own output and

can use all results from previously executed cells. An example notebook (`SinDemo.ipynb`) can be seen in figure 2.2, that is made in Jupyterlab. Here the sine and cosine are plotted in a single plot using the Matplotlib Python library that is explained in section 2.6. This notebook contains three cells with the first being Markdown interpreted text. The second one used to import both libraries needed to plot and a Python print expression printing "Hello world!". The last one plots the cosine and sine using the Matplotlib plotting library. The printing of the text "Hello world!" has been included into this example to illustrate the multiple points of getting different output in a single Jupyter notebook.

A core feature of the Jupyter project is its extendable nature, allowing for other programming languages to use the framework. Jupyter has two paths of extending the framework. The first way is by creating kernels (see 2.5.1), that are used for evaluation of the notebook cells contents. These kernels usually hold the programming language's interpreter. The second way is changing the browser interface and features that are called Jupyter extensions (see 2.5.2). Both ways are considered in this research. Existing projects that use Jupyter to extend languages' capabilities is for example for one of the core languages Julia [5]. A list of existing third-party kernels supporting different languages has been created and published on Github [3].

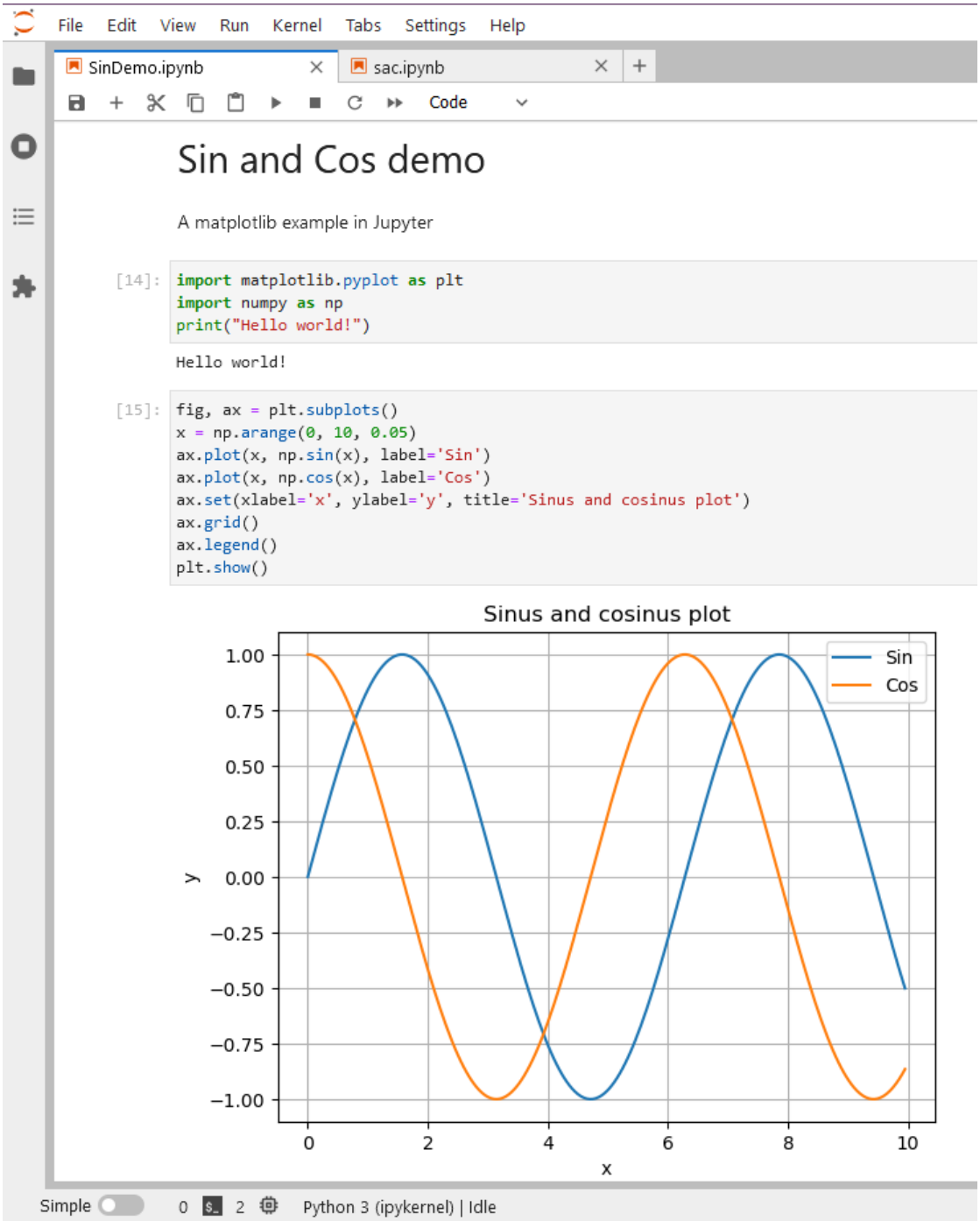

Figure 2.2: Example of a Jupyter notebook connected to the default Python kernel in a browser plotting the sine and cosine using Matplotlib

2.4 Jupyter project architecture

To understand the different ways of extending Jupyter, the architecture of the Jupyter project needs to be addressed. The architecture of the Jupyter program, as seen in figure 2.3, uses the Jupyter server as a central communication and control hub. The browser on the left is the user's interface to its notebooks and communicates via the Jupyter server to access the notebooks and the kernel. One of the ways to extend Jupyter are the extensions mentioned above, these are installed inside of the browser's Jupyter interface. The kernel is a piece of software that acts as the interpreter for the expressions in the Jupyter notebook cells. Most kernels hold interpreters like Cling for C [29] or the Python interpreter by default. When a user wants to execute a cell from a notebook in the browser, the Jupyter server will send the cell's expression to the kernel to be evaluated and return the output to the user's browser. Because of the central location of the Jupyter server and its architecture the kernel does not know anything about the notebook and will only get expressions from the server to evaluate, making the connection of multiple servers on one kernel possible as well as being able to use notebooks for all Jupyter and Jupyterlab versions. If no kernel is connected and cell expressions will not be able to be evaluated by a kernel, the Jupyter server still enables editing of notebooks.

Kernels communicate with the Jupyter server via socket-based messaging protocols using JavaScript Object Notation (JSON) sent over the ZeroMQ sockets [2]. This is an asynchronous messaging library, that can run without a dedicated message broker. A message broker is a program that serves as an intermediary between applications. Developers can use these protocols in kernels via a lightweight interface that can be wrapped in Python. For this research the most important communication socket is the IOPub broadcast channel. This channel is used by the kernel to publish all side effects such as standard output and input that should be displayed in the notebook. The current SaC kernels communication as seen in figure 2.3 are further explained in section 2.5.1.

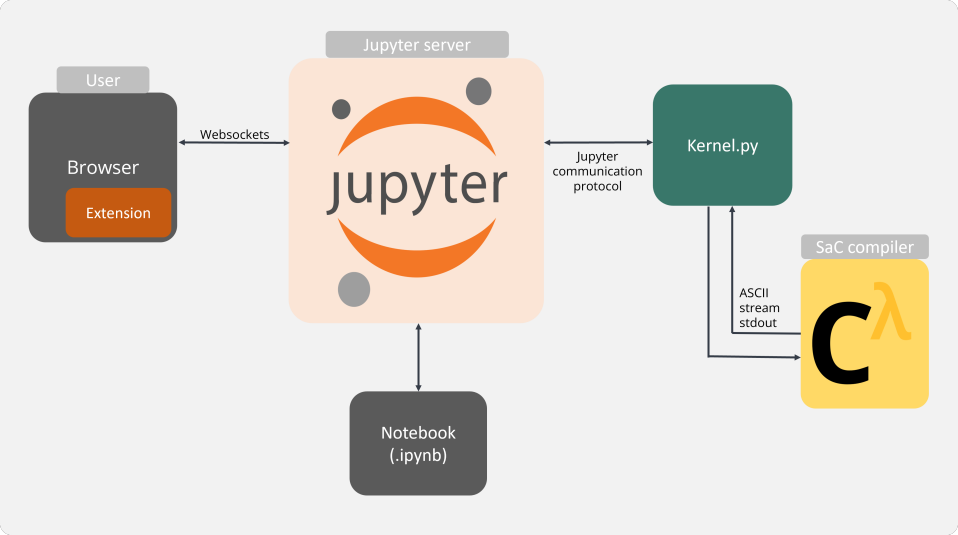

Figure 2.3: Jupyters architecture diagram including the SaC kernel and compiler

2.5 Jupyter extension possibilities

The Jupyter project allows for additions and extensions by developers. In this section both ways are further explained as mentioned before. Firstly, there are the kernels that are mainly used to connect a language to Jupyter by holding an interpreter. By default the Jupyter project connects to the Python interpreter kernel. A kernel however does not need to be a language interpreter but can also be used for widgets or other forms of abstractions that can be achieved via the cells in the notebook. For example, a kernel that can output sophisticated plots in just one line of code. Secondly, extensions can be created that are additions to the Jupyter browser interface and are mainly used to build on top of kernel functionality. They will alter how the browser interface of Jupyterlab/notebook works and looks by adding for example, commands, new widgets, buttons, key shortcuts or a variable explorer. These extensions build upon Jupyter libraries and packages that each have their own use-case, for example nbconvert [17] for exporting notebooks into different formats or Lumino [14], used for widgets, layouts, events, and data structures. Extensions are also easily distributed and downloadable by other users.

2.5.1 SaC Jupyter kernel

There exist three ways to connect a kernel to Jupyter. Firstly, using the wrapper kernels that reuse the communications from IPykernel, and implement only the core execution part. Usually these kernels hold the interpreters for programming languages. Secondly, by using native kernels that are written in a different language than Python and implement the execution and communications with the Jupyter server in the target language. And lastly, by using the Xeus kernels that use the Xeus library wrapper [31] that does not depend on a Python runtime.

The current SaC kernel [27] is a wrapper kernel that does not hold an interpreter, but it passes expressions from the Jupyter server to the SaC compiler and back. This kernel was originally made for easy showcasing the language and its behaviour without the manual compile-execute cycle. Because of the position that kernels usually have of being in between an interpreter and the Jupyter server, kernels can also create functions that are not directly tied to the kernels interpreter language. These functions are called *magics* and are used to get functionalities in the users notebook that is only usable in Jupyter in combination with the kernel. An example of this is to have a distinct function recognised by the kernel that can return the interpreters version to the users notebook.

The current SaC wrapper kernel needs to keep track of all the previous defined definitions, imports, variables and functions from other cells in the notebook, other than communicating with the SaC compiler. Keeping track of all this is called the kernel state. This state and saving of definitions is needed because the kernel will only receive expressions from a single cell at a time while every cell in the notebook should be able to access definitions defined in previous cells. This state will grow with every cells execution send to the kernel. Because the kernel still communicates with the SaC compiler, the current state needs to be transformed into a SaC program that is sent to the compiler. The kernel will gather the stored definitions together with the given expression and will create a SaC program as seen in listing 2 filling it in at all the `<...>` parts.

The expression from the notebook cell will be evaluated depending on the type of SaC syntax. For example if it is a definition, it will be put in the main function under `// statements`. If just a variable name is received by the kernel it will be put inside a SaC `print()` function inside of the main function in listing 2. This communication enables the kernel to catch compiler output via the ASCII standard out that `print()` prints to (also seen in figure 2.3). This is the current way of getting output in the users notebook. Because the kernel acts between the Jupyter server and the SaC

```
// use
<...>
// import
<...>
// typedef
<...>
// functions
<...>
int main () {
    // statements
    <...>
    return 0;
}
```

Listing 2: Empty SaC kernel program format

compiler, the kernel can utilise auxiliary commands not known by the SaC compiler, making a clear distinction between SaC syntax and these extra functionalities. These functions can, for example, give the user insights into the kernels state. These kernel commands are called *magics* and they commonly start with % to distinguish them. The *magics* present in the SaC wrapper kernel are:

- `%print`, prints the current kernel state as a filled SaC program in listing 2.
- `%setflags <flags>`, used to set SaC compiler flags.
- `%flags`, prints all current flags.
- `%help`, prints all *magics* with their description.

The existing wrapper kernel was originally constructed by A. Shinkarov and has been rewritten while this research was going on. This was done to make the kernel more abstract and uniform than the previous one, allowing for easier implementation of future additions to the kernel and better ways to communicate with the SaC compiler. It defines all the types of input *magics* functions, statements, expressions and all SaC types as action classes that have generalised behaviour and structure as seen in listing 3. The other classes present in the kernel are the Jupyter `subprocess` class, a sub-process that communicates with the SaC compiler and listens for ASCII output on standard out and can also read this in real time, and the `Kernel` class that holds the actual implementation of the kernel and inherits from the wrapper Jupyter kernel class. The `kernel` class holds information needed for the

Jupyter server to register it and connect to it as a kernel. Every Action in

```
class Action:
    def __init__(self, kernel):
        self.kernel = kernel

    def check_input(self, code):
        return {'found': False, 'code': code}

    def process_input(self, code):
        return {'failed': False, 'stdout':"", 'stderr':""}

    def revert_input (self, code):
        pass

    def check_magic (self, magic, code):
        code = code.strip ()
        if code.startswith (magic):
            return {'found': True, 'code': code[len (magic):]}
        else:
            return {'found': False, 'code': code}
```

Listing 3: SaC kernel action class

the kernel should have the `check_input`, `process_input` and `revert_input` methods defined. `check_magic` is only used for *magics* functions to check for the presence of the correct identifier of the given expression. `check_input` checks whether the given action is applicable to the given input and it returns a record {`'found', 'code'`}, indicating if the action has been found. If so, the input called `code` here will be used by `process_input` to perform the action. It returns a record { `'failed', 'stdout', 'stderr'` } to output an error to the Jupyter notebook if this is signalled. `revert_input` resets the internal state to the one before processing the input. As much state as possible is kept local to the actions.

2.5.2 Jupyter extensions

The second way to extend Jupyter other than a kernel is the Jupyter extensions. In particular this research looks at Jupyterlab extensions [1]. The extensions from Jupyter notebook and Jupyterlab differ, so that they cannot be used in both. The extensions for Jupyterlab have sparse documentation but the developers made a helpful Github repository [25] consisting of several examples. The idea of an extension is that the user can decide whether to use them or not by installing them separately.

Extensions in Jupyterlab are written in javascript or typescript because they are included into the browser Jupyter interface. Jupyterlab's developers have set up a template [1] that contains all the basic files needed for an extension. The most important files are: `src/index.ts`, that contains the actual code of the extension and can be extended with more typescript files, `package.json`, that contains information about the extension such as dependencies, and `tsconfig.json`, that contains information for the typescript compilation.

Jupyterlabs extension system was initially similar to the Jupyter notebook extensions called nbextensions, called `labextensions`. This system is now deprecated but was used to install, uninstall and edit extensions and could be called directly from a terminal. NodeJS is needed to build the extension package web assets such as TypeScript and CSS. The commands mostly used in development are the following:

- `jlpm`. The jlpm command is JupyterLab's pinned version of yarn (a package manager for Javascript) that is installed with JupyterLab. It allows you to update the Javascript code each time you modify your extension code and is used to install class dependencies such as the widget providing the Lumino class.

- `pip install -ve .` installs the dependencies that are specified in the setup.py file and in package.json. The TypeScript code gets converted to javascript using the compiler tsc and wrapped to be used and loaded into Jupyterlab.

The structure of the typescript file needs an initialisation function as seen in listing 4. This is an instance of the `JupyterFrontEndPlugin` that is a plugin that builds the extension and holds some required information, such as the ID and what packages it needs in order to function.

[1] Since the Jupyterlab project is in constant development, some statements in this thesis might no longer be valid for a newer version of Jupyterlab. The version used during this research was 3.6.1

```
const plugin: JupyterFrontEndPlugin<MyToken> = {
  id: 'my-extension:plugin',
  autoStart: true,
  requires: [...],
  optional: [...],
  provides: MyToken,
  activate: activateFunction
};
```

Listing 4: Initialisation function needed in the extension source file

The extension spawns from the `activate` function defined in the activate option in the initialisation function. It can be defined inside or outside of this class. `requires` and `optional` hold lists of Jupyter classes that are either needed to run the extension or are optional. They are mainly used when plugins interact with each other by providing a service to the system and requiring services provided by other plugins. For example if `ICommandPalette` is used for creating and adding commands to the existing ones the programmer needs to put the new command into the `required` field in order for the extension to stop working if this class is not available.

2.6 Matplotlib Python plotting library

The plotting library used in the implementation of this research is the Python Matplotlib. It is an open source plotting library that serves as a visualisation utility tool usable across several user interfaces and operating systems. It was first released in 2003 based and inspired by MATLAB [10] and has evolved into an extensive plotting library, supporting multiple types of plots in 2D and 3D as well as support for interactive plots. The library also uses a relatively small memory space [9]. This library is widely used by data scientists research using Python [12]. For example the library consists of box plots, scatter plots, images, contour plots, histograms and much more. All possible plotting options are found in the Matplotlib online documentation [24]. In Matplotlib the plots are divided into containers where each container can be altered to change certain properties inside of it. A simplified version can be seen in figure 2.4. The outer most container is the *figure* container, this container holds all the plots and data associated with its place such as size restrictions and plot arrangements. *Axes* holds information of a single plot such as the type of plot or its title. There are even more containers such as legends, lines, ticks and text boxes that are left out of the figure for simplicity.

Matplotlib makes use of the numerical mathematics library called Numpy.

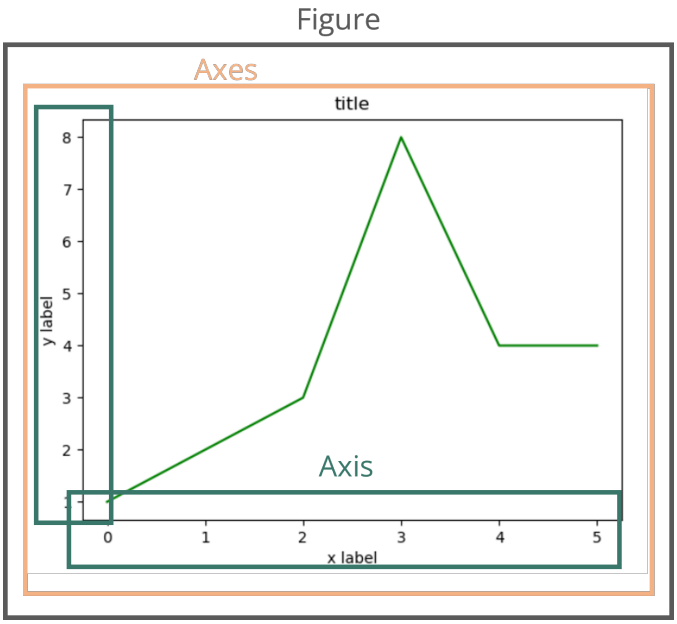

Figure 2.4: Simplified Matplotlib containers and their place in a plot

Numpy was created in 2005 and introduces a new array object into Python that has better supports for large, multi-dimensional arrays and matrices [7]. The primary reason Matplotlib uses Numpy arrays in some of the plots is because Numpy includes a collection of high-level mathematical functions to operate on the Numpy arrays. Matplotlib can be used without Numpy with the drawback that some plots will not be able to be used, such as the 3D surface plot. Having Numpy unlocks the full potential of Matplotlib.

3. Plotting SaC in Jupyter

The goal of this research is to enrich the interaction between the compiled language SaC and the Jupyter server by including plotting visualisations for SaC. As explained in section 2.5.1, the current SaC kernel was created only for the evaluation of SaC expressions having communications with the compiler via ASCII streams. The only insight into SaC arrays is in text form, as seen in listing 5. The SaC kernel was originally built for tutorials and showcasing of the SaC language and its functionality, which normally would require a manual compile-execute cycle to get any output. Visualisations are a great tool that helps in the interpretation of numerical data as substantiated in section 2.1. This also holds for SaC because the language was developed for operations on large data and with the implementation of visualisations in Jupyter this will give an easier way of interpreting large data arrays. SaC is also being used and taught at the Radboud University to Master Software Science students, where visualisations of arrays would aid in teaching its concepts to students. This research focuses on plotting, which is one of the best and most versatile way to visualise numerical data to recognise data trends. Moreover, plotting is heavily used in data exploration before any analysis or operation is done. This exploration speeds up the whole process by being able to spot mistakes or faults in the data early on or even change the whole approach to the problem [30].

```
Dimension:  2
Shape    : <  2,  3>
| 1  2  3 |
| 4  5  6 |
```

Listing 5: SaC `print()` function output from a small two dimensional array

3.1 Approach

This section is about the "intuition and design" of this research and finding an optimal way to be able to plot SaC arrays in Jupyter. To extend Jupyter, either extensions or kernels can be used (section 2.5). For the plotting functionality, the kernel is the best place to do so because of the direct connection it has with the SaC compiler and it receives the notebook cells expressions firsthand. Using extensions to plot is also possible but would still require either a connection with the kernel, or an external connection with the SaC compiler because of its position in the browser (3.1). The created plots in the extensions would also not directly be stored in the notebook but only in the browsers view or manually be fitted in. Saving images in a notebook is one of the strengths of the Jupyter notebook format, that will be harder when using extensions for this purpose.

The biggest hurdle with creating a plotting library in SaC is that the current kernel only gets output ASCII streams via standard out from the SaC compilers output (figure 3.1). Either stream encoding, that would need to be reconstructed to an image in the kernel, or a new way of handing over files is needed to get the created plots to the Jupyter server and consequently the notebook. The Jupyter communication protocol conveniently already has support for receiving images from the kernel, so no further additions need to be made here. SaC already has a module for creating PDF images and some file saving methods that could support the implementation. However, a plotting library needs to be created in the SaC library when using this route. This would take a tremendous amount of time to create such a library from scratch. Especially if the library needs to be very versatile, such as changing styles adding legends, and plot more than just 2D line-graphs. This is especially important because of the multidimensional arrays in SaC. This effort of repeating the creation of a full plotting library in SaC is beyond the scope of this research. Because the existing Jupyter wrapper kernel for SaC is written and interpreted by Python, it conveniently has access to everything Python has access to, including plotting libraries that already had years of development. Instead of reinventing the wheel, for this research the Matplotlib [9] library is chosen as described in section 2.6. Using this library gives all the features one could ever need and it has the benefit of being maintained and expanded upon constantly. It also already supports exporting plots to an image and can be directly used in the kernel with no need of setting up a communication protocol.

Now that there is a way to plot using the Matplotlib library, a method is needed for the user to be able to use it from inside the notebook. Because even though the kernel's state stores all declared variables, it can not use SaC array declarations that use list generator functions like `iota()`, that

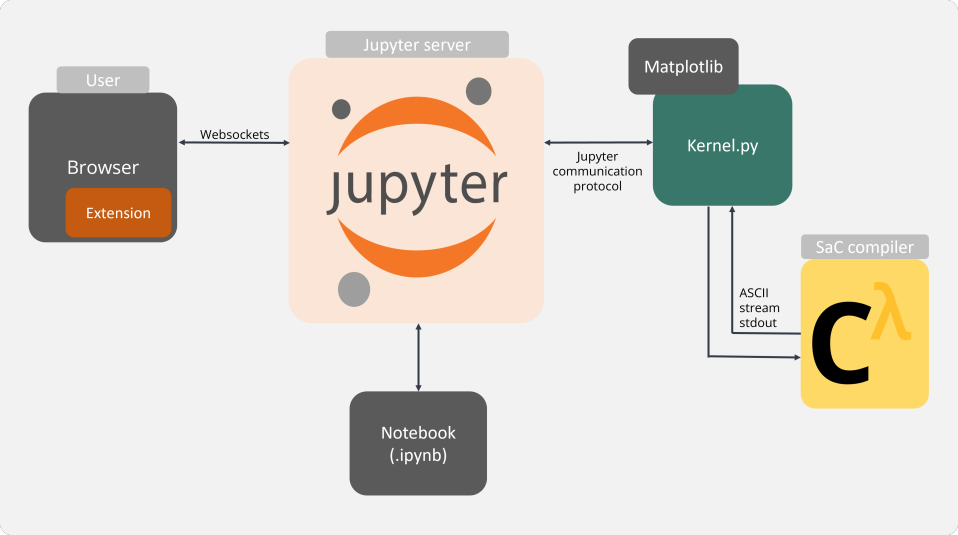

Figure 3.1: Expanded visual representation of the Jupyter architecture as seen in figure 2.3 including the Matplotlib library directly callable from the SaC kernel

are present inside of the SaC library of functions. This function creates an array with counted up numbers until the function input number. This function will be stored inside of the kernels state as for example for the variable called `a`, `a = iota(100)` creating an array with 100 integer values counting up from 0 till 99. Thus SaC arrays need to be altered in the kernel to be used by Python. Instead of creating many methods in the kernel that try to decode such expressions or converting it from the normal SaC print functions, the SaC compiler can be used to do this conversion. This is convenient because the SaC library has everything needed for this conversion. A conversion function called `pyPrint()` is added as a separate module to the SaC library. This function outputs, given any size or dimension array, a Python representation of it to an ASCII stream to standard out. This stream can be read by the kernel. It functions the same as the SaC `print()` function. This function behaves similarly to the normal print function in SaC by providing an ASCII output stream. The kernel can now directly interpret the output as a Python expression. The `pyPrint()` function can handle all SaC's fourteen built-in simple datatypes holding numbers and characters.

To convert input from the user in Jupyter to a plot and distinguish them from normal SaC syntax, a *magics* function is created in the kernel. These auxiliary functions are only known by the SaC kernel and thus can only be used from the notebook. To make this function work it needs two things, the SaC variables to be plotted and some syntax or instructions on the layout

and type of the plot. This function enables the user to pass Matplotlib Pyhton syntax directly into this *magics* function, to give the user access to all the Matplotlib library functionality and not be restricted to only a few plotting options. The following syntax is used for this magic function:

```
%plot (<SaC variables>) {<Python Matplotlib code>}
```

The `%` symbol is the *magics* syntax indicator that makes it easier for the kernel to distinguish it from SaC expressions. The variables represent the names of the SaC arrays separated by commas. The function looks like this because it is the most uniform syntax for programming languages functions and the same as the SaC syntax, resulting in a more uniform and intuitive look for users. The brackets make it easy to separate the Python Matplotlib code, the variables and the plot *magics* name for the kernel to parse. The only requirement for the code inside of the function is that the *figure* from a plot has to be defined as `fig` in order for it to be recognised by the kernel. In Matplotlib the *figure* is the outer most container that holds all plots (figure 2.4). The user can best start the plot by using `fig, ax = plt.subplots()`, that instantiates a new *figure* container and its sub containers *axes* called `ax` here (containers are explained more in 2.6). The user can also import other python or Matplotlib libraries by including it into the code body of the plot function.

The Matplotlib library depends partially on the Python Numpy array object as explained in section 2.6, because it uses some of its functionality to create its more complex visualisation options, such as the 3D surface plot. In this plot the list with all the height values of every coordinate needs to be a Numpy array. This conversion can be done using the `asarray` function before feeding it into the Matploltib plot function. Both Numpy and Matplotlib are already imported in the kernel and can be used as `np` and `plt` respectively. This is done to let the user focus on the plots and not have to import the essentials at every plot. The user can still re-import both libraries to change the names if desired.

Now that there is a way to plot and a function to get SaC values in Python syntax, the plot still needs to be returned to the user in Jupyter, which only has text based communication by default using the IOPub stream channels in the Jupyter architecture. This channel is used by the kernel to publish all side effects such as output and input to the Jupyter server but also other data (in ZeroMQ JSON format) that should be displayed. The primary goal is to be able to return an image of the plot. Matplotlib has this image creation build in already and can then be converted to a base64-encoded PNG representation and given to the Jupyter server using a different IOPub stream called `display_data`. Base64 is an encoding of binary to text stream limited to 64 different characters. The difference with the other input/out-

put streams is that it can attach extra data and metadata to the message, that in this case holds the width and height of the image [20][23].

3.2 SaC plotting execution cycle

The workflow of the kernel when receiving the plot *magics* function is as follows. The Jupyter server has sent the contents of a notebook cell to the kernel where the kernel recognises the *magics* indicator and it being the plot function. The kernel will first check whether Matplotlib is available on the current machine and return an error message to the user's notebook if not. The kernel will then try to extract the variables and Python code from this expression using a regular expression from the Python `re` module. This will return an error to the user if the desired plot *magics* function syntax is not met. A regular expression is used because it will also be able to return the variables in a list using the Python `Match` object returned by the regular expression instantly, by using its `group()` method. During the extraction of the SaC variables and the Python code, any error will be returned to the user in Jupyter, signalling a syntax error has occurred. Errors need to be caught and send to the Jupyter server because otherwise they are contained in the kernel and will not be visible to the user. After this the list of SaC variable names will be sent to the SaC compiler encapsulated in the `pyPrint()` function in order for the SaC compiler to immediately return the converted array back to the kernel using an ASCII stream. The returned Python arrays can be instantly assigned to a variable in the kernel using the `eval()` Python built-in function that parses and evaluates input as a Python expression. These newly created Python variables will then be fed into a Python local dictionary in order to be used in executing the Matplotlib Python code given by the user. To execute this code the `exec()` Python built-in function is used that dynamically executes Python code. Because `exec()` uses only the current scope, the aforementioned dictionary holding the SaC variables has to be given to `exec()` in order to access any assigned variables and plot objects after its execution. The figure from the executed block is extracted using the same local dictionary used for the SaC variables and converted to an image. This image will then be sent back to the Jupyter server signalling it as output from the current cell.

This implementation also makes sure any error that could occur is passed on as output to the users notebook instead of remaining in the kernel or being translated to an understandable error message.

3.3 Visualisation example

The plotting functionality constructed for SaC in this research is freely available from Github [27]. With this functionality users can for example create a 3D surface plot as in figure 3.2. Here SaC variables are used to plot the Ricker wavelet or commonly referred to as the Mexican hat wavelet used for image feature detection [22] for example used for face recognition [6]. In this example a SaC function is defined called `createZ` that creates a 2D array (`Z`) holding the height values of all x and y coordinates from this Ricker wavelet. This function uses the SaC `with`-loops, that are a key construct in the language. In this example it is similar to using a nested `for`-loop to fill a 2D array. The coordinates fed into the Ricker wavelet function (`sombrero()`, not visible in the screenshot) are offset by 20 to be able to translate it into the plot view. The Ricker wavelet is centred at coordinates (0,0). Inside of the plot *magics* function a 3D surface plot is created with lists `X` and `Y` being the coordinates for the plot. The module Colormap (`cm`) has been imported to get the colour gradient.

The full notebook as seen in figure 3.2 with additional plots and all code can be found in section A.

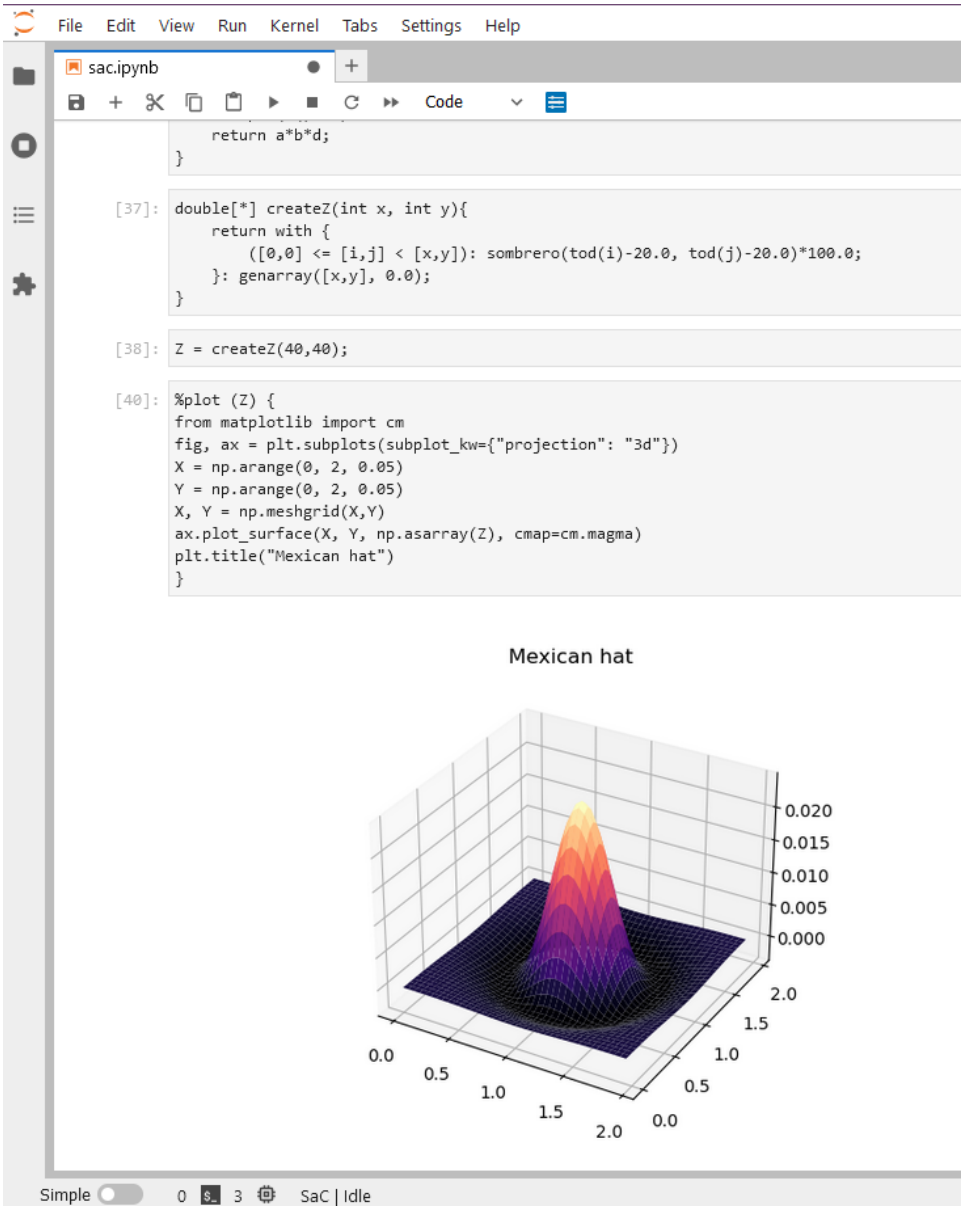

Figure 3.2: Example result of plotting a 3D Ricker wavelet from a SaC array into a 3D surface plot

4. Discussion

In this section the plotting implementation described in this research is evaluated and discussed. Also some technical remarks on the design and future ideas and improvements are made here.

4.1 Scope

The implementation of the plotting in SaC is a great tool enhancing the possibilities with SaC and the Matplotlib library used in this research gives access to a massive library of plotting visualisations. However, plotting after this research is only contained in the SaC kernel and the Jupyter project. SaC has outside of Jupyter still no way to plot its variables and programmers are forced to use Jupyter to get this feature when using SaC.

A potential drawback of using the current kernel structure still using the program compile-execute cycle, is the increased turnaround time. The notebook still needs to wait for program creation in addition to the creation of plots, that can result in additional waiting time. The bigger a notebook gets it inevitably has to wait longer for output because of the growing internal kernel state which it has to pass through the compilation loop for every execution. This can slow down programming development in SaC. Return time higher than 5 seconds (on a laptop with 4 cores) has not occurred during this research, but no real stress tests have been conducted. Since the use of big programs is not the current intended functionality of the Jupyter integration, long turnaround times are not expected to be encountered. The development process using the plotting implementation in Jupyter will still be faster than users struggling to interpret and evaluate the default ASCII SaC data output.

4.2 Future work

To expand upon the integration with Matplotlib, *magics* functions could be made to create some methods of plotting basic standard plots. These

functions enable plotting without the knowledge of Matplotlib and no direct need for any customisation of the plot. For example use `%plot line (<SaC arrays>)` to quickly get a line graph. This will make it easy to quickly plot simple arrays without needing much code and speeding up the rapid prototyping the Jupyter project is so good at. Or instead there is a simple command where the kernel decides what a good plot would be given a SaC array.

Another addition in the future, that would require a bit more work, is to eliminate the compile-execute delay from the kernel by creating an interpreter for SaC. This will remove the dependency with the compiler and enhance the interactive environment even more by having this quicker response-time for prototyping and testing.

Because of still unknown reasons, the SaC integration into Jupyter has problems with evaluating functions in the windows subsystem for Linux 1 (WSL1) [15], which is a Linux distribution that can be used directly from Windows. Normal Linux, Windows and IOS do not have this problem.

The implementation of the plotting in SaC is a great tool enhancing the possibilities with SaC, but there are many more extension possibilities in the Jupyter framework that can further enhance SaC. For this research a fully working Jupyterlab extension (see section 2.4.2) has not been achieved. The initial plan of this research was to create a kernel and an extension feature to show both ways of enhancing the SaC language in Jupyter. The extension should have been able to display the contents of the kernel state and updates whenever this state gets updated by for example new declarations of variables or changing flags. This would enhance the experience by eliminating the scrolling through a notebook needed in order to access a cell holding `%print` command output with the current state of the kernel. With this extension this output can be displayed next to the notebook in a separate window, saving time and showing the window structure in Jupyterlab. Another idea for an extension for SaC in Jupyterlab was to further enhance the interactive environment by the implementation of a proper conversion from a SaC notebook to a SaC file. This is only possible at the moment using the print *magics* function and manually copying it to a newly created SaC file. It would be easier to let Jupyter handle this automatically. This could be done using the Jupyter `nbconvert` module.

5. Conclusion

This research is a case study of implementing plotting visualisations for the compiled language SaC in the Jupyter interactive environment. The solution uses an existing Python plotting library in the SaC Jupyter kernel, giving access to a plethora of options without repeating the creation of this massive module in SaC. The communication between the SaC compiler and the kernel is based on a text stream. Using an existing plotting library directly in the kernel removes the implementation of an additional communication protocol between the SaC compiler and the kernel. The Matplotlib library is selected because it is usable on several user interfaces and operating systems while also having extensive plotting customisation and supporting multiple types of plots in 2D and 3D. The kernels job and integration of this plotting functionality in SaC is passing the right variables to SaC and using them to create the user's plot. The additions this implementation needed was a kernel *magics* function distinguishable from the normal SaC syntax. This *magics* function combines Python Matplotlib syntax and SaC variables. Furthermore multiple parsers to handle the input in combination with the upgrading of parts of the kernel. To get the right values of defined variables in the notebook, a module in the SaC library has been developed to convert SaC variables to Python syntax for direct use in the kernel. This implementation also makes sure any error that could occur is passed on as output to the users notebook instead of remaining in the kernel When the plot function is called from the Jupyter notebook, the plot is displayed to the user as a PNG image using the Jupyter communication protocol.

6. Acknowledgements

Many thanks go out to my supervisor Sven-Bodo Scholz for helping me and updating the SaC kernel that made it possible to improve my code and have a clearer focus. I also want to thank my family and friends for continuously supporting me through this project and giving feedback on my writing.

Bibliography

[1] Jupyterlab extension templates. `https://github.com/jupyterlab/extension-template`, last visited 4 January 2024.

[2] Zeromq, 2007. `https://zeromq.org/`, last visited 31 December 2023.

[3] Jupyter kernels, 2017. `https://github.com/jupyter/jupyter/wiki/Jupyter-kernels`, last visited 16 November 2023.

[4] Abdulmalek Al-Gahmi, Yong Zhang, and Hugo Valle. Jupyter in the classroom: An experience report. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education - Volume 1*, SIGCSE 2022, page 425–431, New York, NY, USA, 2022. Association for Computing Machinery.

[5] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. Julia: A fresh approach to numerical computing. *SIAM Review*, 59(1):65–98, 2017.

[6] Steven Gillan, Pan Agathoklis, and Mohamed Seddeik Yasein. A feature based technique for face recognition using mexican hat wavelets. In *2009 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, pages 792–797, 2009.

[7] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020.

[8] Konrad Hinsen. The promises of functional programming. *Computing*

in Science & Engineering, 11(4):86–90, 2009.

[9] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.

[10] The MathWorks Inc. Matlab version: 9.13.0 (r2022b), 2022. `https://www.mathworks.com`.

[11] Wolfram Research, Inc. Mathematica, Version 13.3. Champaign, IL, 2023.

[12] Jeremiah W. Johnson. Benefits and pitfalls of jupyter notebooks in the classroom. In *Proceedings of the 21st Annual Conference on Information Technology Education*, SIGITE '20, page 32–37, New York, NY, USA, 2020. Association for Computing Machinery.

[13] Konstantin Laufer and George K. Thiruvathukal. Scientific programming: The promises of typed, pure, and lazy functional programming: Part ii. *Computing in Science & Engineering*, 11(5):68–75, 2009.

[14] Jupyterlab Lumino. Jupyterlab extensions package for building interactive web applications. `https://github.com/jupyterlab/lumino`, last visited 10 January 2024.

[15] Microsoft. Windows subsystem for linux. last visited 3 January 2024.

[16] Thomas L. Naps, Guido Rößling, Vicki Almstrum, Wanda Dann, Rudolf Fleischer, Chris Hundhausen, Ari Korhonen, Lauri Malmi, Myles McNally, Susan Rodger, and J. Ángel Velázquez-Iturbide. Exploring the role of visualization and engagement in computer science education. *SIGCSE Bull.*, 35(2):131–152, jun 2002.

[17] Jupyter nbconvert tool. nbconvert: Convert notebooks to other formats. `https://nbconvert.readthedocs.io/en/latest/index.html`, last visited 10 January 2024.

[18] Fernando Pérez and Brian E. Granger. IPython: a system for interactive scientific computing. *Computing in Science and Engineering*, 9(3):21–29, May 2007.

[19] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2021. `https://www.R-project.org/`.

[20] Cyrille Rossant. *IPython Interactive Computing and Visualization*

Cookbook, Second Edition. 2018.

[21] Sven-Bodo Scholz. Single assignment C — efficient support for high-level array operations in a functional setting. *Journal of Functional Programming*, 13(6):1005–1059, 2003.

[22] Abhishek Singh, Aparna Rawat, and Nikhila Raghuthaman. *Mexican Hat Wavelet Transform and Its Applications*, pages 299–317. Springer International Publishing, Cham, 2022.

[23] Jupyter Development Team. Messaging in jupyter. `https://jupyter-protocol.readthedocs.io/en/latest/messaging.html`, last visited 3 January 2024.

[24] Matplotlib Team. *Matplotlib 3.8.2 documentation*, 2024. `https://matplotlib.org/stable/index.html`.

[25] Project Jupyter team. Jupyter extension examples, 2014. `https://github.com/jupyterlab/extension-examples`, last visited 17 January 2024.

[26] Project Jupyter team. Jupyter project, 2014. `https://jupyter.org/`, last visited 25 October 2023.

[27] SaC team. Jupyter kernel for sac, 2018. `https://github.com/SacBase/sac-jupyter`, last visited 14 January 2024.

[28] Guido Van Rossum and Fred L Drake Jr. *Python reference manual*. Centrum voor Wiskunde en Informatica Amsterdam, 1995.

[29] V. Vassilev, Ph. Canal, A. Naumann, L. Moneta, and P. Russo. Cling – the new interactive interpreter for ROOT 6. volume 396, page 052071. IOP Publishing, dec 2012.

[30] Ramanathan Venkatraman and Sitalakshmi Venkatraman. Big data infrastructure, data visualisation and challenges. In *Proceedings of the 3rd International Conference on Big Data and Internet of Things*, BDIOT '19, page 13–17, New York, NY, USA, 2019. Association for Computing Machinery.

[31] Jupyter Xeus. C++ implementation of the jupyter kernel protocol. `https://github.com/jupyter-xeus/xeus`, last visited 3 January 2024.

A. Appendix

In this section some additional plots are shown that were made in Jupyter with the plotting integration created in this research. The figures A.1, A.2, A.3 and A.4 are together the full example notebook from section 3. The additional plots in this notebook are a 2D height-map of the Ricker wavelet [22] and a line plot depicting the middle most slice of the wavelet.

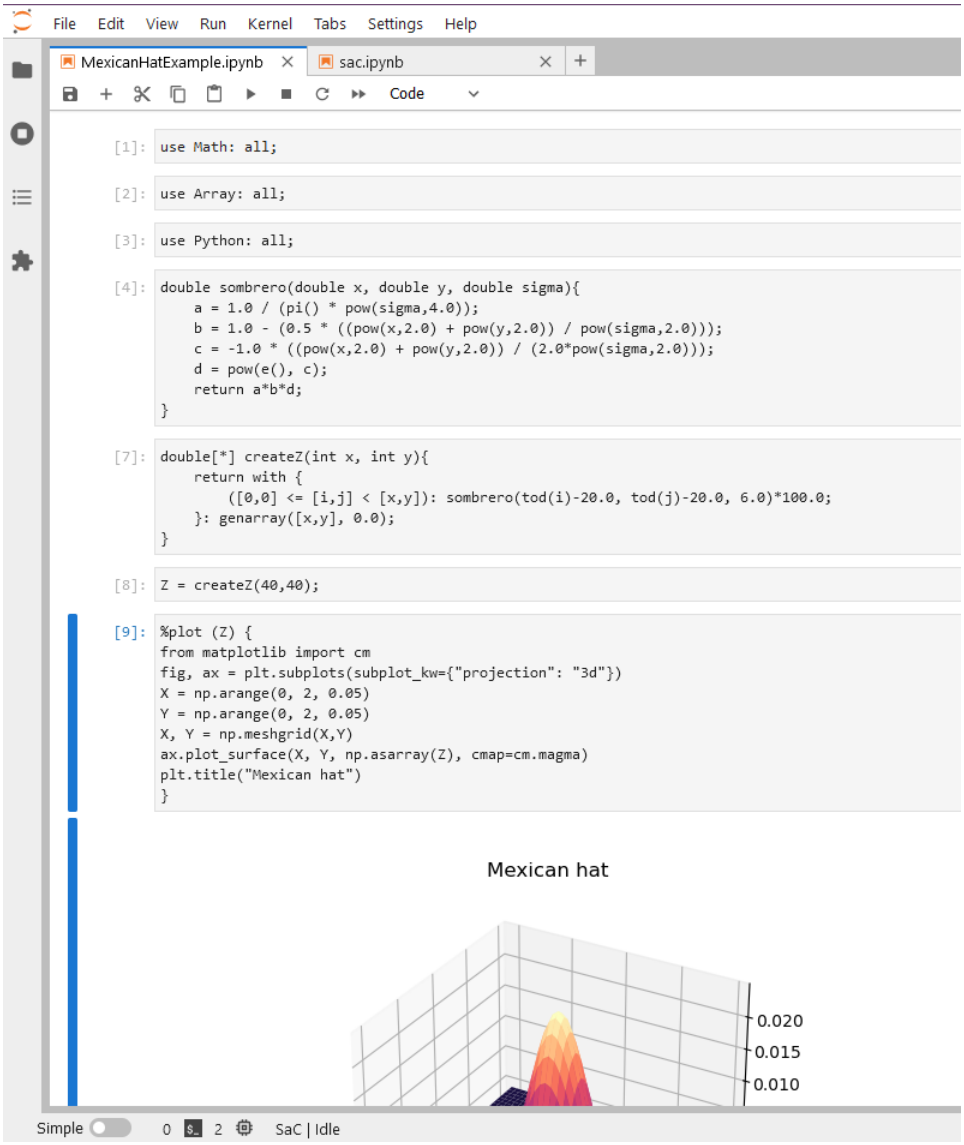

Figure A.1: Beginning of the Ricker wavelet example notebook. First three cells are importing SaC modules. The Python module is used to get SaC variables in Pyhton format from the compiler. The Math module is used to get the `pi()` and power (`pow()`) functions. The Array module is imported to be able to use arrays holding decimal values.

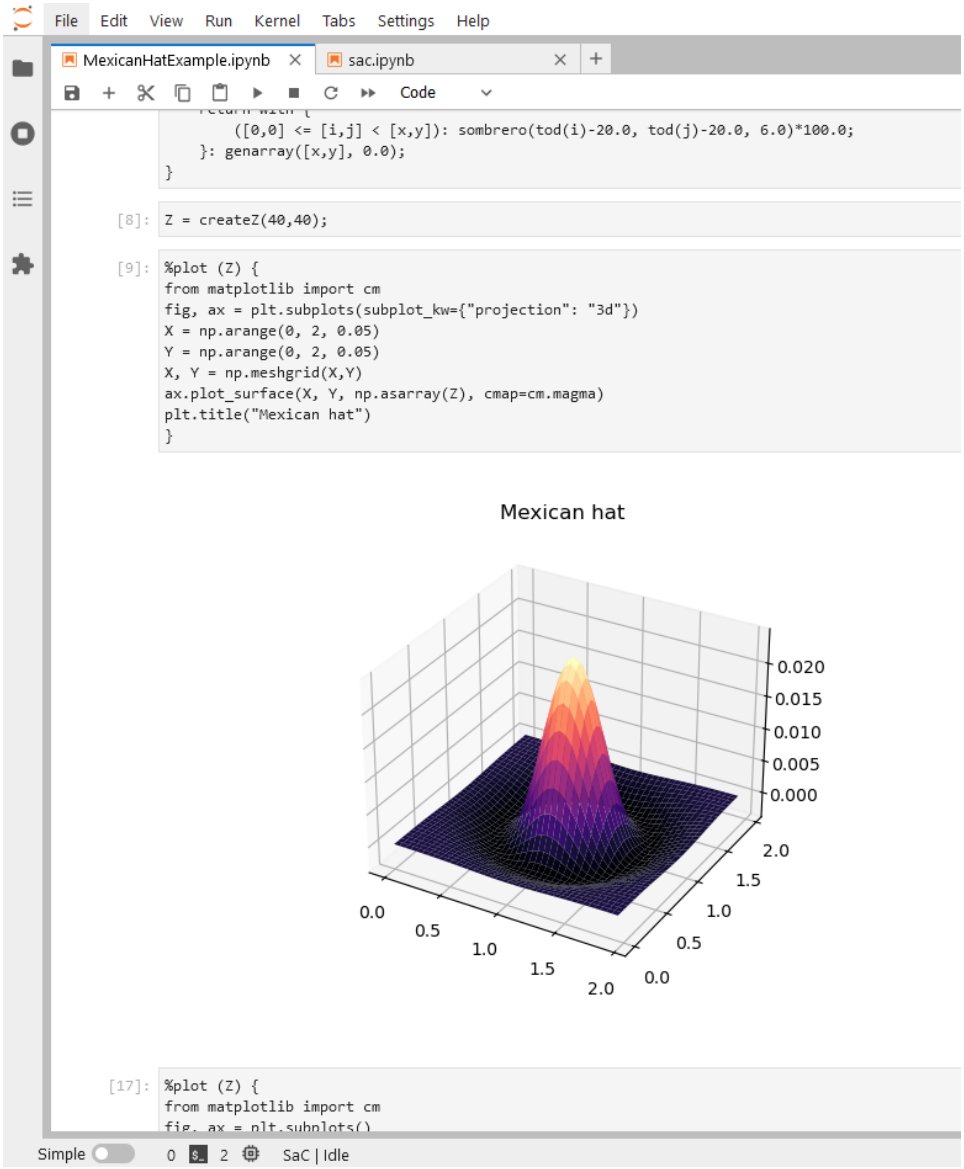

Figure A.2: Depicts the result of the Ricker wavelet in a 3D plot

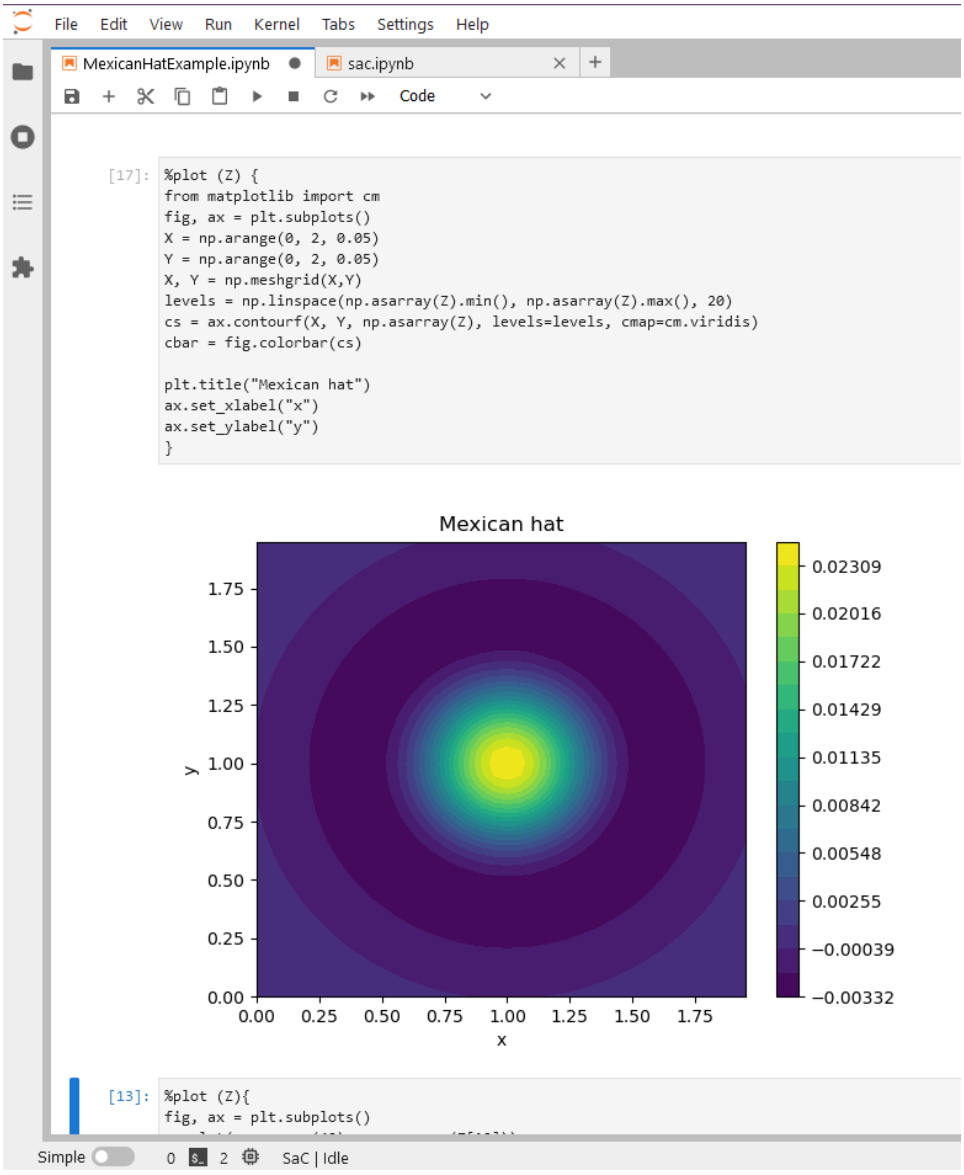

Figure A.3: Holds a cell containing the code to plot the Ricker wavelet as a height-map using a different color-mapping that is distributed in 20 layers from the highest to the lowest point in the wavelet.

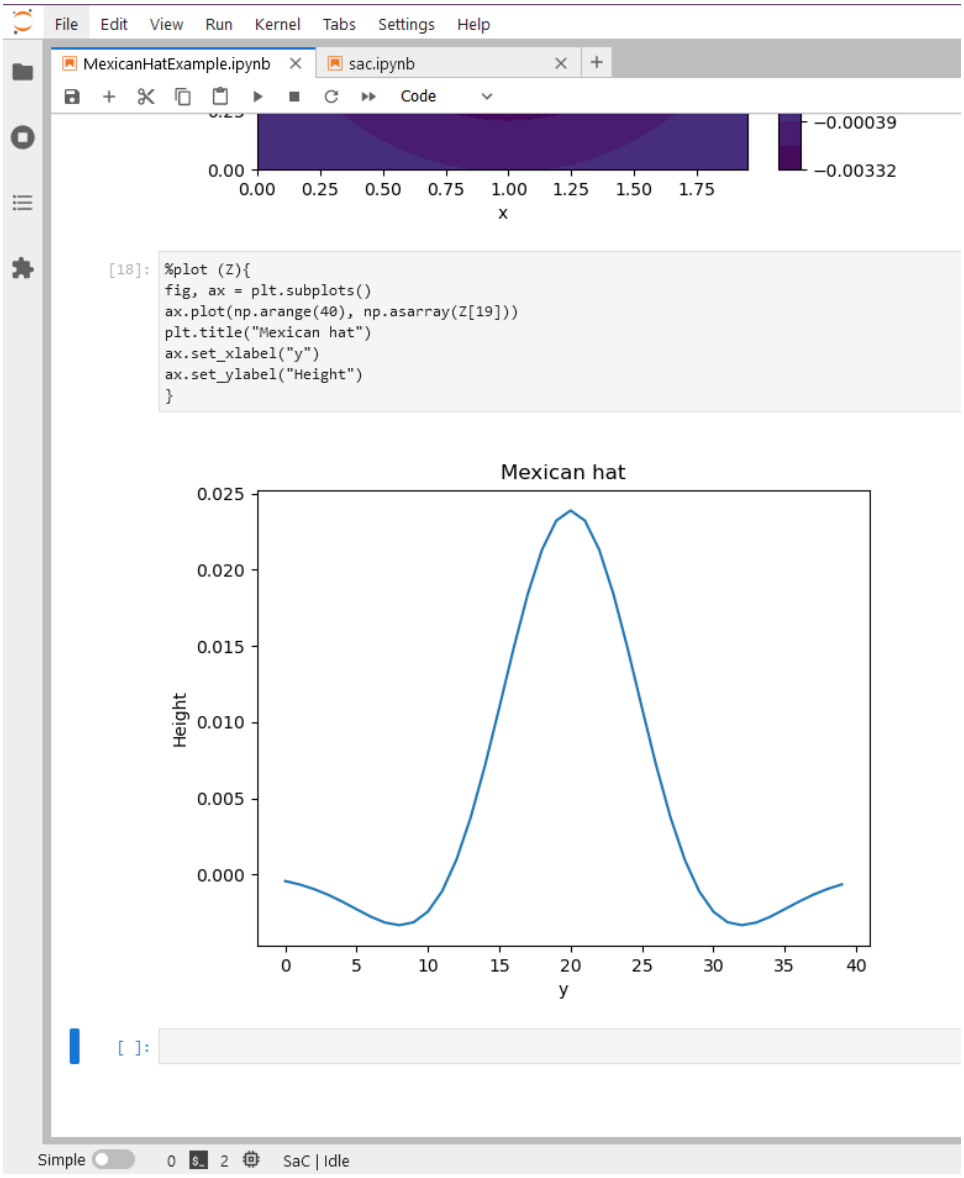

Figure A.4: Shows a cell that plots the height values of the Ricker wavelet on the middle x value.