

Using and Creating a New Python 3.13 Virtual Environment to Run Marimo

As mentioned in class Marimo (<https://marimo.io/>) is a new *reactive* Python notebook that has many nice features over the IPython/Jupyter notebook, Jupyter Lab legacy. For the curious [A marimo is a moss ball of densely packed algal filaments, free-floating filaments, or growth on rocks](#). If you would like to check out Marimo it is easy to create a new *virtual environment* using miniconda. Stanford University is where **marimo** got its start. I have watched a few videos and find many people pronounce **marimo** as "mar ee mo" or "mar i mo"

Create a new virtual environment

The new virtual environment will be created using the base environment of the previously installed **Miniconda** and its **base** environment.

Note: Virtual environments are where you do your work with Python. Creating multiple virtual environments is a safe way of trying new Python packages, such as **marimo**, without the risk of messing up a known good virtual environment. If you do mess up a new virtual environment you can always delete and start over. Leaving the minimalistic base environment of **Miniconda** intact gives you an environment you can always *activate* to safely manage one or more virtual environments.

The command line below will create a Python 3.13 virtual environment named **marimo-dsp-comm313**. This new environment, unlike the previous environment names **dsp-comm313** will not contain **jupyterlab**

```
(base) PS C:\Users\mwickert> conda create --name marimo-dsp-comm313 python=3.13
```

Activate the new environment:

```
(base) PS C:\Users\mwickert> conda activate marimo-dsp-comm313
```

Now install the Marimo package:

```
(marimo-dsp-comm313) PS C:\Users\mwickert> conda install -c conda-forge marimo
```

Now install the scipy stack:

```
(marimo-dsp-comm313) PS C:\Users\mwickert> conda install numpy matplotlib scipy
```

Now install scikit-dsp-comm:

```
(marimo-dsp-comm313) PS C:\Users\mwickert>conda install -c conda-forge scikit-dsp-comm
```

Note: For now I do not recommend installing `pyaudio`, as I have not tested it in `marimo`. You have your Jupyter-based virtual environment to fall back on for `pyaudio_helper` related work, which does use `ipwidgets`

Note: The above shows the use of Windows `Powershell` but your install of `miniconda` may equivalently end with you having to use the Windows `cmd` shell. On `macOS` you most likely will use the `zsh` shell and on Linux (including Windows Subsystem for Linux (WSL)) the `bash` shell.

Verify the `Marimo` Installation

```
(marimo-dsp-comm313) PS C:\Users\mwickert>marimo tutorial intro
```

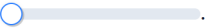
This will open a sample tutorial notebook of `marimo`:

The screenshot shows the Marimo application window. The title bar indicates the file path: `C:\Users\mwick\AppData\Local\Temp\tmp6erf9fbx\intro.py`. The notebook content is as follows:

```
1 import marimo as mo
2
3 mo.md("# Welcome to marimo! ")
```

Below the code cell, there is a slider UI element:

```
1 slider = mo.ui.slider(1, 22)
```

The text below the slider states: "marimo is a **reactive** Python notebook. This means that unlike traditional notebooks, marimo notebooks **run automatically** when you modify them or interact with UI elements, like this slider: .

At the bottom of the notebook, there is a code cell with the following content:

```
1 marimo is a reactive Python notebook.
2
3 This means that unlike traditional notebooks, marimo notebooks run
4 automatically when you modify them or
5 interact with UI elements, like this slider: {slider}.
6
7 {"##" + " " * slider.value}
```

The bottom status bar includes icons for file operations (copy, paste, save, etc.) and a "markdown" button.

Run an `edit` Session to Create and Open Existing Notebooks

To start working with your own `marimo` notebooks run:

```
(marimo-dsp-comm313) PS C:\Users\mwickert>marimo edit
```

You will see a "front page" for doing a wide variety of **marimo** notebook related tasks:



Create a new notebook

Resources



Documentation

Official marimo documentation and API reference



GitHub

View source code, report issues, or contribute



Community

Join the marimo Discord community



YouTube

Watch tutorials and demos



Changelog

See what's new in marimo

Running notebooks

first_comm_notebook.py

Documents\Courses\marimo\first_comm_notebook.py



Recent notebooks

first_comm_notebook.py

Documents\Courses\marimo\first_comm_notebook.py



Jan 29, 2026 at 9:21 PM

- Create a new notebook (very near the top)
- Tutorials pulldown (very top right)
- Five *Resources* tabs (below the new notebook cell)
- Next down, *Running Notebooks* (you can stop a notebook too)
- Next down, *Recent Notebooks* (those running can be stopped too)
- Finally, *Workspace Listing* of notebooks found on your system or in cloud drives

A Sample Start Cell for DSP/Comm Work

To get your notebook up and usable `import marimo as mo`, but also packages for DSP and Comm should be included. A good starting point is the following:

```
import marimo as mo

import numpy as np
import matplotlib.pyplot as plt
import scipy.signal as signal
import sk_dsp_comm.sigsys as ss
```

- Add more packages as needed
- Note that **marimo** wants you to follow the good Python practice of not importing entire packages into the global environment
- This means that all **numpy** functions, such as **arange**, **pi**, **sin**, etc. all need to be prefixed with the traditional alias name **np**
- To create plots using **matplotlib.pyplot** you prefix all plotting functions such as **plot**, **xlabel**, **grid**, etc with the alias **plt**
- Additionally after all the plotting related commands are given you need to show the plot using **plt.show()**, but when animation is present this will result in graphics jumping, so instead use **plt.gca()** (this is a **marimo** trick)

Documents\Courses\marimo\first_comm_notebook.py

```

1 import marimo as mo
2
3 import numpy as np
4 import matplotlib.pyplot as plt
5 import sk_dsp_comm.sigsys as ss

```

A First Comm Systems 1 Marimo Notebook

This is my way of getting started with Marimo. I want $x_1(t)$ to be plotted in both the time domain and the frequency domain via the power spectral density of a deterministic signal.

We are going to create a simple plot of a continuous-time sinusoid, but since it is in a simulation/modeling context, we are actually doing all of the calculations in the discrete time domain. This means that we have a sampling frequency f_s which is the reciprocal of the sample spacing T .

$$x_1(t) = \cos(2\pi f_1 t_1), \quad -4 \leq t \leq 4$$

```

1 slider_f1 = mo.ui.slider(start=1,step=10,stop=500,value=1,label=f'Frequency: ')

```

Frequency:

```

1 {slider_f1} {slider_f1.value}

```

Single Sinusoid at $f_1 = 1.00$ Hz

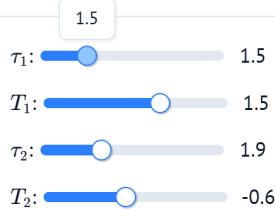
Note In **marimo** you the option to have code above or below the cell output. In the screenshots here I have the output above the code, which is the way Julia reactive **pluto** notebooks work by default. Of course **jupyter** notebooks have outputs below the code.

A Simple Slider Layout using **mo.vstack()**

Here we use a layout container to vertically stack four sliders using **mo.md(f'')** f-strings that include the interactive value of each of the sliders: part1 creating and stacking the sliders, part 2 creating an interactive

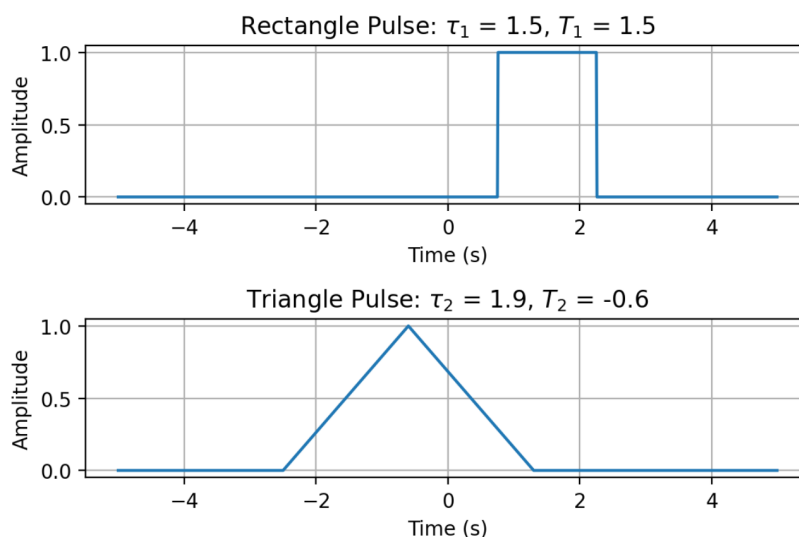
plot:

```
1 # Define the some sliders
2 slider_tau1 = mo.ui.slider(start=0.5,step=0.1,stop=5,value=0.5,label=r'\tau_1$:')
3 slider_T1 = mo.ui.slider(start=-5,step=0.1,stop=5,value=0,label=f'$T_1$:')
4 slider_tau2 = mo.ui.slider(start=0.5,step=0.1,stop=5,value=0.5,label=r'\tau_2$:')
5 slider_T2 = mo.ui.slider(start=-5,step=0.1,stop=5,value=0,label=r'$T_2$:')
6 # mo.vstack([mo.hstack([slider_tau1,slider_tau1.value]),slider_T1,slider_tau2,slider_T2])
```



```
1 # Display the sliders
2 mo.vstack([mo.md(f'{slider_tau1} {slider_tau1.value}'),
3 mo.md(f'{slider_T1} {slider_T1.value}'),
4 mo.md(f'{slider_tau2} {slider_tau2.value}'),
5 mo.md(f'{slider_T2} {slider_T2.value}')])
```

- Code cells above and below in this notebook can be closed

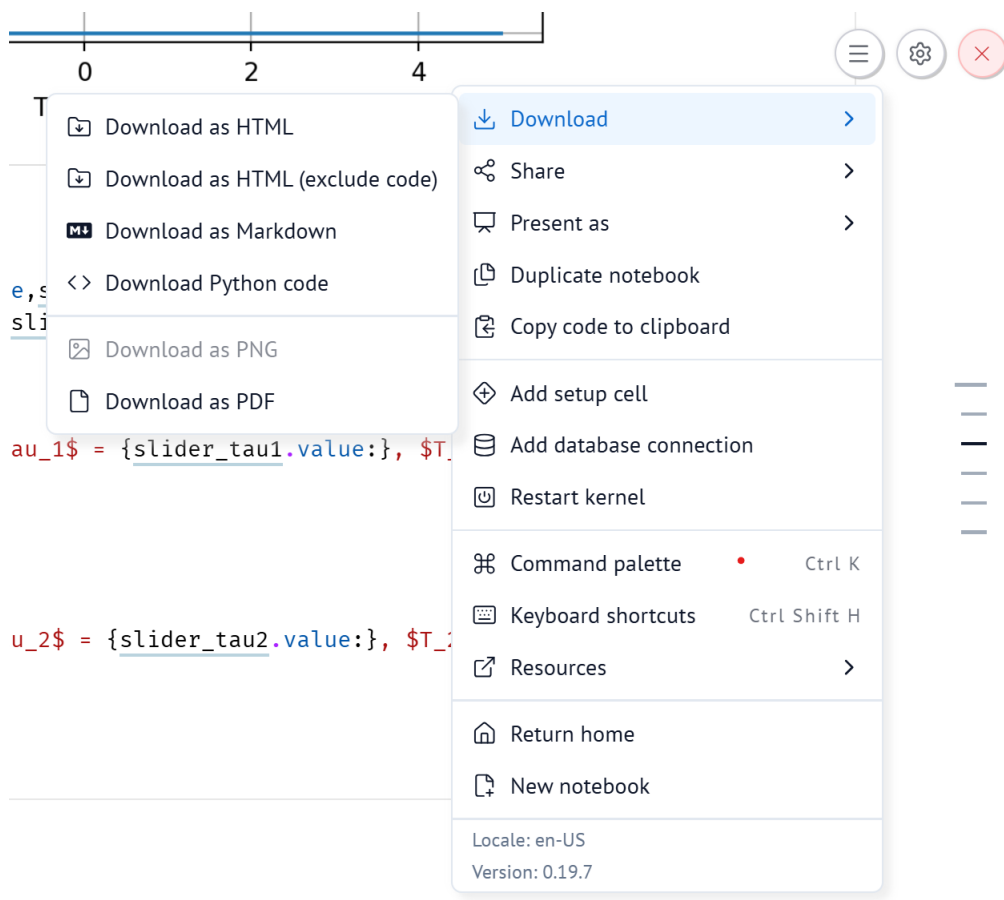


```
1 # Make plots
2 plt.figure(figsize=(6,4.0))
3 t = np.arange(-5,5,.01)
4 x_rect = ss.rect(t-slider_T1.value,slider_tau1.value)
5 x_tri = ss.tri(t-slider_T2.value,slider_tau2.value)
6 plt.subplot(211)
7 plt.plot(t,x_rect)
8 plt.grid()
9 plt.title(f'Rectangle Pulse: \tau_1 = {slider_tau1.value}, T_1 = {slider_T1.value}')
10 plt.xlabel(r'Time (s)')
11 plt.ylabel(r'Amplitude');
12 plt.subplot(212)
13 plt.plot(t,x_tri)
14 plt.grid()
15 plt.title(f'Triangle Pulse: \tau_2 = {slider_tau2.value}, T_2 = {slider_T2.value}')
16 plt.xlabel(r'Time (s)')
17 plt.ylabel(r'Amplitude');
18 plt.tight_layout()
19 plt.gca()
```

Note The use of `plt.gca()` to avoid jumping when the sliders are adjusted works as advertised. To me the interactivity with `marimo` is not as snappy as using `Pluto` reactive notebooks with Julia.

Exporting to PDF

The instructions here apply to version 0.19.7. Click the *hamburger* button in the upper right:



- The download as PDF works the best on a system with a full **LaTeX** setup, but code cells are usually closed
- Printing a notebook to HTML first is easier but has caveats
- Good: Exporting to HTML the widgets are preserved in the HTML document and render when printed to pdf
- Good & Bad: In the current version code cells always seem to *close* when printing to pdf in say Chrome, Safari, and Edge browsers
- To resolve this you can always place a markdown cell immediately below with Python colorization set, e.g.

```
# Plot the time and frequency domains of a rect pulse
tau_ft1 = 1.5
t_ft1 = np.arange(-4,4,.001)
x_ft1 = ss.rect(t_ft1,tau=tau_ft1)
f_ft1 = np.arange(-5,5,.001)
X_ft1 =tau_ft1*np.sinc(tau_ft1*f_ft1)
# Time domain
plt.figure(figsize=(6,3))
plt.subplot(121)
plt.plot(t_ft1,x_ft1)
plt.xlabel(r'Time (s)')
plt.ylabel(r'Amplitude')
plt.title(f'Time Domain  $\tau = \tau_{ft1}$  {tau_ft1:1.1f}')
```

```
# Frequency domain
plt.subplot(122)
plt.plot(f_ft1,np.abs(X_ft1))
plt.xlabel(r'Time (s)')
plt.ylabel(r'Magnitude')
plt.title(f'Frequency Domain  $\tau = \tau_{ft1} \cdot 1.1f$ ')
plt.tight_layout()
plt.gca()
# plt.savefig('ft_example.svg')
```

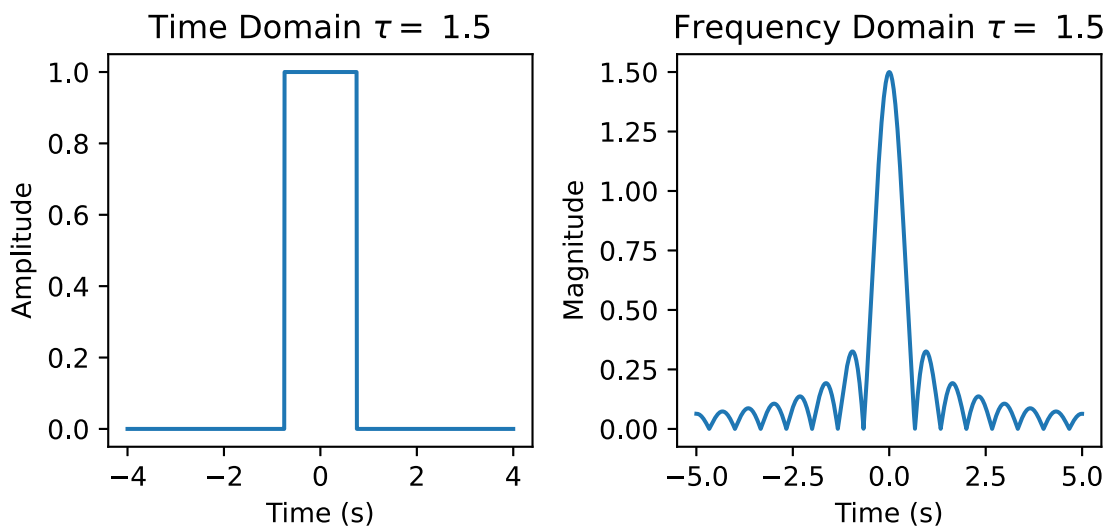
- Now in the printing of the HTML output to PDF the code cells will close and the Markdown cells containing code you want visible will indeed **be visible**, e.g., homework turn in, etc.

Typst and *Download as Markdown*

- With Download as Markdown you get a nice Markdown file that be imported into [Typst](#)
- The figures/graphics have to be dealt with separately, but you can add a single line at the bottom of each computation cell producing `matplotlib` graphics of the form:

```
plt.savefig('ft_example.svg')
```

- The above FT example will return an SVG graphics which is easily managed in Typst
- Here I shown it rendered to PDF from this Markdown file (converted to PDF) I am writing in *vs code* using the `Markdown PDF` extension:



A figure caption can be added in Typst or in Markdown