Xeus-Cpp-Lite

Interpreting C++ in the Browser

QuantStack
Scientific Computing

COMPILER
C|R
RESEARCH

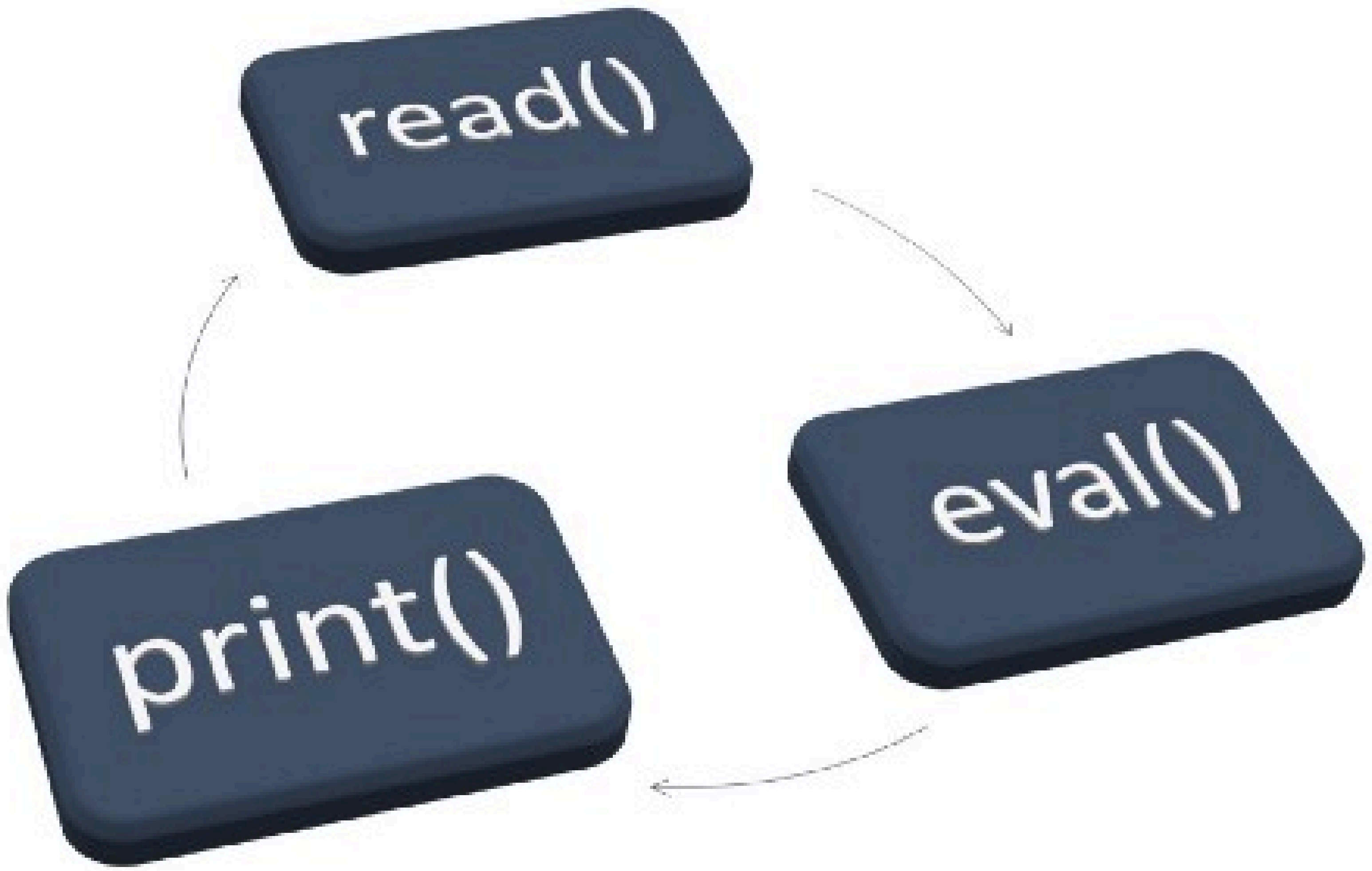
Agenda

- Motivation: Why C++ Needs a REPL
- Native Case: Xeus-Cpp via Clang-Repl
- Enter the Browser: JupyterLite & WebAssemby
- Initial Proof of Concept: Clang-Repl in the browser
- Bringing it All Together: Xeus-Cpp-Lite
- Demos & Use Cases
- Deploying Your Own Setup
- Future work (Near and Far)
- Acknowledgments

Xeus-Cpp-Lite = Xeus-Cpp + JupyterLite

Interpreting C++ in the Browser

REPL



REPL

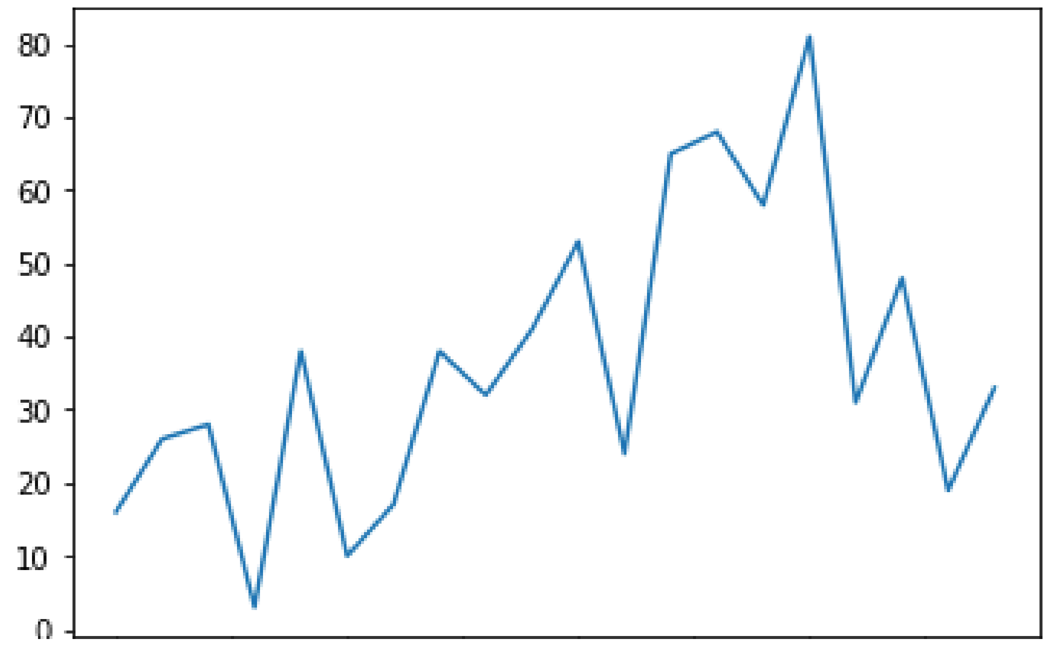
env1 (Python 3.9 (64-bit)) Interactive 1

Environment: env1 (Python 3.9 (64-bit) Module: __main__

```
>>> import math
>>> sines = []
>>> for i in [-0.5, 0, 0.5, 1.0]:
...     sines.append(math.sin(math.pi*i))
...
>>> i
1.0
>>> sines
[-1.0, 0.0, 1.0, 1.2246467991473532e-16]
>>>
```

REPL

```
Jupyter QtConsole 4.3.1
Python 3.6.3 (default, Oct  3 2017, 21:45:48)
Type 'copyright', 'credits' or 'license' for more information
IPython 6.4.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: %matplotlib inline

In [2]: import matplotlib.pyplot as plt

In [3]: import random

In [4]: data = [random.randint(0, 100) for i in range(20)]

In [5]: plt.plot(data)
Out[5]: [<matplotlib.lines.Line2D at 0x7fbf8f04b160>]
```



Motivation: Why C++ Needs a REPL

Scientific Computing is Exploratory

- Scientists and engineers don’t just write software — they explore
- They iterate rapidly: write code, run it, visualize, inspect, repeat

Interpreted Languages Dominate This Space

- Tools like Python and R excel due to their REPL-driven workflows
- C++ is powerful, but it's compile-run-debug loop is friction-heavy

C++ in a REPL = The Best of Both Worlds

- Projects like Cling showed that interactive C++ is possible
- Bringing C++ into a REPL format opens doors for teaching, rapid prototyping, and scientific computing

C++ Kernel through **cling** and **Xeus-cling**

- **cling :** https://github.com/root-project/cling
- **Xeus-cling :** https://github.com/jupyter-xeus/xeus-cling
- **Blog :** https://blog.jupyter.org/interactive-workflows-for-c-with-jupyter-fe9b54227d92
- Around half a million views on the above blog, demonstrating the importance of using C++ as a REPL

C++ Kernel through **cling** and **Xeus-cling**

jupyter xcpp Last Checkpoint: yesterday

File Edit View Run Kernel Settings Help Not Trusted

Markdown JupyterLab C++17 Binder GitHub Download

```
        std::stringstream m_buffer;
    };

    nl::json mime_bundle_repr(const image& i)
    {
        auto bundle = nl::json::object();
        bundle["image/png"] = xtl::base64encode(i.m_buffer.str());
        return bundle;
    }
}
```

[21]:

```
im::image marie("images/marie.png");
marie
```

[21]:

Native Case: Xeus-Cpp via Clang-Repl

Clang-Repl

- Interactive interpreter built into Clang/LLVM.
- Provides the C++ “Read-Eval-Print-Loop” infrastructure.
- Developed by Vassil Vassilev.

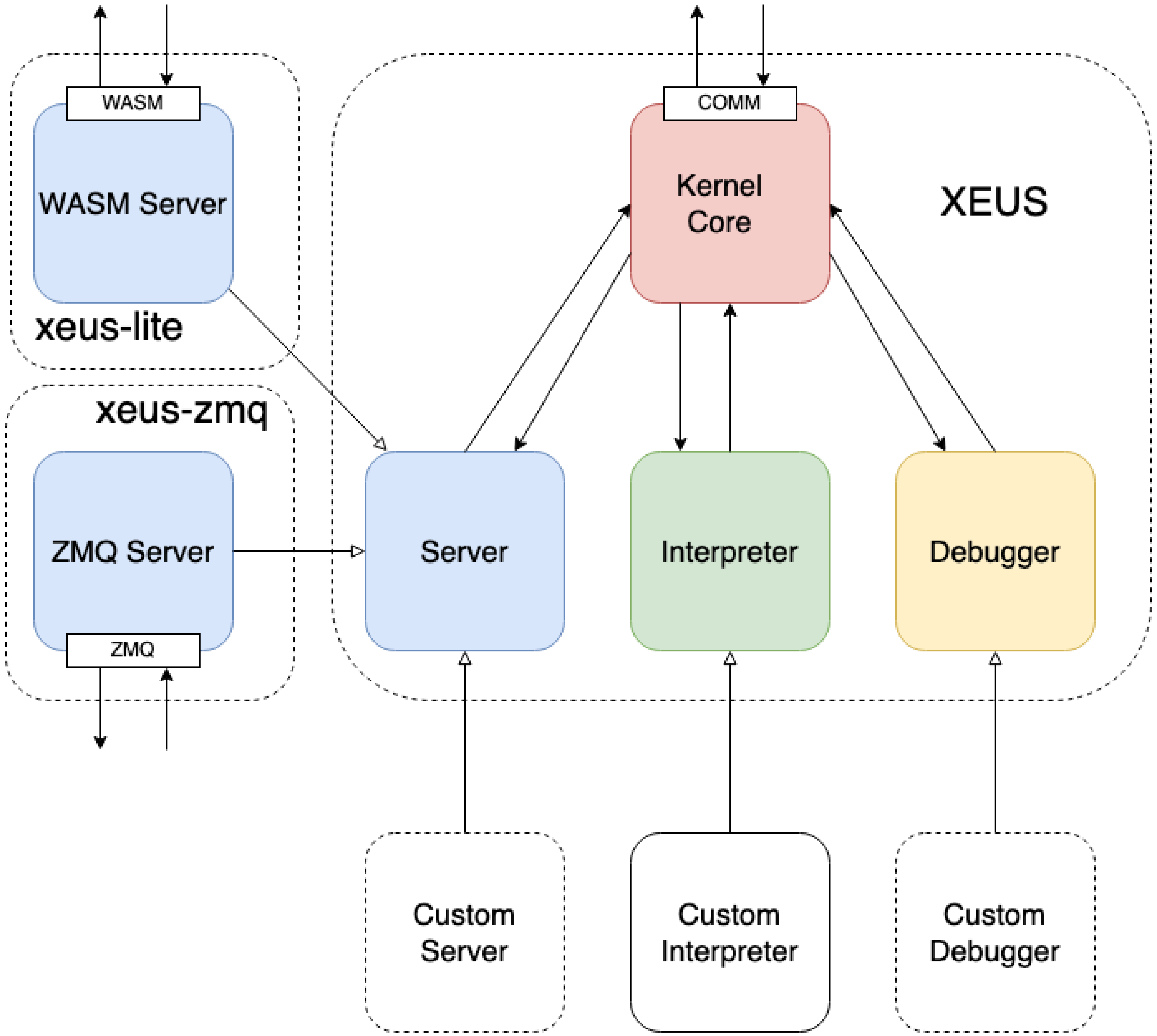
CppInterOp

- A thin C++ layer over Clang-Repl.
- Simplifies interaction and provides a stable API.
- Used by kernels and apps embedding Clang-Repl.

Xeus-Cpp

- A native Jupyter kernel that connects CppInterOp to the Jupyter protocol via Xeus.
- Enables executing C++ code cell-by-cell in Jupyter Notebooks.

Xeus Architecture



Native Case: Xeus-Cpp via Clang-Repl

Bridging the Gap

- Xeus-Cpp fulfills the need for a C++ REPL inside Jupyter.
- Based on Clang-Repl, it offers modern, upstream-supported C++ interpretability.
- CppInterOp provides clean integration into the kernel.

Why This Matters

- Avoids Cling's maintenance burden (patches to Clang).
- Built on mainline LLVM — easy to upgrade and package.
- Forms the foundation for future extensions (debugger, browser support, etc.)

Enter the Browser: JupyterLite & WebAssembly

What is JupyterLite?

Jupyter Without a Backend

- Traditional Jupyter requires a server per user → heavy infra costs.
- JupyterLite runs entirely in the browser using WebAssembly.
- Built to support statically hosted notebooks (e.g., GitHub Pages, Netlify).

Powered by WebAssembly

- Kernels are compiled to Wasm and run client-side.
- No backend. No Docker. No Kubernetes.
- Scales infinitely — the browser becomes the compute engine.

One Kernel Per Tab

- Each user session is isolated, with a fresh in-browser kernel instance.
- Think of it like a self-contained JupyterLab in your browser.

Enter the Browser: JupyterLite & WebAssembly

JupyterLite in Action

Real-World Deployments

- **NumPy.org** includes a live JupyterLite console.
- **SymPy Live** offers symbolic math in the browser via JupyterLite.
- **Capytale (France)** uses JupyterLite for teaching Python to 500K+ students.

Versatile Kernel Support

- Supports Python (via Pyodide & Xeus), R, Lua, Javascript, and more.
- Fully integrated with core Jupyter features (widgets, plotting, rich outputs).

No Compromise on Scientific Use

- Libraries like numpy, pandas, matplotlib, sympy, and scipy all work.
- A full REPL and IDE experience — with zero cloud infrastructure.

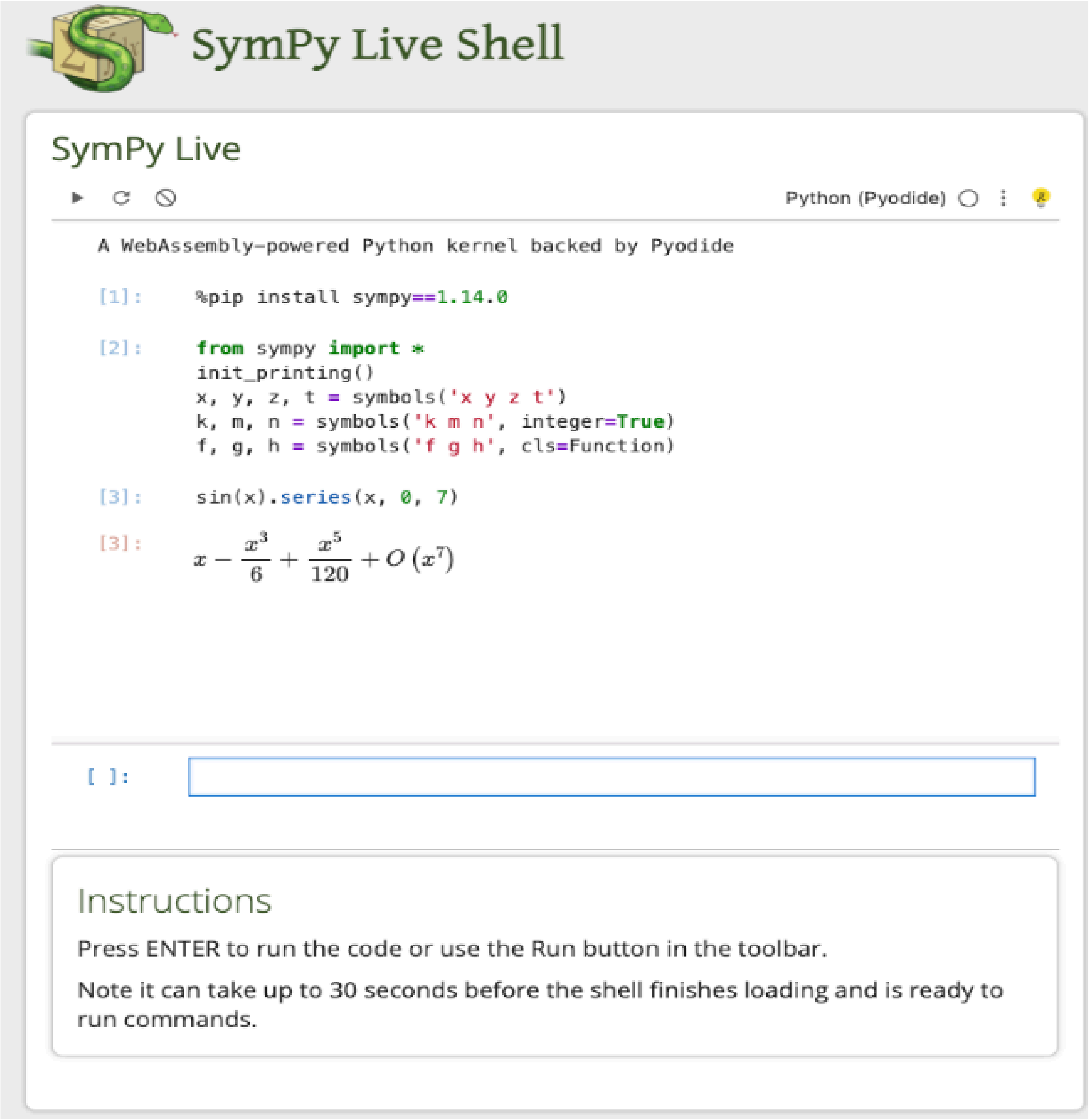
NumPy.org

Python (Pyodide)

A WebAssembly-powered Python kernel backed by Pyodide

```
[1]: import numpy as np

[2]: x = np.arange(15, dtype=np.int64).reshape(3, 5)
     x[1:, ::2] = -99
     x

[2]: array([[  0,   1,   2,   3,   4],
            [-99,   6, -99,   8, -99],
            [-99,  11, -99,  13, -99]], dtype=int64)

[3]: x.max(axis=1)

[3]: array([ 4,  8, 13], dtype=int64)

[4]: rng = np.random.default_rng()
     samples = rng.normal(size=2500)
     samples

[4]: array([ 0.09698012, -0.42595965, -1.933121  , ...,  0.25605178,
            -0.49469399,  1.01040697])

[ ]:
```

SymPy Live


SymPy Live Shell
SymPy Live
Python (Pyodide)
A WebAssembly-powered Python kernel backed by Pyodide
[1]: %pip install sympy==1.14.0
[2]: from sympy import *
init_printing()
x, y, z, t = symbols('x y z t')
k, m, n = symbols('k m n', integer=True)
f, g, h = symbols('f g h', cls=Function)
[3]: sin(x).series(x, 0, 7)
[3]: $x - \frac{x^3}{6} + \frac{x^5}{120} + O\left(x^7\right)$
[ ]:
Instructions
Press ENTER to run the code or use the Run button in the toolbar.
Note it can take up to 30 seconds before the shell finishes loading and is ready to run commands.

Enter the Browser: JupyterLite & WebAssembly

The Need for Emscripten-Forge

Why Not Just Use Conda-Forge?

- Conda-Forge builds native binaries (Linux, macOS, Windows).
- These can't run in browsers — no WebAssembly support.
- It also assumes a traditional filesystem and POSIX APIs.

Enter Emscripten-Forge

- A conda package distribution built for WebAssembly.
- Uses the **emscripten-wasm32** target + **rattler-build** + **mamba**.
- Packages are compiled to Wasm using the Emscripten toolchain.

What It Provides

- Core packages for Python, R, C++, CLI apps, and more.
- Makes kernels like xeus-python, xeus-r, and xeus-cpp-lite possible in JupyterLite.
- Created by **Thorsten Beier**, now maintained by the broader JupyterLite team.

Initial Proof of Concept: Clang-Repl in the browser

Why Doesn't JIT Work in the Browser

- On native platforms, Clang-Repl uses LLVM's ORC JIT, which compiles code at runtime and jumps to that memory to execute — standard Just-In-Time compilation.

- But in the browser, WebAssembly follows a strict sandbox model:
    - You can't write or modify executable memory at runtime.
    - Memory is separated into code and data (Harvard architecture).

- This means: even if clang-repl is compiled to WASM, it can't act as a REPL — it can't emit and run new code dynamically using the JIT model.

- Therefore, we needed a completely different approach for incremental, dynamic execution in the browser.

- This gap is what Anubhab Ghosh explored during his GSoC project — leading to the idea of a new backend for WASM. GSoC Project report link : Anubhab Ghosh Report

Initial Proof of Concept: Clang-Repl in the browser

A New Web Assembly Backend for Clang-repl

- LLVM 17 introduced a **WASM-specific IncrementalExecutor** that avoids JIT and fits the WebAssembly model.
- This new IncrementalExecutor class handled the wasm-specific execution model as follows:
  - Each REPL input is parsed into a **Partial Translation Unit (PTU)**.
  - PTU is lowered to **LLVM IR**, which is compiled to a **WASM object file**.
  - That object is then **linked using wasm-ld** into a **standalone WASM binary** (incr_module_x.wasm).
  - This side module is **dynamically loaded using emscripten's dlopen**, extending the state of the main module.

Initial Proof of Concept: Clang-Repl in the browser

A New Web Assembly Backend for Clang-repl

- These modules:
  - **Share the same memory** as the main wasm module.
  - **Resolve symbols** from earlier cells (cross-cell linking).
  - Mimic **dynamic linking** (even though WASM doesn't support shared libraries traditionally).
- This model effectively turned **clang-repl into a live REPL for WebAssembly**, enabling dynamic incremental C++ in the browser!
- Proof of Concept : https://github.com/anutosh491/clang-repl-wasm
- Lacked dedicated testing upstream :(

Problems using **clang-repl** in the **browser** through **LLVM 19**

[2]:
```
extern "C" int abs(int x);
extern "C" int printf(const char*,...);
auto result = abs(-42);
printf("r=%d\n", result);
```

```
r=42
```

[3]:
```
printf("r=%d\n", result);
```

```
r=42
r=42
```

[4]:
```
printf("r2=%d\n", 100);
```

```
r=42
r=-1
r2=100
```

Problems using **clang-repl** in the **browser** through **LLVM 19**

[1]:
```
#include <iostream>
```

[2]:
```
using namespace std;
```

[4]:
```
cout << "hello world" << endl;
```

```
hello world
```

[6]:
```
cout << "hello world" << endl;
```

```
Could not load dynamic lib: incr_module_6.wasm
CompileError: WebAssembly.Module(): Compiling function #17:"__wasm_call_ctors" failed: not enough arguments on the stack for call (need 1, got
0) @+4340
Failed to execute via ::process:Failed to load incremental module
Error: Compilation error! Could not load dynamic lib: incr_module_6.wasm
CompileError: WebAssembly.Module(): Compiling function #17:"__wasm_call_ctors" failed: not enough arguments on the stack for call (need 1, got
0) @+4340
Failed to execute via ::process:Failed to load incremental module
```

[7]:
```
int x = 10;
```

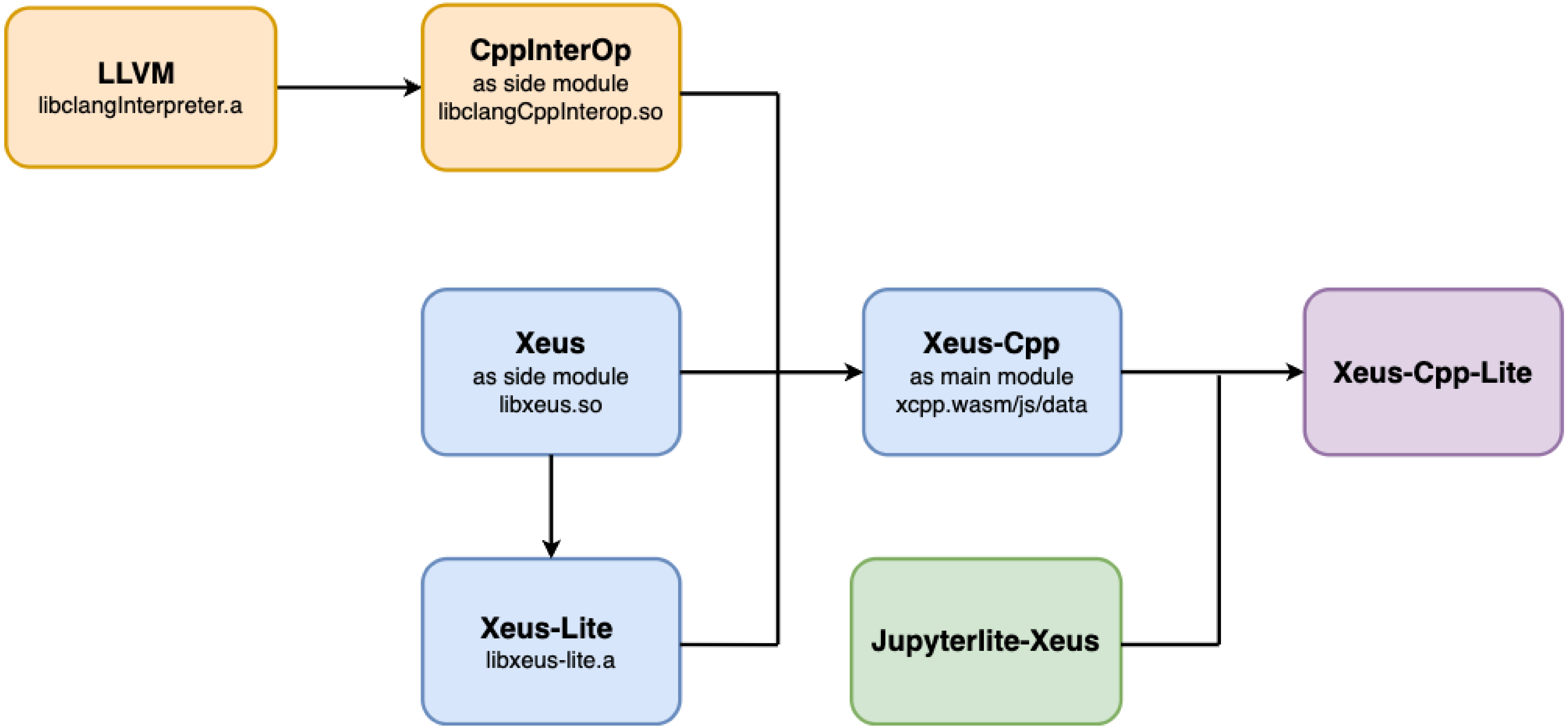
```
Could not load dynamic lib: incr_module_7.wasm
CompileError: WebAssembly.Module(): Compiling function #6:"__wasm_call_ctors" failed: not enough arguments on the stack for call (need 1, got 0)
@+406
Failed to execute via ::process:Failed to load incremental module
Error: Compilation error! Could not load dynamic lib: incr_module_7.wasm
CompileError: WebAssembly.Module(): Compiling function #6:"__wasm_call_ctors" failed: not enough arguments on the stack for call (need 1, got 0)
@+406
Failed to execute via ::process:Failed to load incremental module
```

List of Pull Requests :

Implementing the backbone for clang-repl in the browser

- PR #86402 – Initial WebAssembly support for clang-repl
- PR #113446 – Fix undefined lld::wasm::link symbol
- PR #116735 – Improve flags responsible for generating shared wasm binaries
- PR #117978 – Fix generation of wasm binaries
- PR #118107 – Remove redundant shared flag while running clang-repl in browser

Bringing it all Together

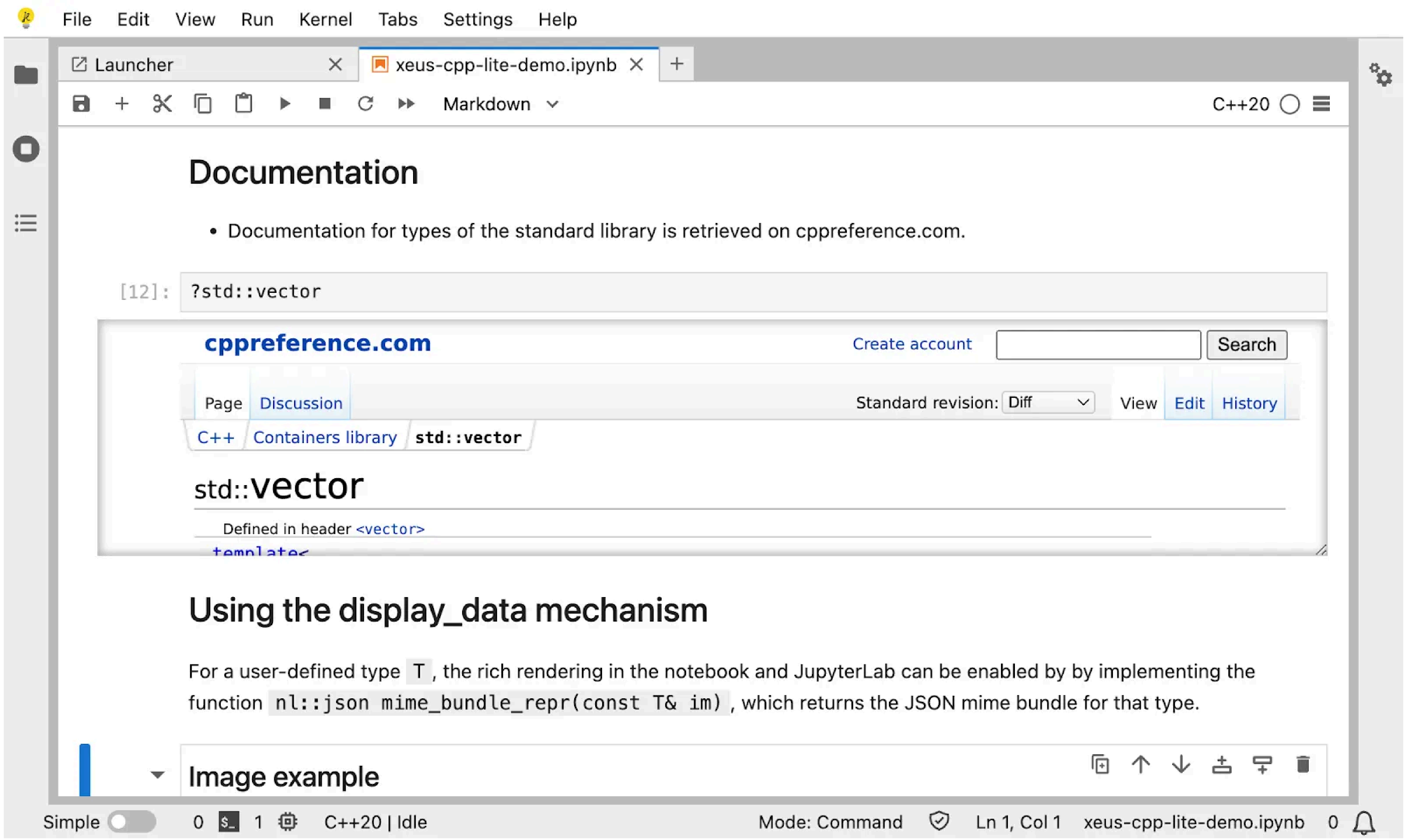


Demo & Use Cases

https://compiler-research.org/xeus-cpp-wasm/lab/index.html

- Basic C++
- Inline Documentation
- Rich Display
- Advanced Graphics
- Symbolic Computation with Symengine
- Array based Computing
- SIMD Acceleration
- Interactive Widgets
- Magic commands
    - %%file
    - %timeit
    - %mamba
- Loading 3rd party custom libraries

Inline Documentation



Rich Display

xeus-cpp-lite-demo.ipynb

Markdown

C++20

```cpp
    nl::json mime_bundle_repr(const image& i)
    {
        auto bundle = nl::json::object();
        bundle["image/png"] = xeus::base64encode(i.m_buffer.str());
        return bundle;
    }
}
```

[14]:
```cpp
im::image marie("marie.png");
```

[15]:
```cpp
#include "xcpp/xdisplay.hpp"
```

[16]:
```cpp
xcpp::display(marie);
```

Advanced Graphics

File Edit View Run Kernel Tabs Settings Help

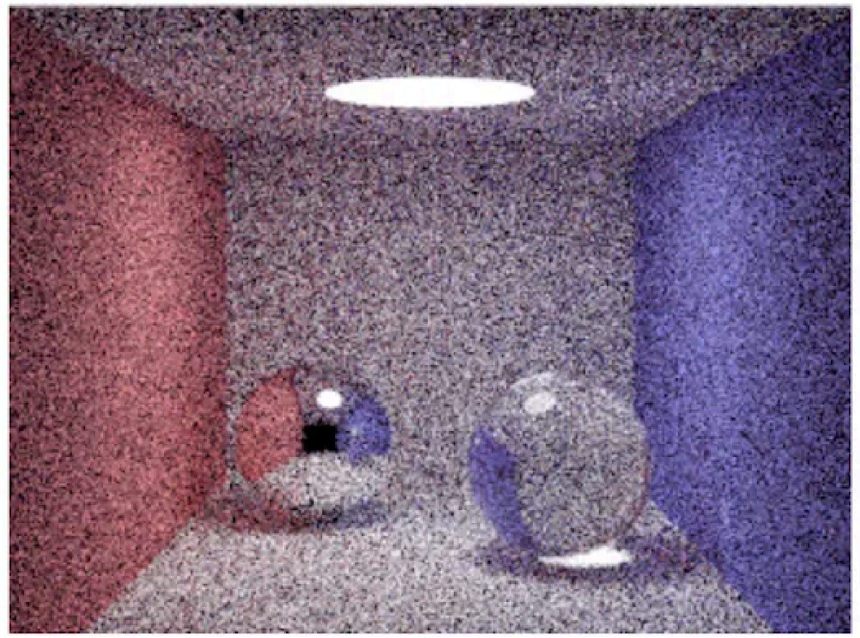
Untitled2.ipynb

Code

C++23

```
}

main();
```

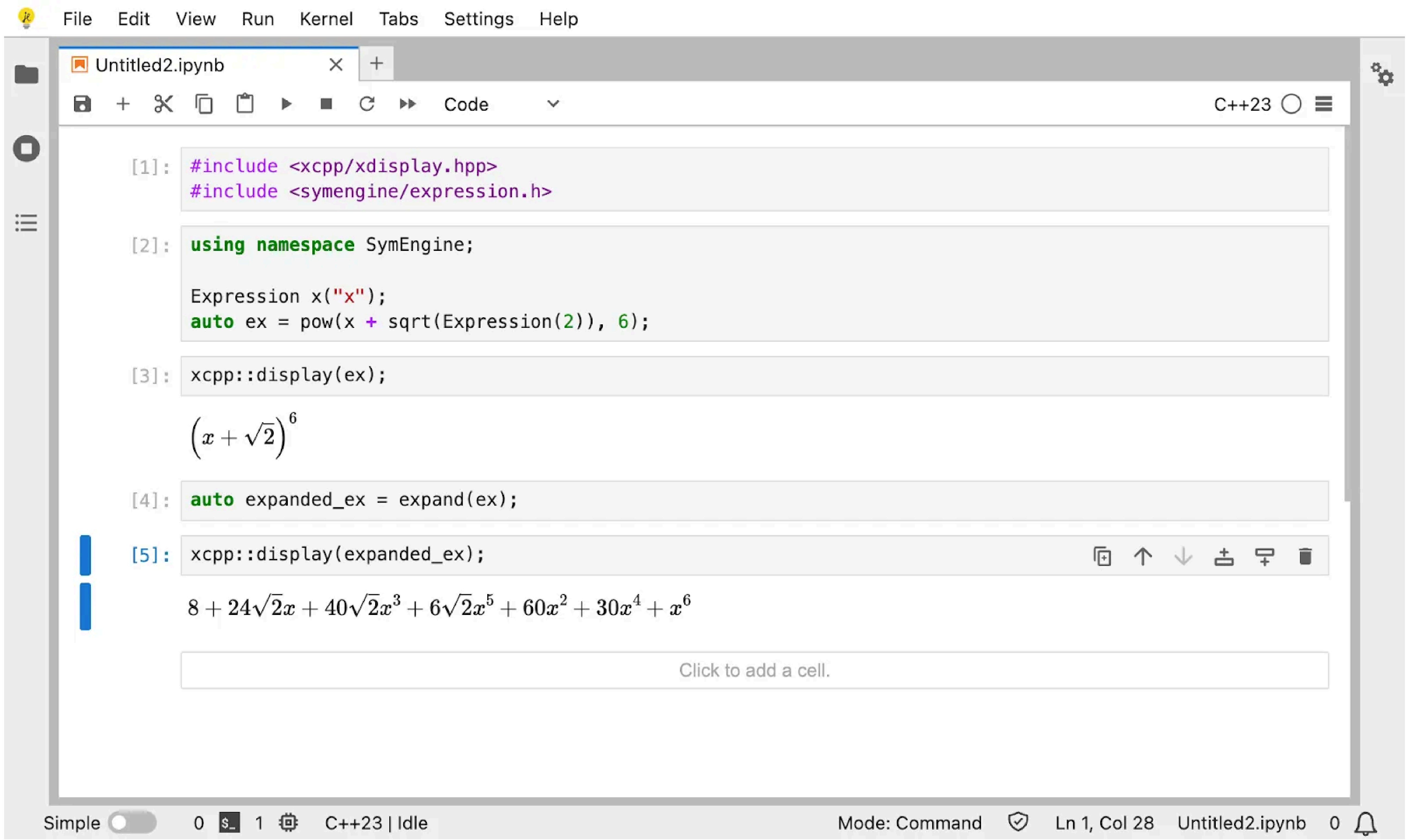
```
[DEBUG] Initializing SDL
[DEBUG] SDL initialized successfully
[DEBUG] Dimensions: 320x240, Samples: 16
[DEBUG] SDL surface created successfully
[DEBUG] The image should be on your screen soon
[DEBUG] Mapped pixel buffer to surface
[DEBUG] Surface saved to render.bmp
[DEBUG] Surface created successfully
```



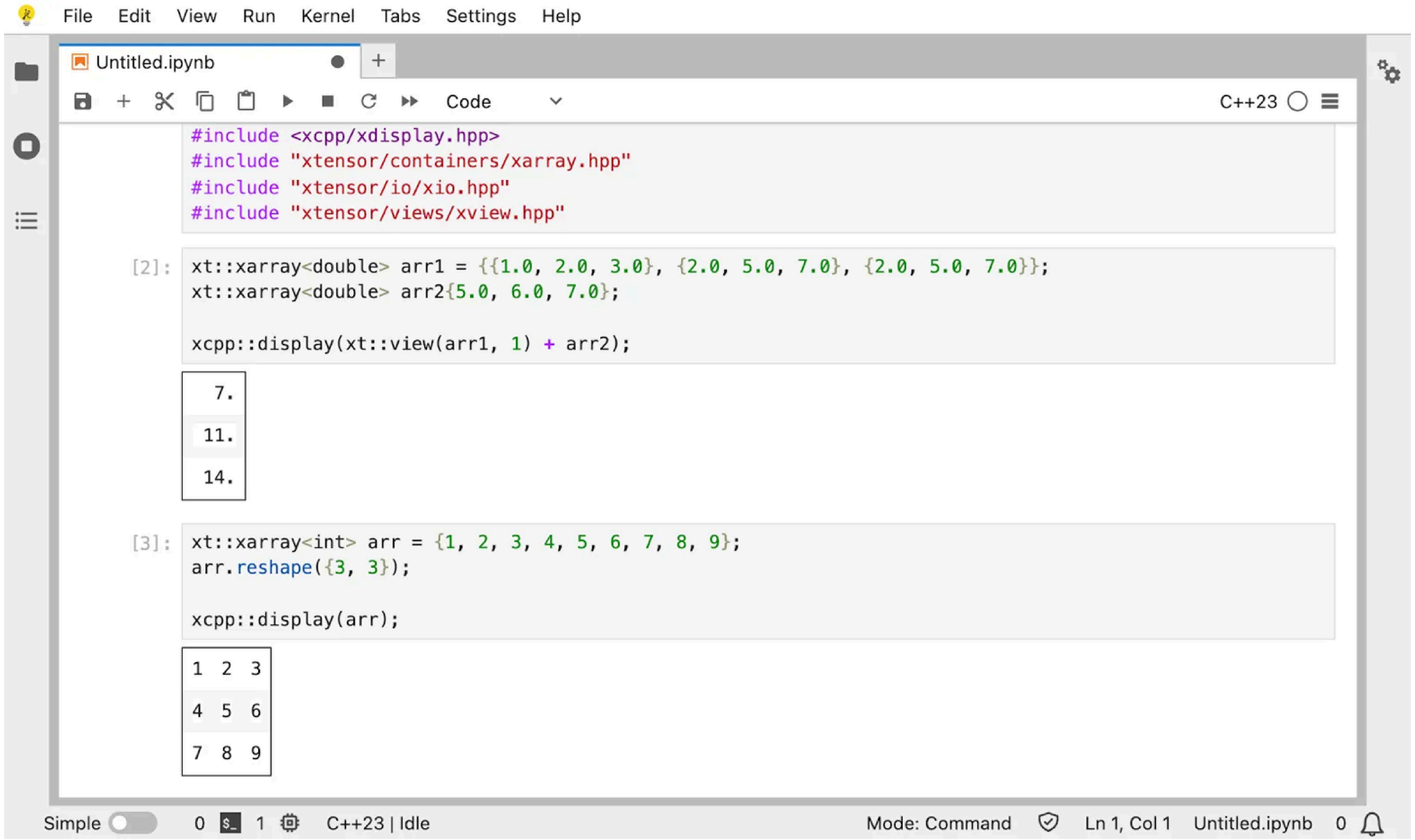
```
[DEBUG] Surface freed
[DEBUG] Exiting main
```

Simple 0 1 C++23 | Idle Mode: Command Ln 80, Col 8 Untitled2.ipynb 0

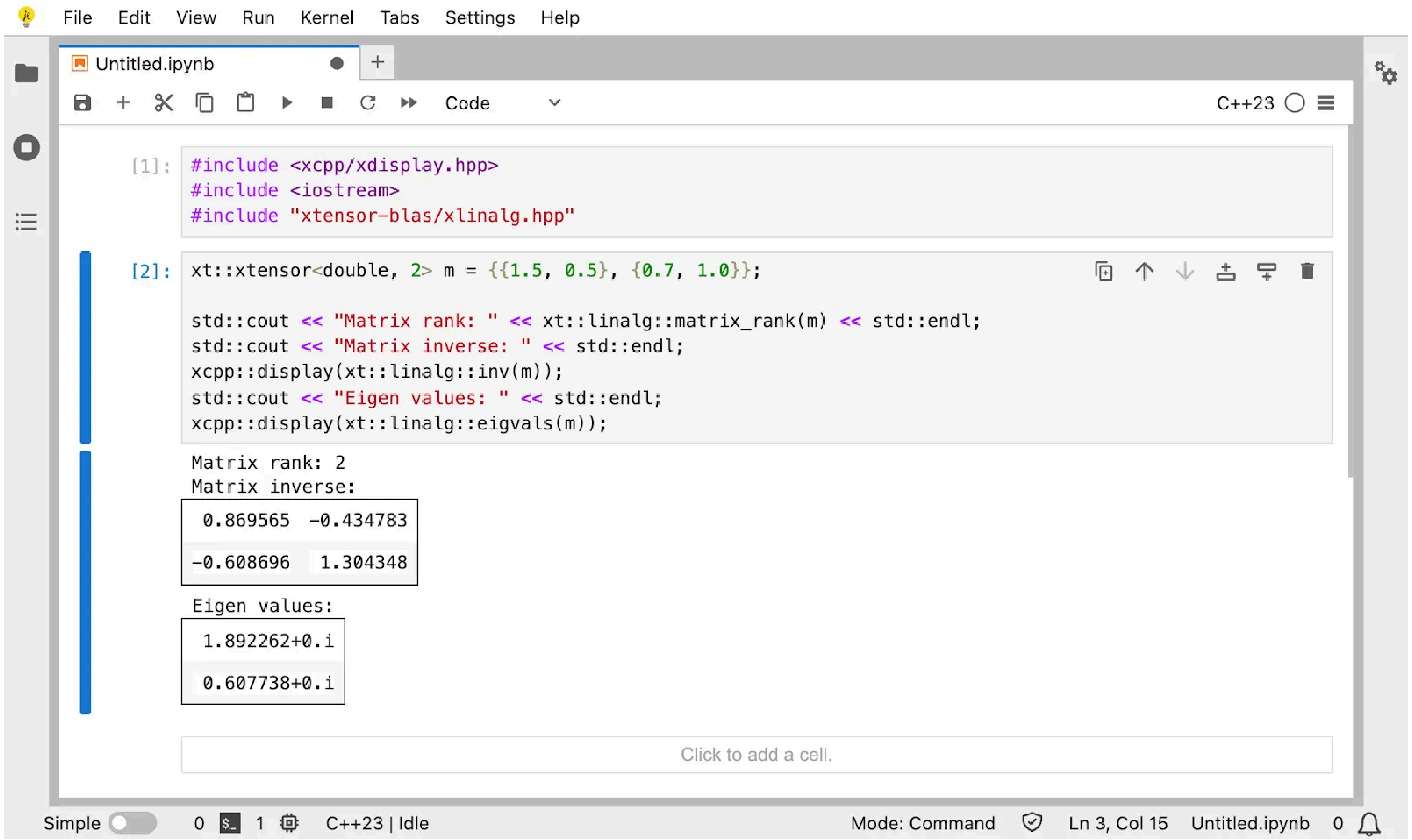
Symbolic Computation with Symengine



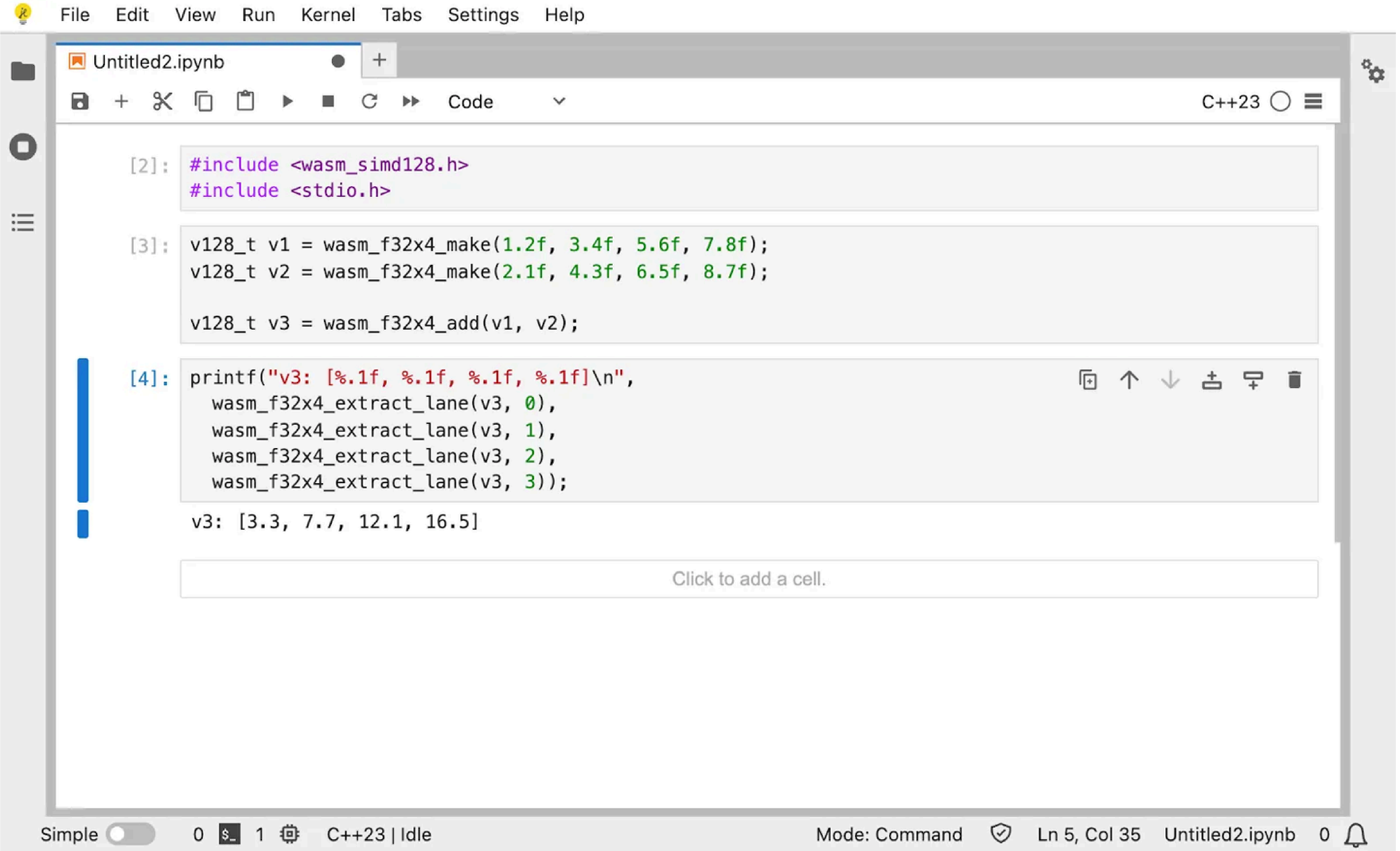
Array based Computing



Array based Computing



SIMD Acceleration



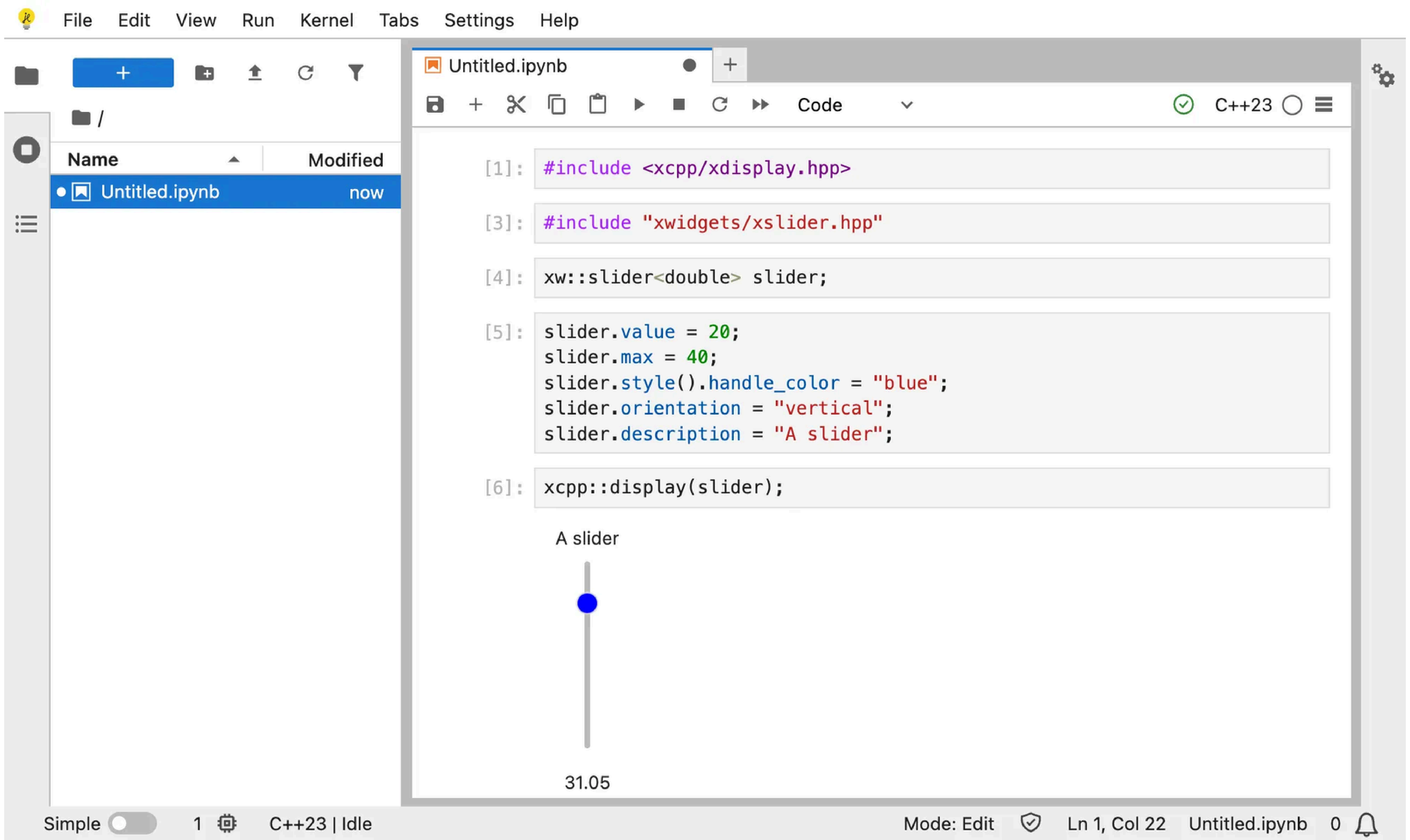
SIMD Acceleration

```cpp
#include <xsimd/xsimd.hpp>
#include <iostream>

namespace xs = xsimd;
```

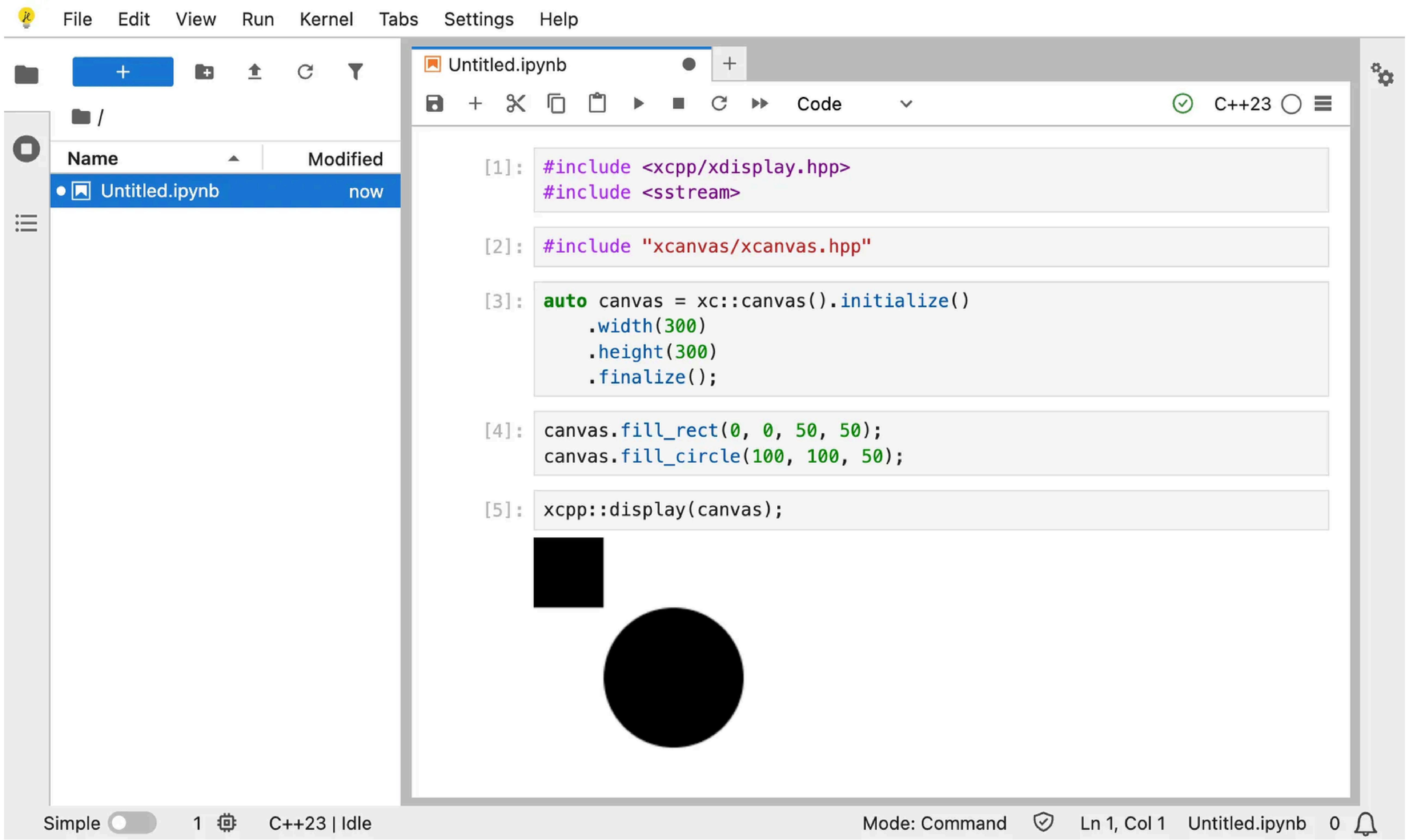
```cpp
// Define two SIMD float vectors using the wasm backend
xs::batch<float, xs::wasm> a = {1.2f, 3.4f, 5.6f, 7.8f};
xs::batch<float, xs::wasm> b = {2.1f, 4.3f, 6.5f, 8.7f};

// Perform SIMD addition
xs::batch<float, xs::wasm> c = a + b;

// Print the result
std::cout << c << std::endl;
```

```
(3.3, 7.7, 12.1, 16.5)
```

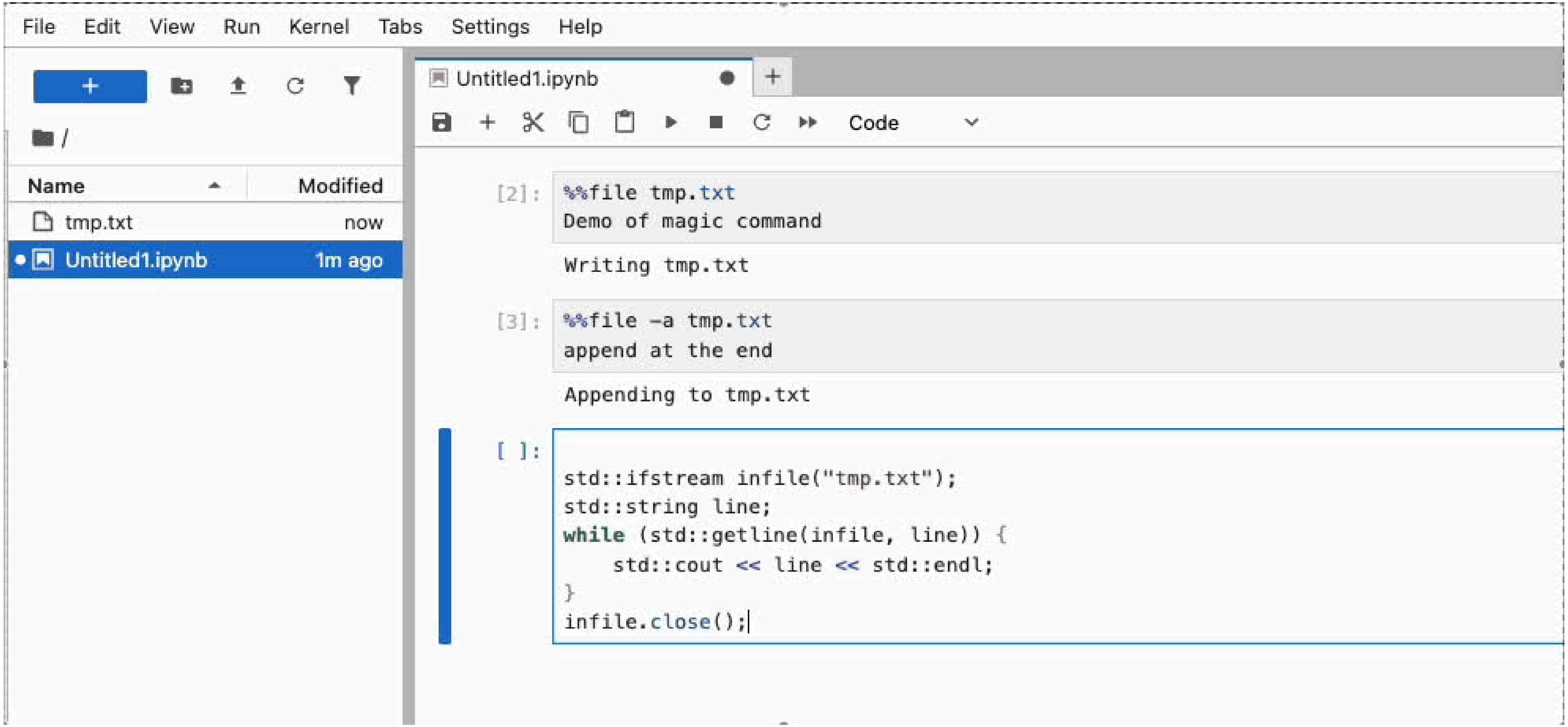
Interactive Widgets



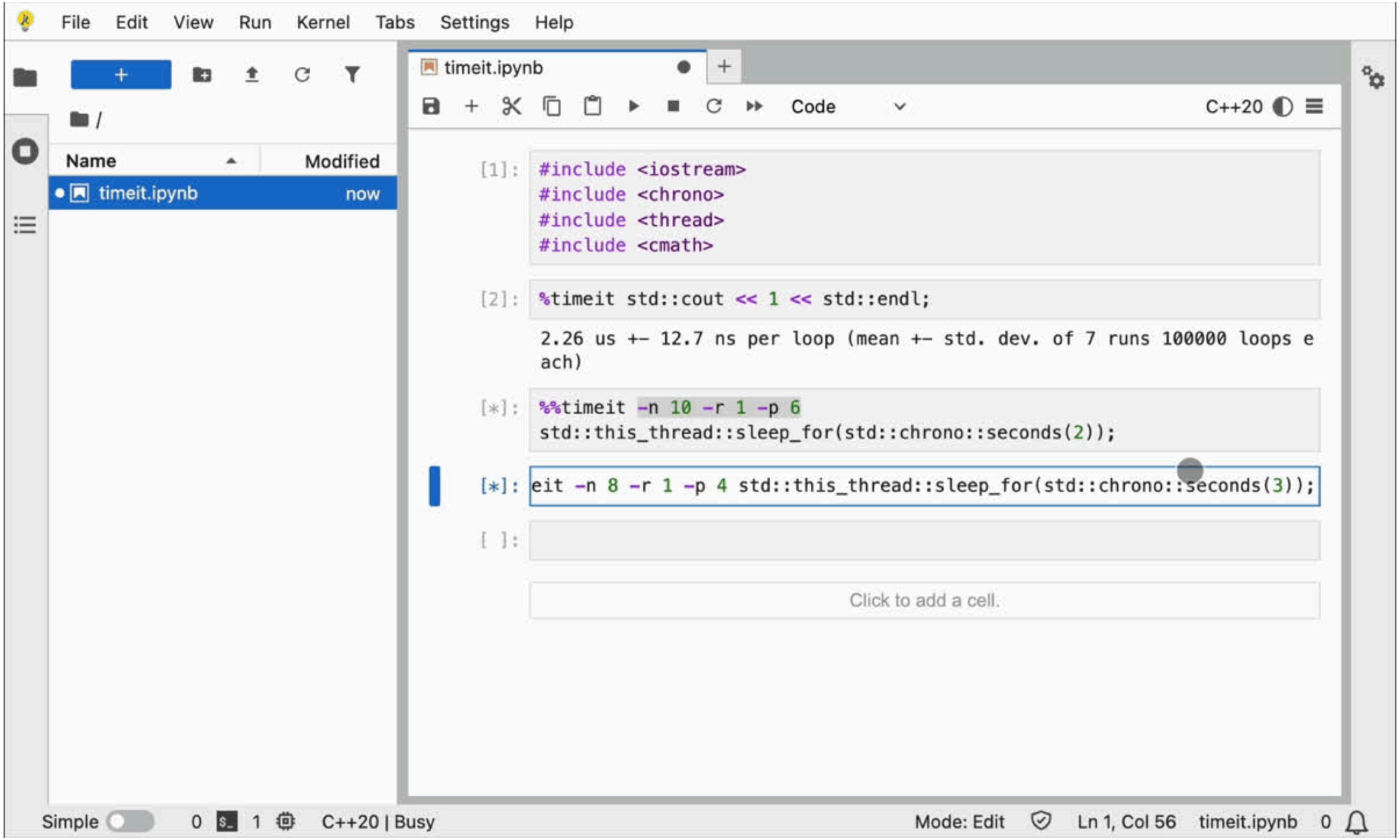
Interactive Widgets



Magic Commands



Magic Commands



Magic Commands

Untitled18.ipynb

Code | C++23

[1]:
```
%mamba install doctest
```

```
Specs: nlohmann_json=3.12.0, xeus-lite, xeus, CppInterOp, cpp-argparse, pugixml, doctest
Channels: https://prefix.dev/emscripten-forge-dev, https://prefix.dev/conda-forge

Solving environment...
Solving took 2.26439999999851 seconds
All requested packages already installed.
```

[2]:
```cpp
#define DOCTEST_CONFIG_IMPLEMENT
#include <doctest/doctest.h>

TEST_CASE("Simple check") {
    CHECK(1 == 2);
}

int main() {
    doctest::Context context;
    context.setOption("success", true); // Show successful tests
    return context.run();
}
```

[3]:
```cpp
main();
```

```
[doctest] doctest version is "2.4.12"
[doctest] run with "--help" for options
===============================================================================
input_line_3:4:
TEST CASE:  Simple check

input_line_3:5: ERROR: CHECK( 1 == 2 ) is NOT correct!
  values: CHECK( 1 == 2 )

===============================================================================
[doctest] test cases: 1 | 0 passed | 1 failed | 0 skipped
[doctest] assertions: 1 | 0 passed | 1 failed |
[doctest] Status: FAILURE!
```

Loading **third-party/custom** libs

1. Create a simple C++ module

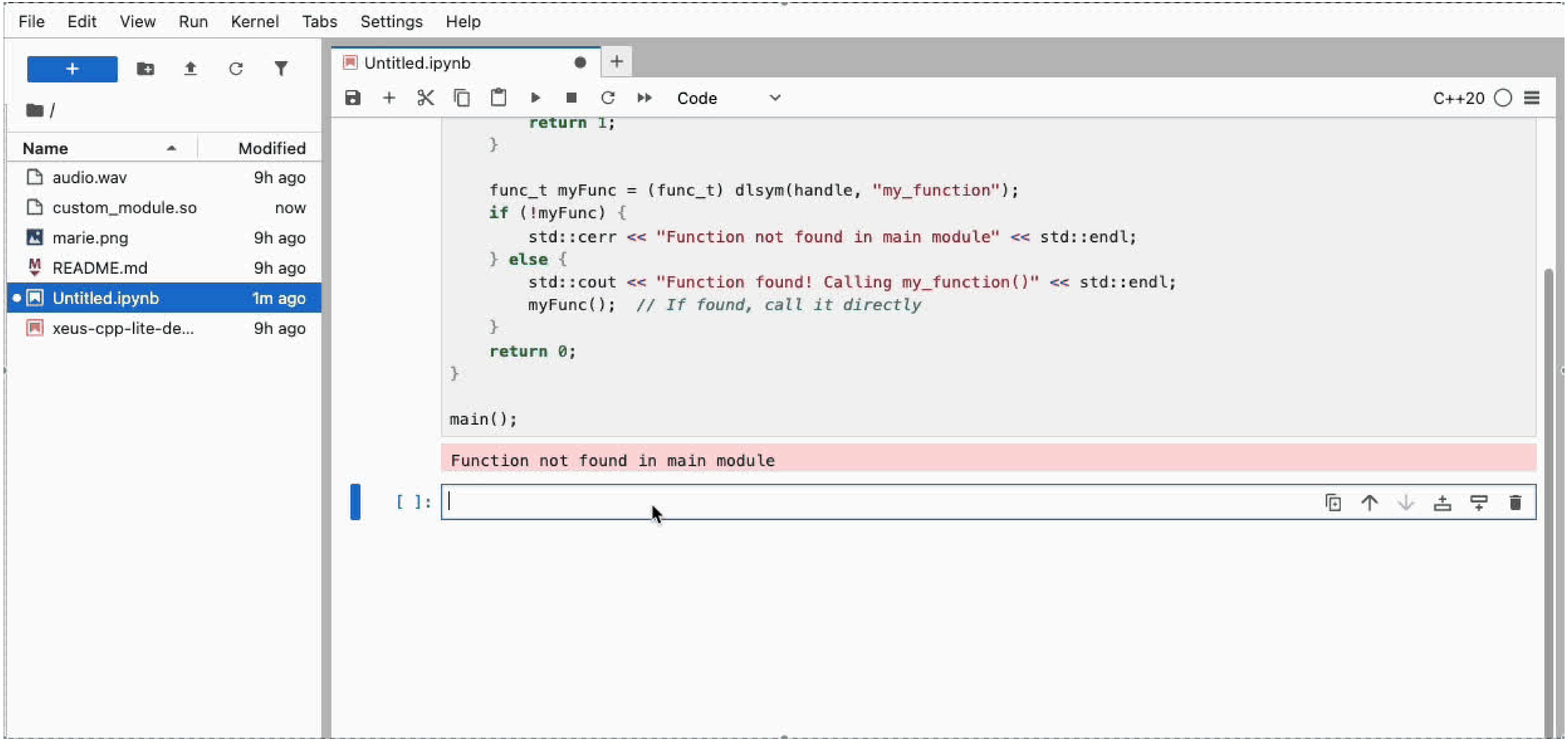
```cpp
// custom_module.cpp
#include <iostream>

extern "C" {
    void my_custom_function() {
        std::cout << "Hello from my_custom_function!" << std::endl;
    }
}
```

Loading **third-party/custom** libs

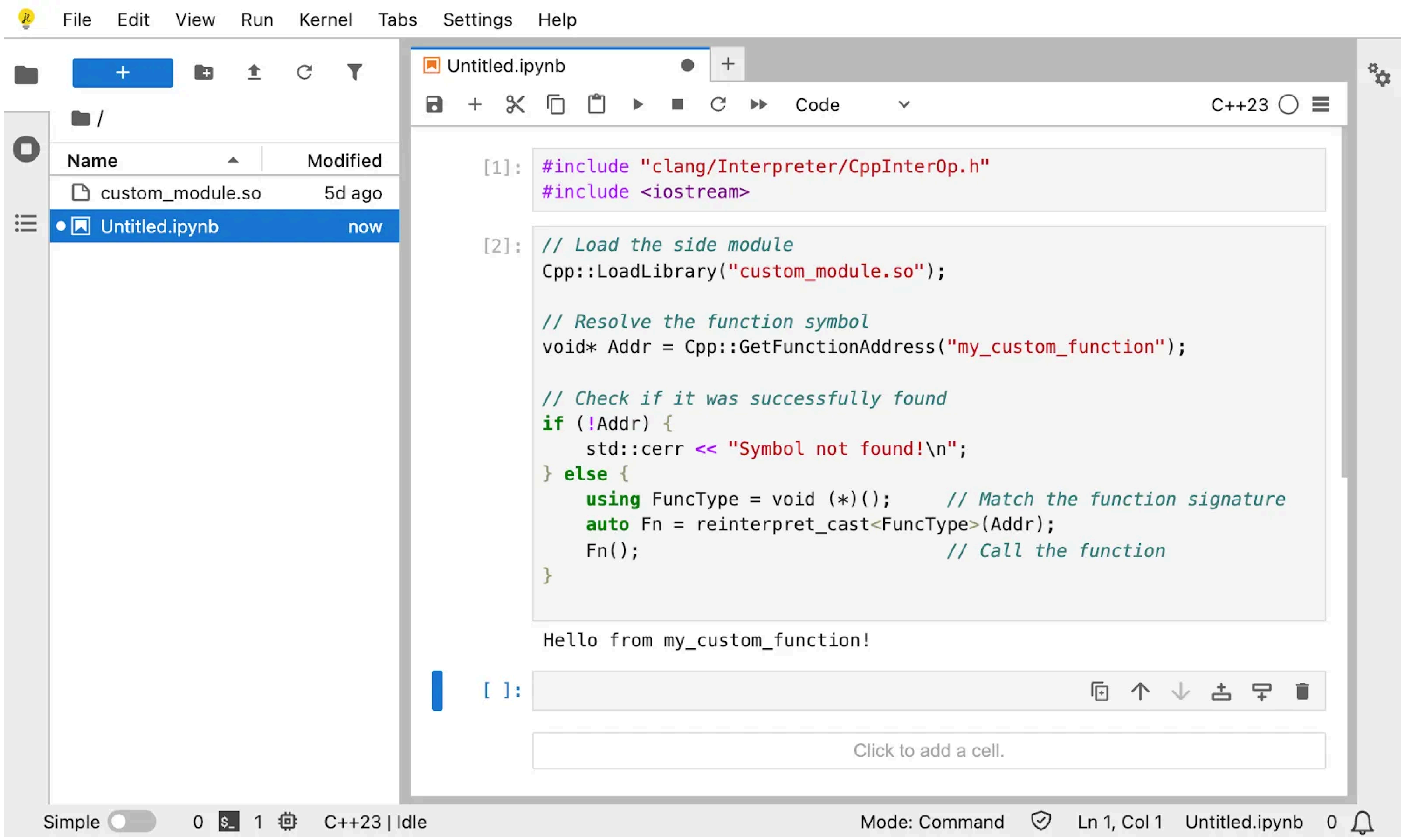
2. Compile it to a WebAssembly shared object

```
emcc custom_module.cpp \
-O2 \
-s SIDE_MODULE=1 \
-s WASM=1 \
-fPIC \
-o custom_module.so
```

Loading third-party/custom libs



Loading third-party/custom libs



Deploying Your Own Setup

https://github.com/jupyterlite/xeus-lite-demo : Template repo for creating a JupyterLite deployment on GitHub pages that includes the packages specified in a conda environment.

The process is as follows:

- Create a new repository from the GitHub template.
- Enable the deployment on GitHub pages from a GitHub action, as shown in the README.
- Edit the environment file to include the desired packages.

Deploying Your Own Setup

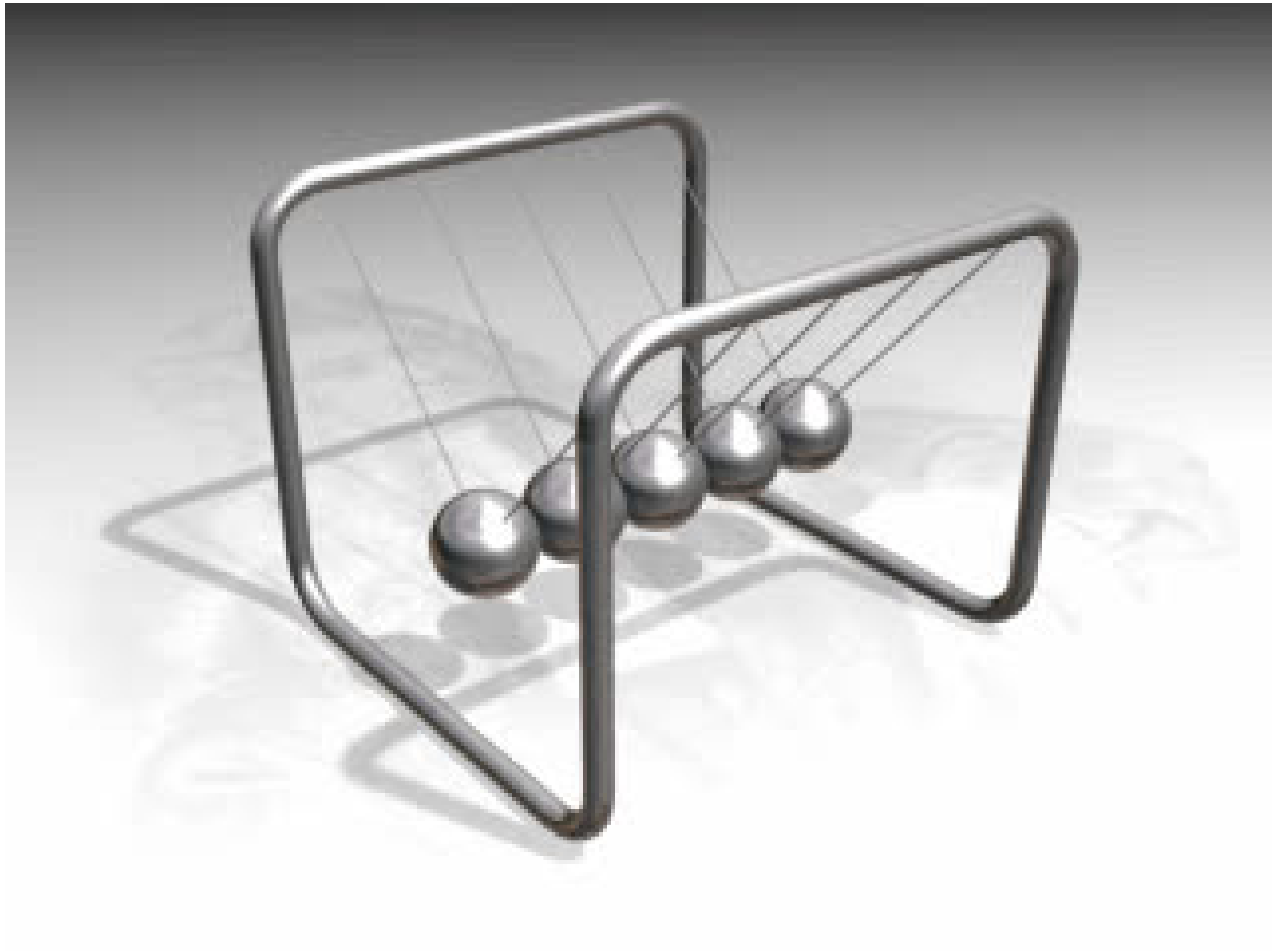
For example, to deploy a C++ kernel with Symengine & Xtensor-blas installed, the environment.yml file would contain the following:

```yaml
name: xeus-cpp
channels:
  - https://repo.prefix.dev/emscripten-forge-dev
  - conda-forge
dependencies:
  - xeus-cpp
  - symengine
  - xtensor-blas
```

Future Work (Near & Far)

- **Near**
  - Explore Off Screen canvas
  - Last Value Printing
  - Integration Testing
- **Far**
  - Multi language Hybrid Kernels
  - Integration with Juyterlite AI
  - Integration with Jupyterlite terminal
  - Debugger support for xeus-cpp-lite
  - Migrate from Emscripten-forge to Conda-forge

Off Screen Canvas



Last Value Printing

```
some output
```

```
In [2]: std::cerr << "some error" << std::endl;
```

```
some error
```

```
In [3]: #include <stdexcept>
```

```
In [4]: throw std::runtime_error("BAAAD");
```

```
Caught a std::exception!
BAAAD
```

Omitting the ; in the last statement of a cell gives an output

```
In [ ]: int i = 4

In [ ]: int j = 5;

In [ ]: j

In [ ]:

In [ ]:

In [ ]:
```

Integration Testing

Clang-repl + WebAssembly needs a robust Ci pipeline

- Currently, **LLVM lacks dedicated tests for Clang-Repl targeting WebAssembly**. This makes it difficult to detect regressions or behavior changes across LLVM versions.

- Why this matters: The migration from LLVM 19 to LLVM 20 introduced subtle breaking changes — especially in wasm-ld's defaults and symbol handling — which silently affected clang-repl's behavior in the browser.

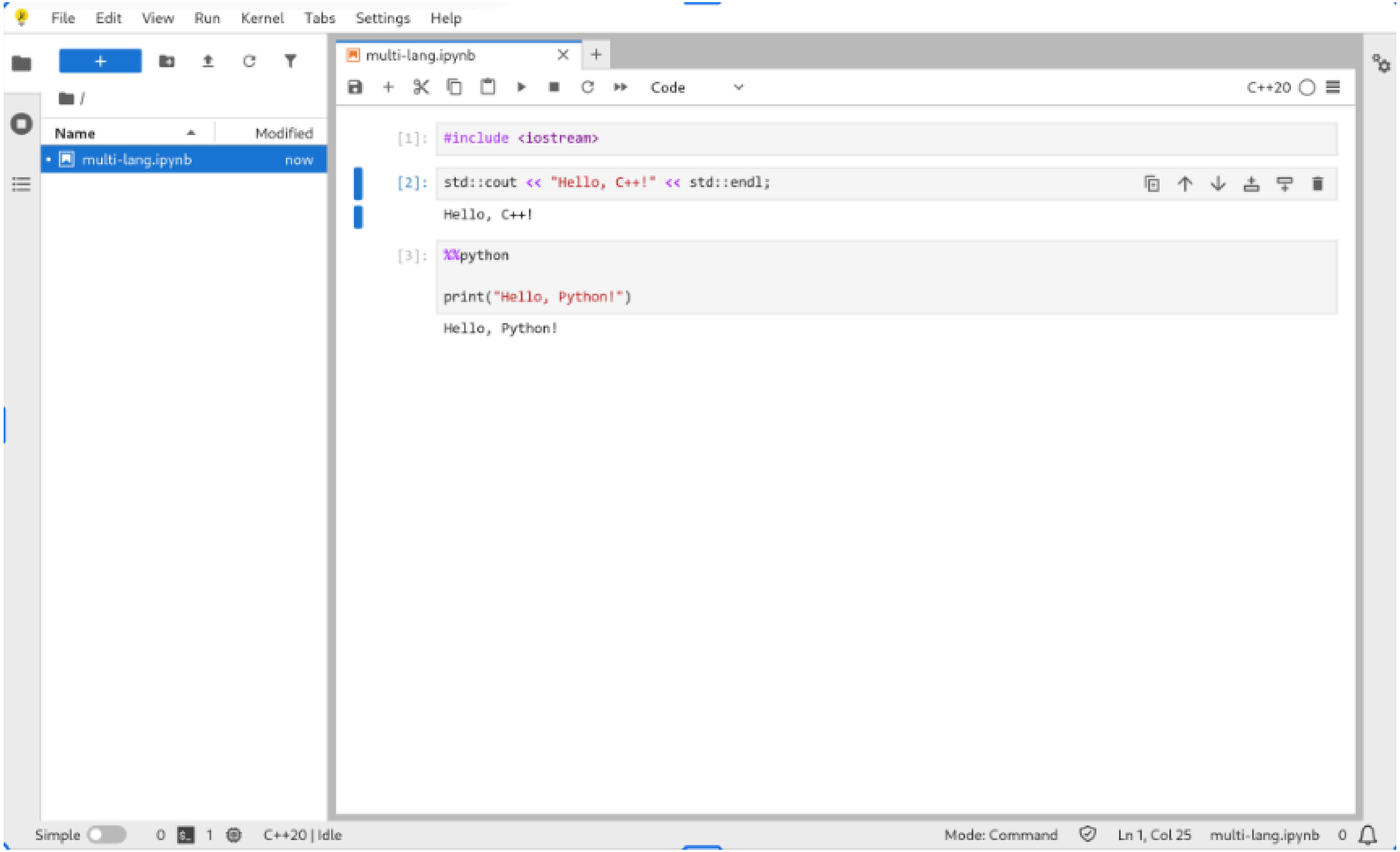
- Without proper integration tests, such changes go undetected until runtime — sometimes deep into downstream projects like Xeus-Cpp-Lite.

- **Today, testing is only indirect**:

    - Through **CppInterOp's Emscripten test suite**, which exercises Clang-Repl in a browser-like setting.

    - Through **Xeus-Cpp’s kernel-level Emscripten build**.

Multi Language Hybrid Kernels



Integration with JupyterLite AI

- Repo : https://github.com/jupyterlite/ai
- Tharun Anandh added LLM support for native kernels through GSoC 2024

```
[3]: %%xassist gemini
     write a cpp code to get square of a number
```

````
Escaped: write a cpp code to get square of a number
```c++
#include <iostream>

using namespace std;

int main() {
  int number;

  // Get the number from the user
  cout << "Enter a number: ";
  cin >> number;

  // Calculate the square
  int square = number * number;

  // Display the result
  cout << "The square of " << number << " is: " << square << endl;

  return 0;
}
```
````

Integration with JupyterLite Terminal

- Repo : https://github.com/jupyterlite/terminal
- Fairly experimental and not ready for general use
- But should address lot of use cases once ready

Debugger Support for Xeus-Cpp-Lite

- **Native Debugging (In Progress)** : As part of GSoC 2025, I'm mentoring Abhinav, who's working on implementing debugger support for native Xeus-Cpp using LLDB and lldb-dap.

- We’ve made solid progress — we can now:
  - Set breakpoints
  - Step into and out of functions
  - Work in progress: variable inspection, stepping over, call stacks, etc.

- **Debugging in the Browser (Very Early Exploration)**

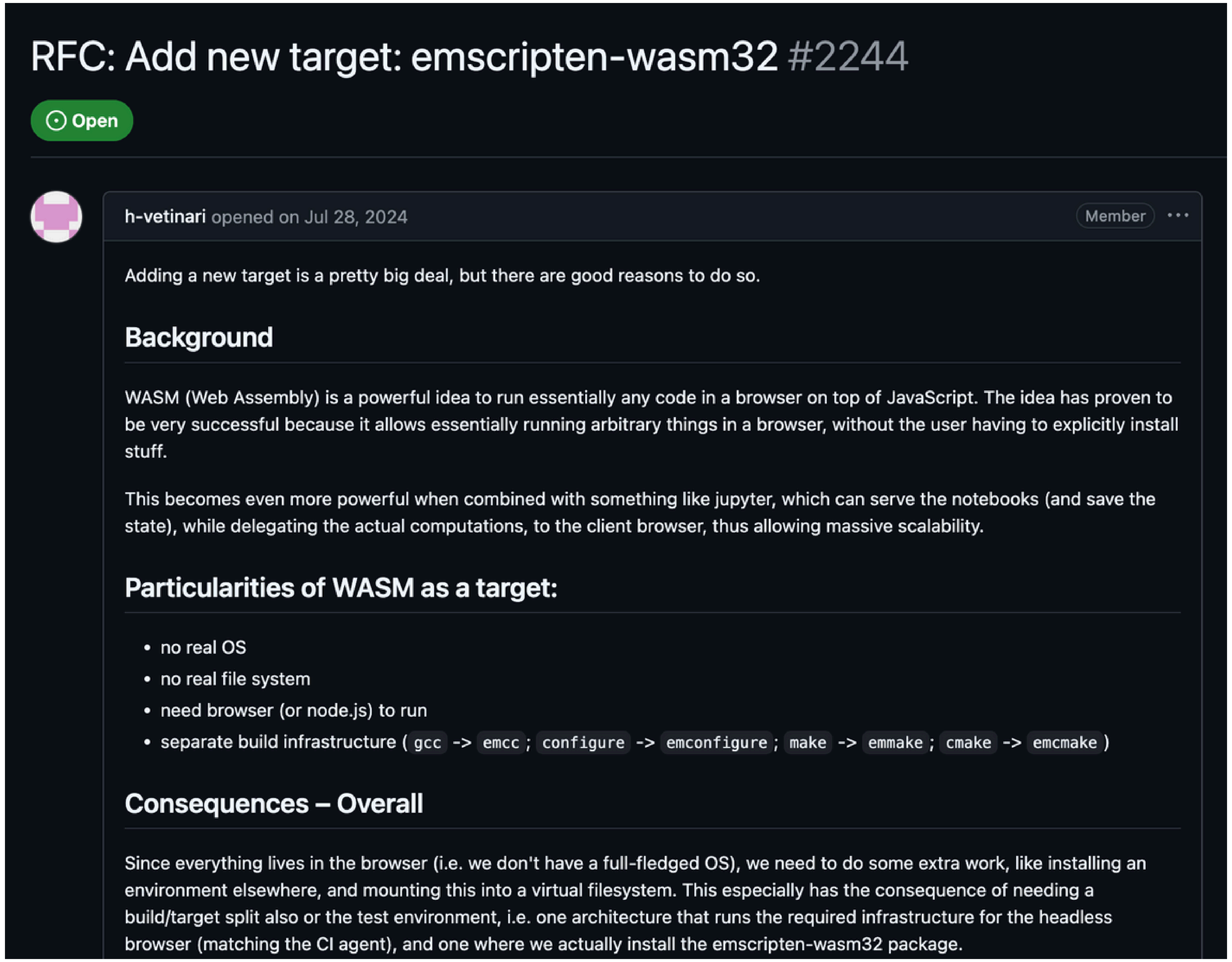
  - Bringing debugger support to WebAssembly (WASM) is a bigger challenge — especially in the browser.
  - Jonas Devlieghere (lead LLDB developer), who’s actively improving LLDB + lldb-dap for WASM.
  - https://jonasdevlieghere.com/post/wasm-debugging/

Migrate from **Emscripten-forge** to **Conda-forge**

Unifying package ecosystems: Our long-term goal is to migrate from emscripten-forge to conda-forge, avoiding the duplication of recipes across both platforms.

Why it matters:

- Easier maintenance
- Better integration with the broader scientific Python ecosystem
- Simplifies WASM support for downstream package authors



RFC: Add new target: emscripten-wasm32 #2244

Open

h-vetinari opened on Jul 28, 2024 Member

Adding a new target is a pretty big deal, but there are good reasons to do so.

Background

WASM (Web Assembly) is a powerful idea to run essentially any code in a browser on top of JavaScript. The idea has proven to be very successful because it allows essentially running arbitrary things in a browser, without the user having to explicitly install stuff.

This becomes even more powerful when combined with something like jupyter, which can serve the notebooks (and save the state), while delegating the actual computations, to the client browser, thus allowing massive scalability.

Particularities of WASM as a target:

- no real OS
- no real file system
- need browser (or node.js) to run
- separate build infrastructure (`gcc` -> `emcc` ; `configure` -> `emconfigure` ; `make` -> `emmake` ; `cmake` -> `emcmake`)

Consequences – Overall

Since everything lives in the browser (i.e. we don't have a full-fledged OS), we need to do some extra work, like installing an environment elsewhere, and mounting this into a virtual filesystem. This especially has the consequence of needing a build/target split also or the test environment, i.e. one architecture that runs the required infrastructure for the headless browser (matching the CI agent), and one where we actually install the emscripten-wasm32 package.

Acknowledgements

Vassil Vassilev	LLVM, cling, clang-repl, CppInterOp, Xeus-Cpp
Sylvain Corlay, Johan Mabille, Loic Gouarin	Xeus, Xeus-cling, Xeus-Cpp
Anubhab Ghosh	Initial proof for clang-repl in the browser
Thorsten Beier	Emscripten-forge, Xeus-lite
Jeremy Tuloup	Jupyterlite, Jupyterlite/ai, Jupyterlite/terminal
Martin Renou, Anastasiia Sliusar	Jupyterlite-Xeus, Empack, Mambajs
Matthew Barton, Tharun Anandh, Abhinav Kumar	Significant contributions to Xeus-Cpp & CppInterOp