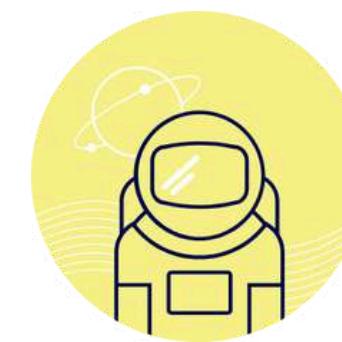


# Xeus-Cpp-Lite

Interpreting C++ in the Browser

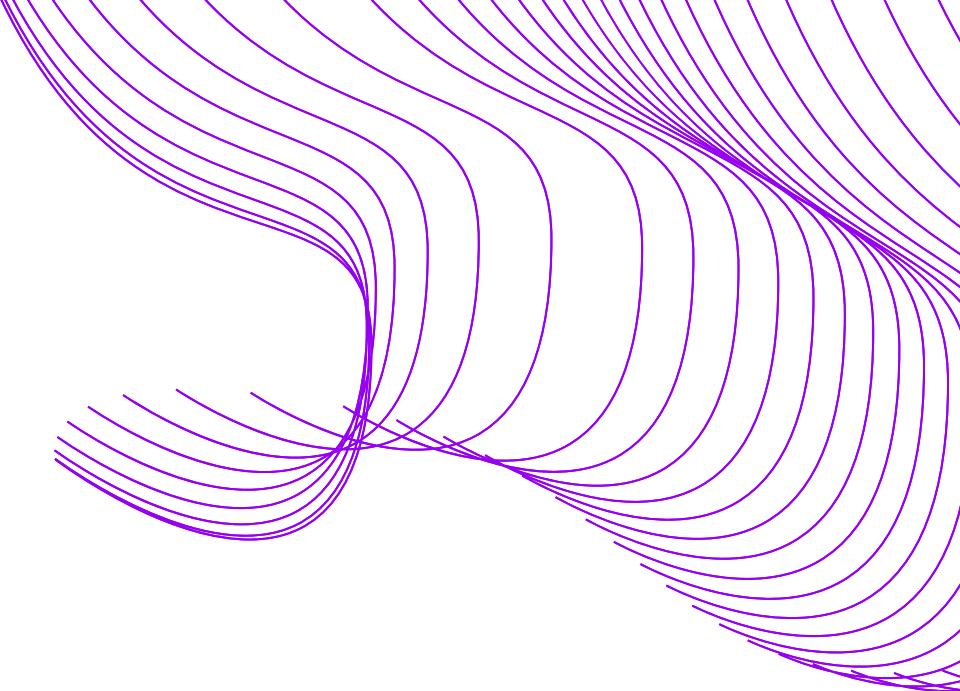


QuantStack  
Scientific Computing



# Agenda

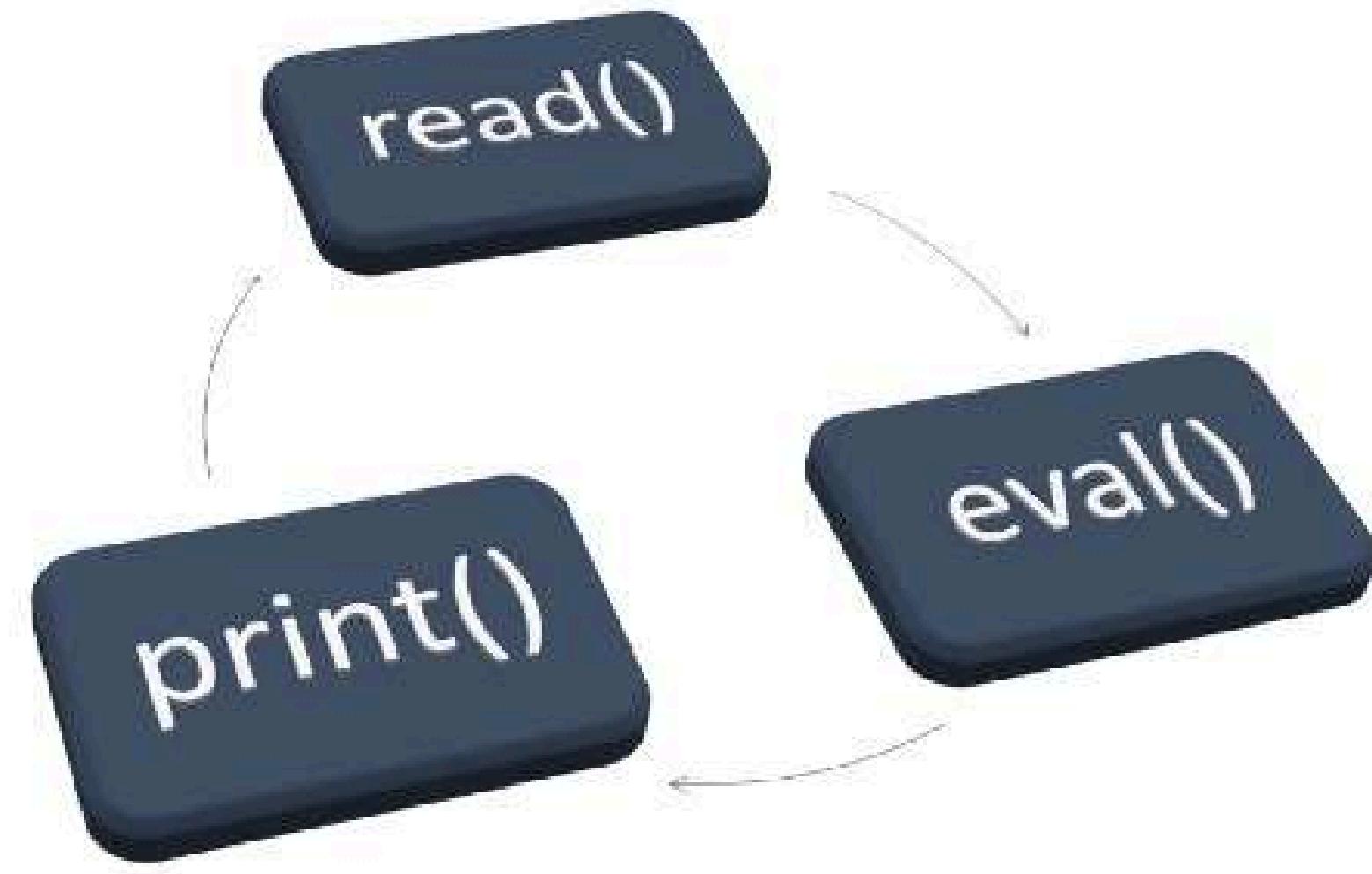
- Motivation: Why C++ Needs a REPL
- Native Case: Xeus-Cpp via Clang-Repl
- Enter the Browser: JupyterLite & WebAssembly
- Initial Proof of Concept: Clang-Repl in the browser
- Bringing it All Together: Xeus-Cpp-Lite
- Demos & Use Cases
- Deploying Your Own Setup
- Future work (Near and Far)
- Acknowledgments



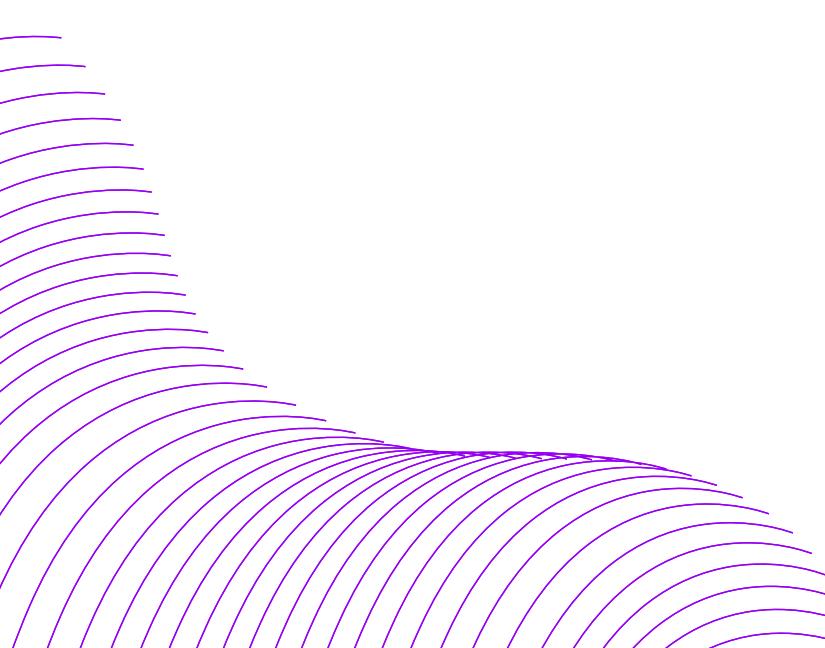
# **Xeus-Cpp-Lite = Xeus-Cpp + JupyterLite**

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
```

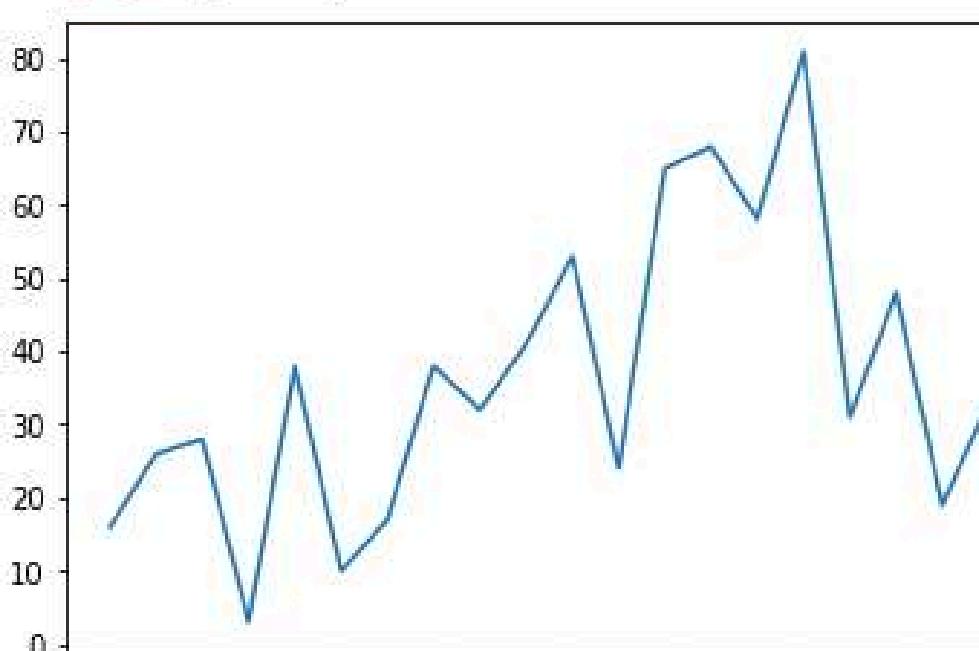
```
In [2]: import matplotlib.pyplot as plt
```

```
In [3]: import random
```

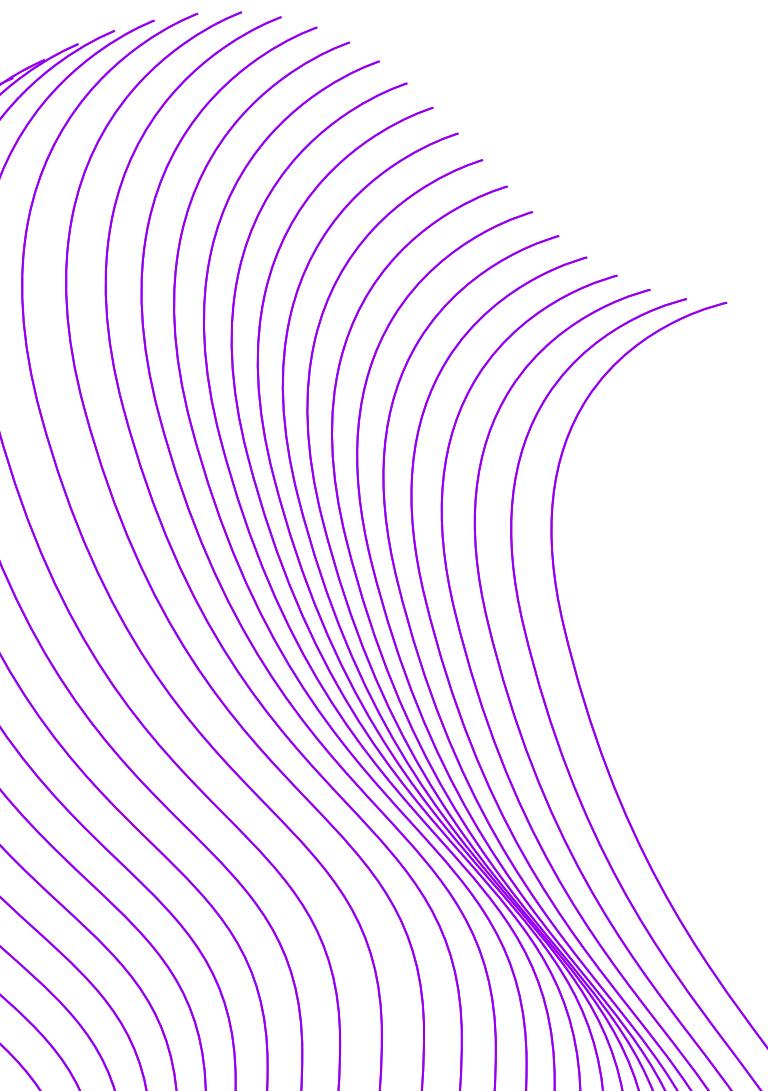
```
In [4]: data = [random.randint(0, 100) for i in range(20)]
```

```
In [5]: plt.plot(data)
```

```
Out[5]: [
```



# 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 its 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

JupyterLab C++17 Binder GitHub Download

```
    std::stringstream m_buffer,
};

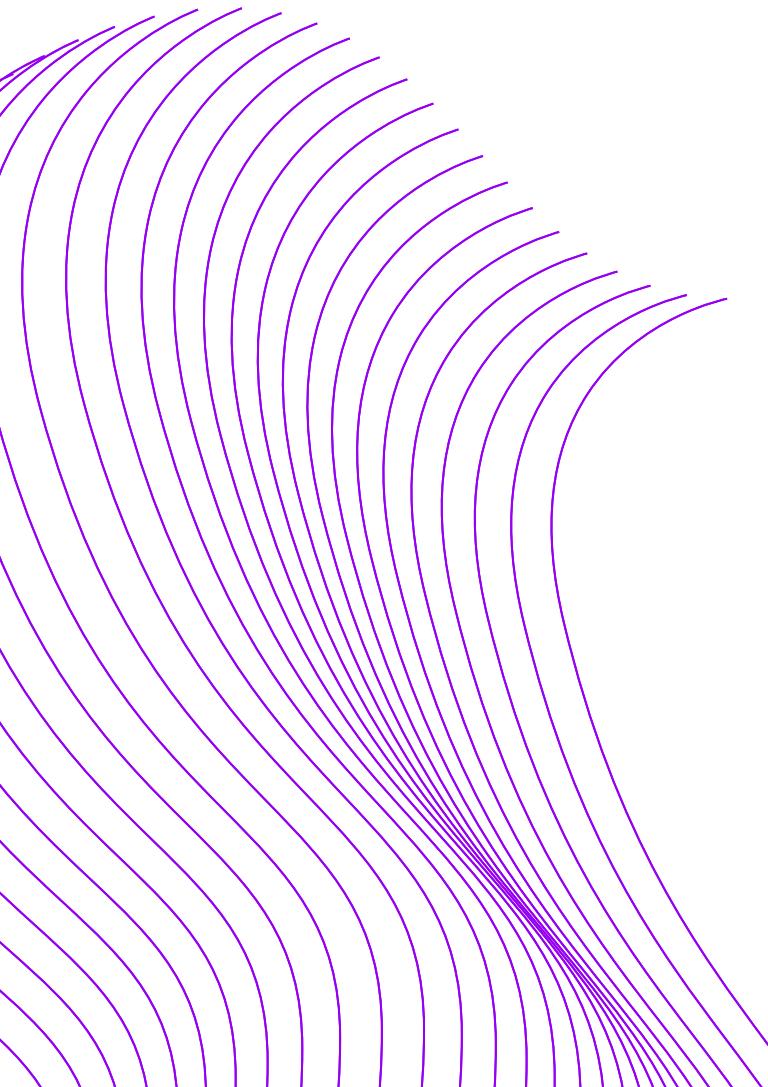
nl::json mime_bundle_repr(const image& i)
{
    auto bundle = nl::json::object();
    bundle["image/png"] = xt::base64encode(i.m_buffer.str());
    return bundle;
}
}

[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.

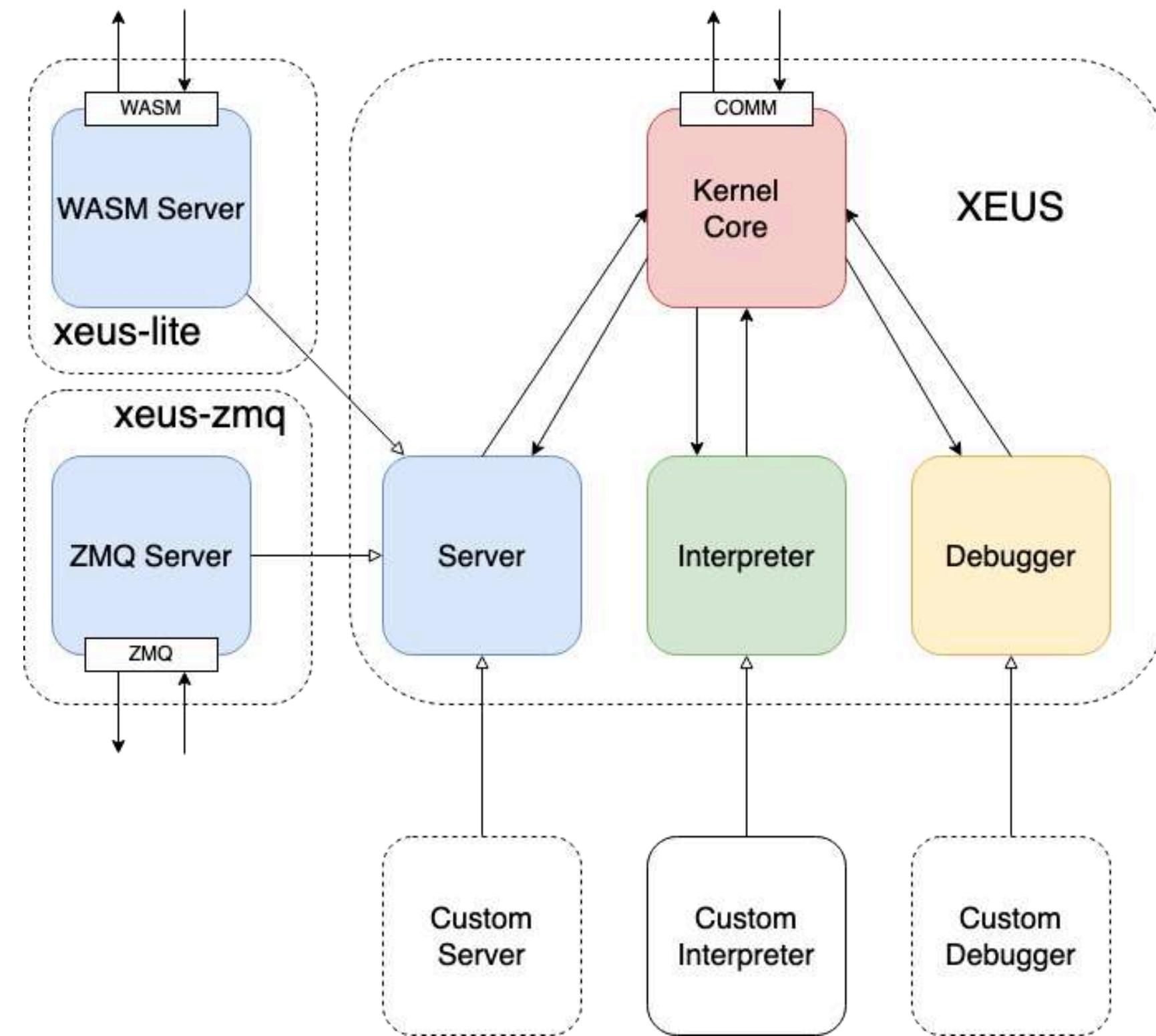
## CppInterOp

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

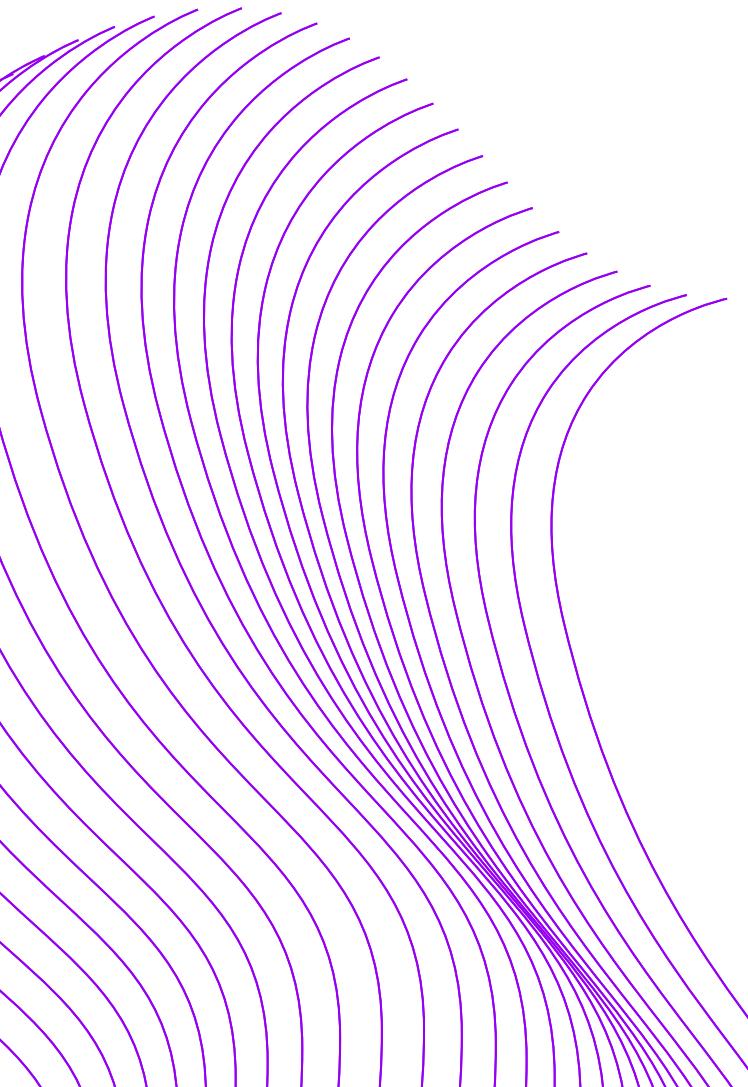
## Xeus-Cpp

- 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.

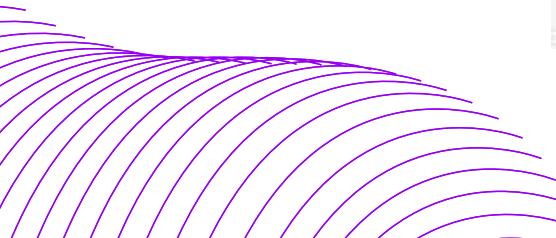
## Versatile Kernel Support

- 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



```
▶ C ⚡ 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]: 
$$x - \frac{x^3}{6} + \frac{x^5}{120} + O(x^7)$$

```

[ ]:

**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.

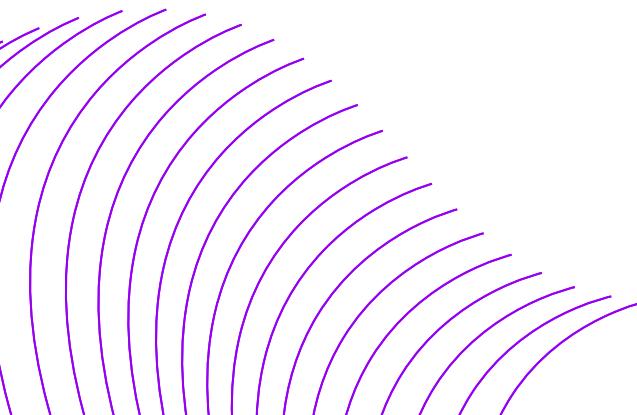
## Enter Emscripten-Forge

- 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

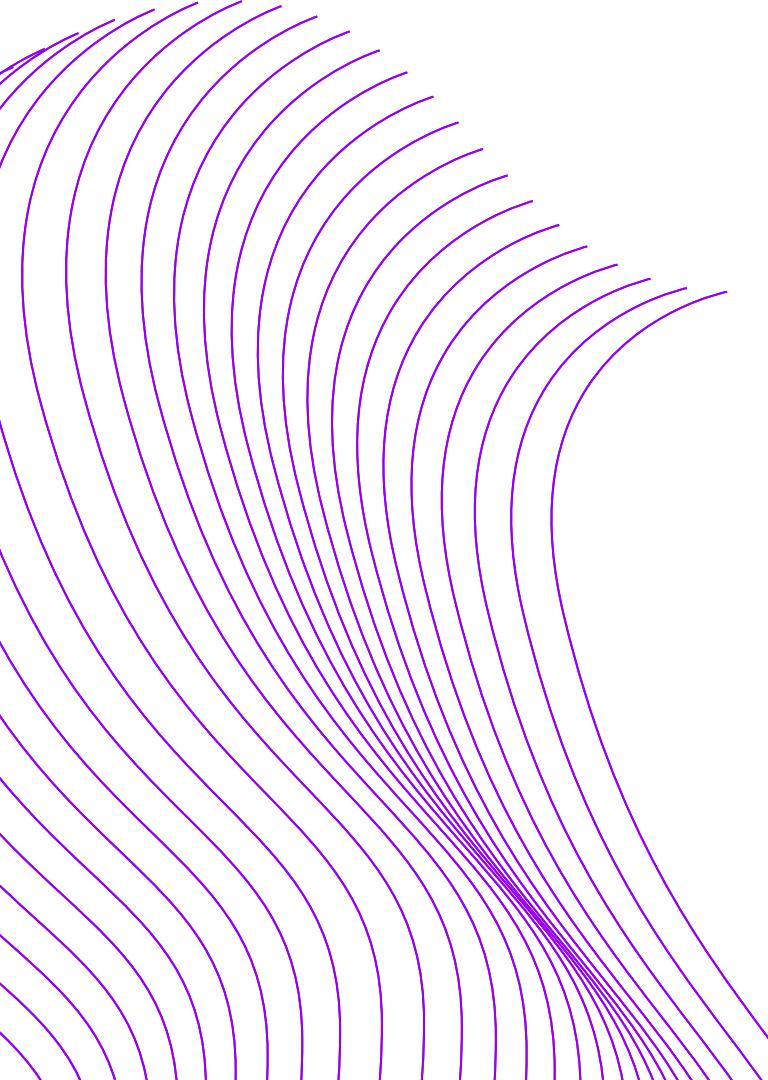
# Problems using **clang-repl** in the **browser** through **LLVM 19**

```
[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
```

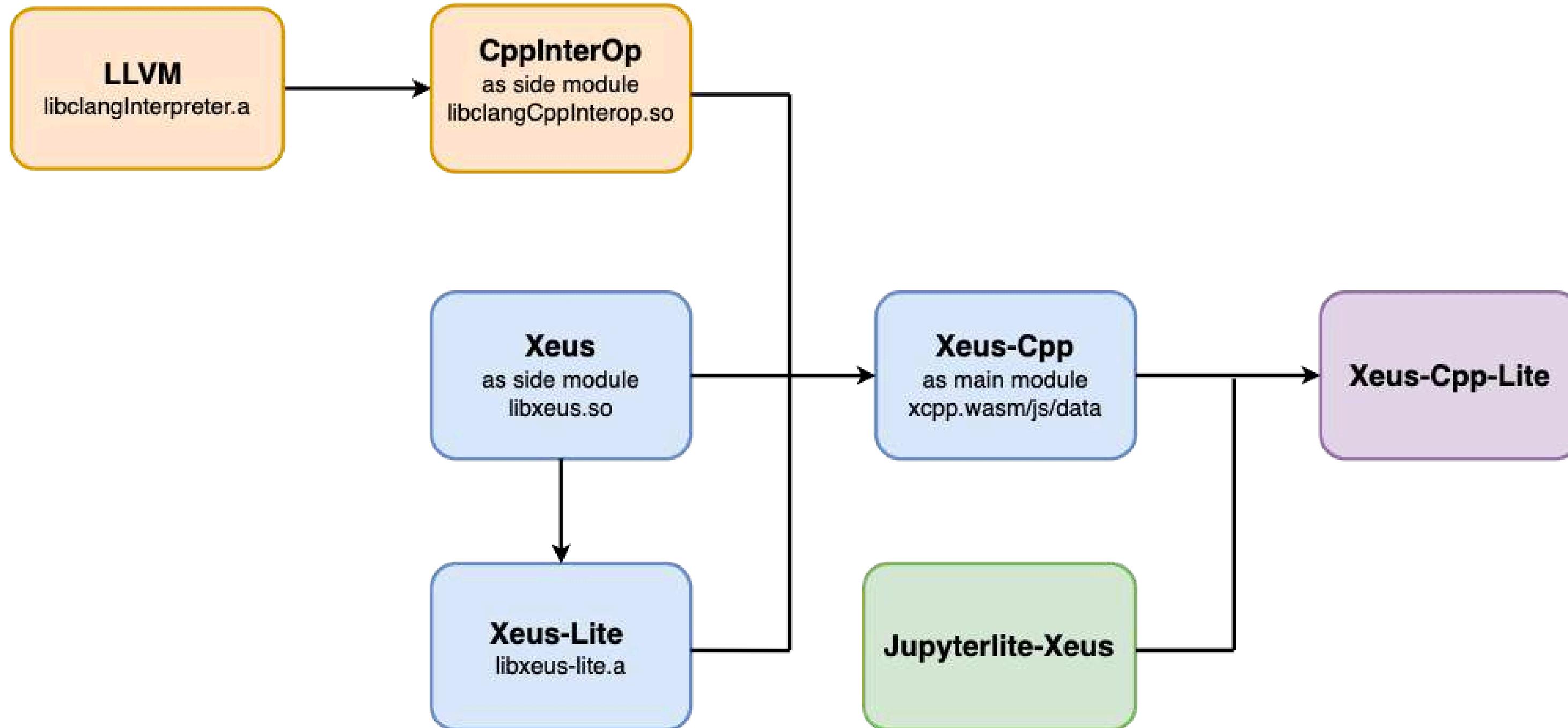
# List of Pull Requests :

## 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

The screenshot shows a JupyterLab interface with the following details:

- File Bar:** File, Edit, View, Run, Kernel, Tabs, Settings, Help.
- Tab Bar:** Launcher (closed), xeus-cpp-lite-demo.ipynb (active), +.
- Toolbar:** File, +, X, Open, Save, Run, Kernel, C, Markdown (selected), C++20, Mode: Command.
- Section Header:** Documentation
- List Item:** Documentation for types of the standard library is retrieved on [cppreference.com](http://cppreference.com).
- Code Cell:** [12]: ?std::vector
- Preview Area:** A screenshot of the cppreference.com page for std::vector. It shows the navigation bar with Page selected, Discussion, Standard revision: Diff, View, Edit, History, and C++ Containers library std::vector. The main content is titled "std::vector" and defines it as a template defined in header <vector>. The preview area has scroll bars.
- Section Header:** Using the `display_data` mechanism
- Text:** For a user-defined type `T`, the rich rendering in the notebook and JupyterLab can be enabled by implementing the function `nl::json mime_bundle_repr(const T& im)`, which returns the JSON mime bundle for that type.
- Section Header:** Image example
- Bottom Bar:** Simple (button), 0 \$ 1 ⚙️ C++20 | Idle, Mode: Command, ⚡ Ln 1, Col 1 xeus-cpp-lite-demo.ipynb 0, Bell icon.

# Rich Display

xeus-cpp-lite-demo.ipynb ● +

File + X Close □ ▶ ■ C ▶ Markdown ▾ C++20 ⚙

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

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

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



# Advanced Graphics

S Jupyter Notebook interface showing the output of a C++ code execution.

The notebook tab bar shows "Untitled2.ipynb".

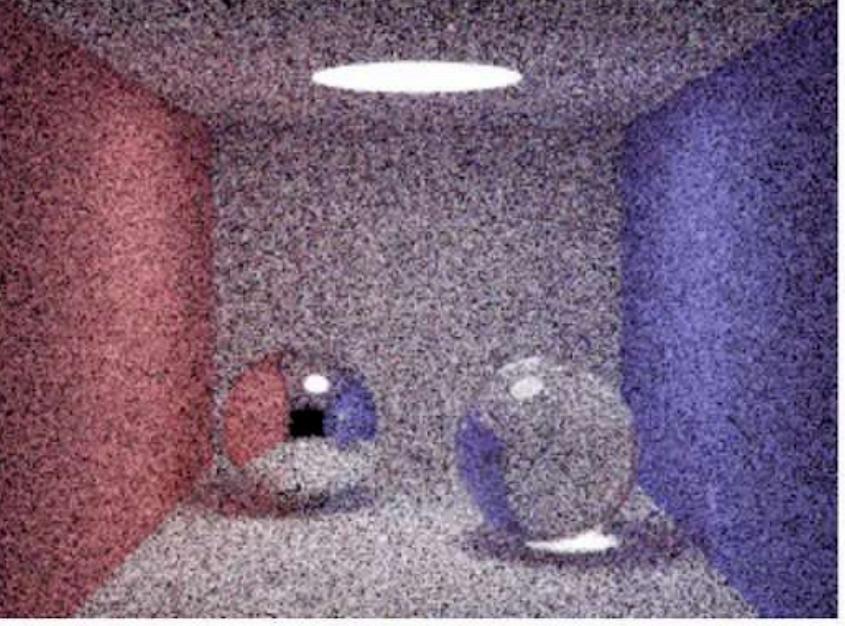
The toolbar includes icons for file operations (+, X, folder, square, circle), a search bar, and a "Code" dropdown.

The status bar at the bottom shows "Mode: Command" and "Ln 80, Col 8 Untitled2.ipynb 0".

The main area displays the following text and image:

```
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

# Symbolic Computation with Symengine

The screenshot shows a Jupyter Notebook interface with the following details:

- File Bar:** File, Edit, View, Run, Kernel, Tabs, Settings, Help.
- Toolbar:** Untitled2.ipynb, +, Code, C++23.
- Code Cells:**
  - [1]: 

```
#include <xcpp/xdisplay.hpp>
#include <symengine/expression.h>
```
  - [2]: 

```
using namespace SymEngine;

Expression x("x");
auto ex = pow(x + sqrt(Expression(2)), 6);
```
  - [3]: 

```
xcpp::display(ex);


$$(x + \sqrt{2})^6$$

```
  - [4]: 

```
auto expanded_ex = expand(ex);
```
  - [5]: 

```
xcpp::display(expanded_ex);


$$8 + 24\sqrt{2}x + 40\sqrt{2}x^3 + 6\sqrt{2}x^5 + 60x^2 + 30x^4 + x^6$$

```
- Bottom Status Bar:** Simple, 0 \$ 1, C++23 | Idle, Mode: Command, Ln 1, Col 28, Untitled2.ipynb, 0, Bell icon.

# Array based Computing

The screenshot shows a Jupyter Notebook interface with the following details:

- File Bar:** File, Edit, View, Run, Kernel, Tabs, Settings, Help.
- Kernel:** C++23.
- Code Cell [2]:**

```
#include <xcpp/xdisplay.hpp>
#include "xtensor/containers/xarray.hpp"
#include "xtensor/io/xio.hpp"
#include "xtensor/views/xview.hpp"

[2]: xt::xarray<double> arr1 = {{1.0, 2.0, 3.0}, {2.0, 5.0, 7.0}, {2.0, 5.0, 7.0}};
      xt::xarray<double> arr2{5.0, 6.0, 7.0};

      xcpp::display(xt::view(arr1, 1) + arr2);
```

Output:

7.
11.
14.
- Code Cell [3]:**

```
xt::xarray<int> arr = {1, 2, 3, 4, 5, 6, 7, 8, 9};
arr.reshape({3, 3});

xcpp::display(arr);
```

Output:

1	2	3
4	5	6
7	8	9
- Status Bar:** Simple, Mode: Command, Ln 1, Col 1, Untitled.ipynb, 0.

# Array based Computing

The screenshot shows a Jupyter Notebook interface with the following details:

- File Menu:** File, Edit, View, Run, Kernel, Tabs, Settings, Help.
- Title Bar:** Untitled.ipynb, a tab icon, and a '+' button.
- Toolbar:** Includes icons for file operations like new, open, save, and a mode switch labeled "Code".
- Code Cells:**
  - [1]:

```
#include <xcpp/xdisplay.hpp>
#include <iostream>
#include "xtensor-blas/xlinalg.hpp"
```
  - [2]:

```
xt::xtensor<double, 2> m = {{1.5, 0.5}, {0.7, 1.0}};

std::cout << "Matrix rank: " << xt::linalg::matrix_rank(m) << std::endl;
std::cout << "Matrix inverse: " << std::endl;
xcpp::display(xt::linalg::inv(m));
std::cout << "Eigen values: " << std::endl;
xcpp::display(xt::linalg::eigvals(m));
```

Output:  
Matrix rank: 2  
Matrix inverse:  
0.869565 -0.434783  
-0.608696 1.304348

Eigen values:  
1.892262+0.i  
0.607738+0.i
- Bottom Status Bar:** Click to add a cell, Simple (radio button), 0 \$ 1, C++23 | Idle, Mode: Command, checked, Ln 3, Col 15, Untitled.ipynb, 0, and a bell icon.

# SIMD Acceleration

The screenshot shows a Jupyter Notebook interface with the following details:

- File Bar:** File, Edit, View, Run, Kernel, Tabs, Settings, Help.
- Title Bar:** Untitled2.ipynb, C++23 mode indicator.
- Toolbar:** Includes icons for file operations (+, -, X, etc.) and a gear icon for settings.
- Code Cells:**
  - [2]: 

```
#include <wasm_simd128.h>
#include <stdio.h>
```
  - [3]: 

```
v128_t v1 = wasm_f32x4_make(1.2f, 3.4f, 5.6f, 7.8f);
v128_t v2 = wasm_f32x4_make(2.1f, 4.3f, 6.5f, 8.7f);
```
  - [4]: 

```
v128_t v3 = wasm_f32x4_add(v1, v2);

printf("v3: %.1f, %.1f, %.1f, %.1f]\n",
    wasm_f32x4_extract_lane(v3, 0),
    wasm_f32x4_extract_lane(v3, 1),
    wasm_f32x4_extract_lane(v3, 2),
    wasm_f32x4_extract_lane(v3, 3));
```
- Output Cell [4]:**

```
v3: [3.3, 7.7, 12.1, 16.5]
```
- Bottom Status Bar:** Simple mode switch, cell count (0), C++23 | Idle, Mode: Command, Line 5, Column 35, Untitled2.ipynb, 0 errors, and a bell icon.

# SIMD Acceleration

The screenshot shows a Jupyter Notebook interface with the following details:

- File Bar:** File, Edit, View, Run, Kernel, Tabs, Settings, Help.
- Title Bar:** Untitled2.ipynb, C++23.
- Toolbar:** Includes icons for file operations like new, open, save, and run, along with tabs for Code, Text, and a gear icon for settings.
- Code Cells:**
  - [5]:

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

namespace xs = xsimd;
```
  - [6]:

```
// 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)
- Output Area:** Shows the result of cell [6] as (3.3, 7.7, 12.1, 16.5).
- Bottom Status Bar:** Simple, 0 \$ 1, C++23 | Idle, Mode: Command, Ln 9, Col 29, Untitled2.ipynb, 0, a bell icon.

# Interactive Widgets

The screenshot shows a Jupyter Notebook interface with the following details:

- File Menu:** File, Edit, View, Run, Kernel, Tabs, Settings, Help.
- Toolbar:** Includes icons for file operations like new, open, save, and a gear icon for settings.
- Left Sidebar:** Shows a file tree with a single file named "Untitled.ipynb" selected. It also includes a search bar labeled "Name" and a modified timestamp "now".
- Code Editor:** The notebook tab "Untitled.ipynb" is active. The code cell [1] contains the header "#include <xcpp/xdisplay.hpp>". Cells [3] and [4] are empty. Cell [5] contains the following C++ code:

```
5]: slider.value = 20;
slider.max = 40;
slider.style().handle_color = "blue";
slider.orientation = "vertical";
slider.description = "A slider";
```

Cell [6] contains the command "xcpp::display(slider);".
- Output Area:** Displays the output of cell [6]. It shows the text "A slider" above a vertical slider widget. The slider has a blue handle at the top and is labeled "31.05" below it.
- Bottom Status Bar:** Shows "Simple" mode, cell index "1", kernel status "C++23 | Idle", mode "Edit", line "Ln 1, Col 22", file "Untitled.ipynb", and a bell icon.

# Interactive Widgets

The screenshot shows a Jupyter Notebook interface with the following details:

- File Bar:** File, Edit, View, Run, Kernel, Tabs, Settings, Help.
- Toolbar:** Includes icons for file operations like new, open, save, and a gear icon for settings.
- Left Sidebar:** Shows a file tree with a single item: Untitled.ipynb (modified now).
- Code Editor:** Displays five code cells:
  - [1]: `#include <xcpp/xdisplay.hpp>`  
`#include <sstream>`
  - [2]: `#include "xcanvas/xcanvas.hpp"`
  - [3]: `auto canvas = xc::canvas().initialize()`  
`.width(300)`  
`.height(300)`  
`.finalize();`
  - [4]: `canvas.fill_rect(0, 0, 50, 50);`  
`canvas.fill_circle(100, 100, 50);`
  - [5]: `xcpp::display(canvas);`
- Output Area:** Shows the rendered output of the last cell, which consists of a black square located at (0, 0) and a black circle located at (100, 100), both with a radius of 50.
- Bottom Status Bar:** Shows "Simple" mode, kernel status (C++23 | Idle), mode (Command), line and column (Ln 1, Col 1), notebook name (Untitled.ipynb), and a bell icon.

# Magic Commands

The screenshot shows the Jupyter Notebook interface with a file tree on the left and a code editor on the right.

**File Tree:**

- Untitled1.ipynb (selected)
- tmp.txt

**Code Editor:**

```
[2]: %%file tmp.txt
Demo of magic command
Writing tmp.txt

[3]: %%file -a tmp.txt
append at the end
Appending to tmp.txt

[ ]:
std::ifstream infile("tmp.txt");
std::string line;
while (std::getline(infile, line)) {
    std::cout << line << std::endl;
}
infile.close();
```

# Magic Commands

The screenshot shows a Jupyter Notebook interface with the following details:

- File Bar:** File, Edit, View, Run, Kernel, Tabs, Settings, Help.
- Toolbar:** Includes icons for file operations like new, open, save, and a gear icon for settings.
- Title Bar:** Shows the notebook name "timeit.ipynb".
- Code Editor:** Displays the following code cells:

  - [1]:  
```c++  
#include <iostream>  
#include <chrono>  
#include <thread>  
#include <cmath>
  - [2]:  
```c++  
%timeit std::cout << 1 << std::endl;  
2.26 us +- 12.7 ns per loop (mean +- std. dev. of 7 runs 100000 loops each)
  - [\*]:  
```c++  
%%timeit -n 10 -r 1 -p 6  
std::this\_thread::sleep\_for(std::chrono::seconds(2));
  - [\*]:  
```c++  
exit -n 8 -r 1 -p 4 std::this\_thread::sleep\_for(std::chrono::seconds(3));
  - [ ]: Placeholder for a new cell, with the text "Click to add a cell."

- Status Bar:** Shows "Simple" mode, a progress bar at 0, "C++20 | Busy", "Mode: Edit", a shield icon, "Ln 1, Col 56", "timeit.ipynb", a "0" count, and a bell icon.

# Magic Commands

The screenshot shows a Jupyter Notebook interface with the title bar "Untitled18.ipynb". The notebook contains three code cells:

- Cell 1:** `%mamba install doctest`  
Output:  
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.
- Cell 2:**

```
#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();
}
```
- Cell 3:**

```
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

```
// 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

The screenshot shows a Jupyter Notebook interface with the following details:

- File Menu:** File, Edit, View, Run, Kernel, Tabs, Settings, Help.
- Toolbar:** Includes icons for file operations like new, open, save, and run, along with a kernel dropdown and a "Code" button.
- Code Cell:** Untitled.ipynb tab is active. The code attempts to load a shared library and call a function from it:

```
return 1;
}

func_t myFunc = (func_t) dlsym(handle, "my_function");
if (!myFunc) {
    std::cerr << "Function not found in main module" << std::endl;
} else {
    std::cout << "Function found! Calling my_function()" << std::endl;
    myFunc(); // If found, call it directly
}
return 0;
}

main();
```
- Output:** A red error bar indicates a runtime error: "Function not found in main module".
- File Explorer:** Shows the local directory contents:

| Name                | Modified |
|---------------------|----------|
| audio.wav           | 9h ago   |
| custom_module.so    | now      |
| marie.png           | 9h ago   |
| README.md           | 9h ago   |
| Untitled.ipynb      | 1m ago   |
| xeus-cpp-lite-de... | 9h ago   |

# Loading third-party/custom libs

The screenshot shows a Jupyter Notebook interface with the following details:

- File Bar:** File, Edit, View, Run, Kernel, Tabs, Settings, Help.
- Toolbar:** Includes icons for file operations like new, open, save, and a search/filter icon.
- Left Sidebar:** Shows a file tree with a folder containing "custom\_module.so" (modified 5d ago) and the current notebook "Untitled.ipynb" (modified now).
- Code Cell [1]:**

```
#include "clang/Interpreter/CppInterOp.h"
#include <iostream>
```
- Code Cell [2]:**

```
// Load the side module
Cpp::LoadLibrary("custom_module.so");

// Resolve the function symbol
void* Addr = Cpp::GetFunctionAddress("my_custom_function");

// Check if it was successfully found
if (!Addr) {
    std::cerr << "Symbol not found!\n";
} else {
    using FuncType = void (*)();
    auto Fn = reinterpret_cast<FuncType>(Addr);
    Fn(); // Call the function
}
```
- Output Cell:** Displays the text "Hello from my\_custom\_function!".
- Bottom Status Bar:** Shows "Simple" mode, cell count (0), kernel status (C++23 | Idle), mode (Command), line/col (Ln 1, Col 1), notebook name (Untitled.ipynb), and a bell icon.

# 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

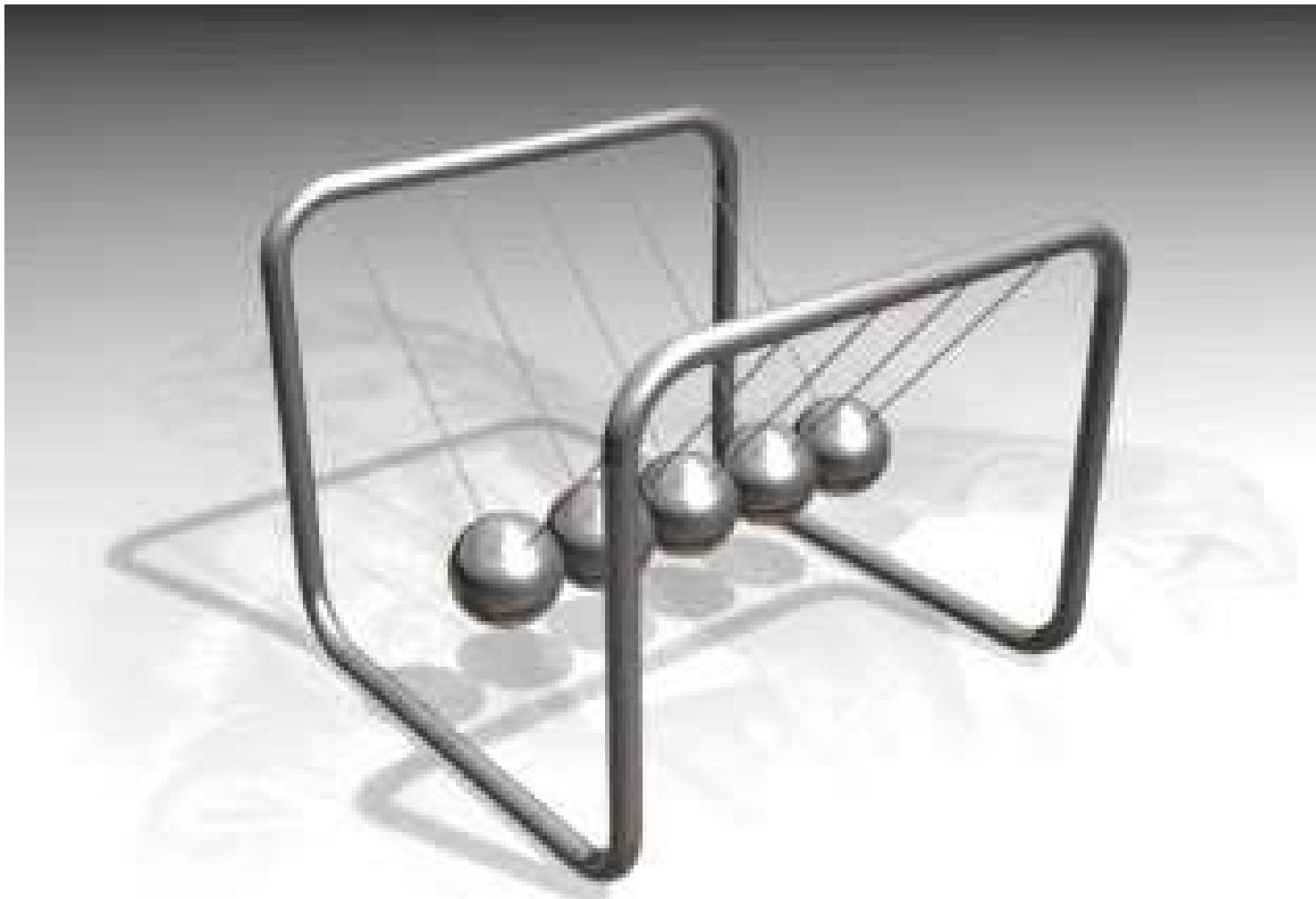
For example, to deploy a C++ kernel with Symengine & Xtensor-blas installed, the environment.yml file would contain the following:

```
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 Jupyterlite 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 [ ]:
```

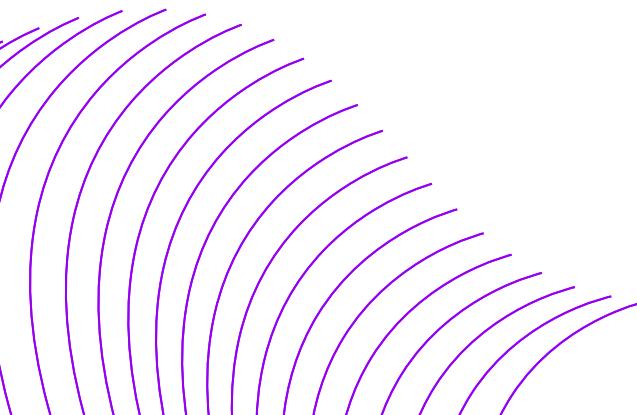
```
In [ ]:
```

```
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

The screenshot shows a Jupyter Notebook interface with a single open notebook titled "multi-lang.ipynb". The notebook contains three code cells:

- [1]:  
```c++  
#include <iostream>  
```
- [2]:  
```c++  
std::cout << "Hello, C++!" << std::endl;  
Hello, C++!  
```
- [3]:  
```python  
print("Hello, Python!")  
Hello, Python!  
```

The interface includes a file browser on the left, a toolbar at the top, and status indicators at the bottom.

# 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

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