

Ananya's Documentation Writing TODO: qkrylov

Welcome! This file is your complete brief. qkrylov is a polyglot (C++, Python, Julia) quantum many-body physics engine. Your job is to write the narrative documentation.

The Big Picture: Read This First

Why is the docs system unusual?

Normally, documentation sites like this are built from plain .md (Markdown) files. That is exactly what we are using: a tool called **Zensical**, which is a very fast static site generator (essentially MkDocs written in Rust).

However, when you are documenting a physics engine, you don't just want to *describe* things: you want to *demo* them. And the natural way to demo computational physics is a **Jupyter notebook** (.ipynb), because you can have narrative text, live code, and output all in one place.

On top of that, qkrylov is a **polyglot** library. It has Python, Julia, and C++ interfaces. Writing a tutorial that shows all three languages inside a single Jupyter notebook would be a nightmare: you cannot mix kernels, and the narrative flow gets tangled.

So Aritra built a small hack to make your life easier.

The Hack: build_polyglot.py

Instead of running Zensical directly, you run:

```
python build_polyglot.py serve
```

This script acts as a smart wrapper. It does the following: 1. It looks for any tutorial folders you've set up (more on that in a second). 2. For each one, it reads your main.ipynb notebook and generates a beautiful Markdown page with synchronized multi-language tabs and download buttons for all three languages. 3. It boots Zensical and serves the live docs site with hot-reload: **so every time you save a file, the browser refreshes automatically**. 4. When you kill the server (Ctrl+C), it deletes all the auto-generated files so your working directory stays completely clean. Although this might not work always, your working directory sometimes might remain dirty with generated files.

You will never need to touch build_polyglot.py itself. Just use it.

Two Modes of Writing Documentation

Mode 1: Plain Markdown (for reference pages, theory, API overviews)

If you are writing a page that does NOT need interactive demos or live code output, just write a regular `.md` file and list it in `zensical.toml`.

Example: `docs/theory/lanczos_memory.md` is a plain markdown file explaining the single-Pass vs Two-Pass memory trade-off. It has (not yet, you would add) math and code blocks, but they are not interactive: they are just illustrative snippets. This is totally fine.

In `zensical.toml`, you must list it in the `nav` block so it appears in the sidebar:

```
{ "Single-Pass vs Two-Pass Lanczos" = "theory/lanczos_memory.md" },
```

You create the file. You list it in `zensical.toml`. Done.

Mode 2: Polyglot Tutorial (for hands-on demos across all 3 languages)

If you are writing a tutorial that shows actual working code in Python, Julia, AND C++ see the Polyglot Tutorial format.

Instead of creating a `something.md` file, you create a folder called `something/`.

Inside that folder, you put three files:

```
docs/foo/bar/something/
├─ main.ipynb    # You write this. Python narrative + code.
├─ main.jl       # You write this. Same logic in Julia.
└─ main.cpp      # You write this. Same logic in C++.
```

Behind the scenes, Aritra will point the navigation to the auto-generated `index.md`:

```
{ "Your Tutorial Title" = "foo/bar/something/index.md" },
```

He will also register the folder in the `[extra]` section of `zensical.toml` so the build script knows to process it:

```
[extra]
polyglot_tutorials = [
  "docs/foo/bar/something",
]
```

You create the folder and the three files. You tell Aritra the folder name. He wires it up. Done.

(Note: If you want to see a live example of exactly how this looks, open the `docs/getting_started/tutorial_hamiltonian/` folder and look at how the three files are structured.)

How the Three Files Work Together

The build script stitches the three files into synchronized tabs. The key is that each code “step” has an **ID annotation** that must match across all three files.

In `main.ipynb` (Jupyter notebook, Python):

Each julia or c++ code cell that you want to appear in the docs gets a comment at the top and bottom (python don't coz we know the cell boundaries already):

```
# [ID: init_basis]
basis = SpinHalfBasis(10)
sector = SpinHalfBasis.sector(Sz=0)
```

In `main.jl` (Julia):

```
# [ID: init_basis]
b = SpinHalfBasis(10, Sector(0))
# [END: init_basis]
```

In `main.cpp` (C++):

```
// [ID: init_basis]
basis::SpinHalf b(10, sector::Sz(0));
// [END: init_basis]
```

The build script finds matching IDs and places them into synchronized language tabs in the output page.

Important: The ID names must be identical across all three files. If a step has no meaningful C++ or Julia equivalent, you can simply omit it from those files.

The narrative text (markdown prose, headings, explanations) lives in the **Markdown cells** of `main.ipynb`. The build script extracts that and uses it as the page content between code blocks.

Live Preview

To see your work as you write, from the `qkrylov/` root directory run:

```
python build_polyglot.py serve
```

Then open your browser at `http://localhost:8000`.

When you save any of the three source files, the browser will refresh automatically.

The Golden API References

Always use these files as ground truth for exact function names and syntax. Do not invent function names: copy them from here:

- Python: [docs/api/python.md](#)
- Julia: [docs/api/julia.md](#)
- C++: [docs/api/cpp.md](#)
- C ABI: [docs/api/c-abi.md](#)

Critical API note for Julia: SinglePass was recently renamed to OnePass. Always write `Lanczos(variation=OnePass())` or simply `Lanczos()` (defaults to OnePass).

High Priority: Pages Are in the Nav But Broken

These pages are already linked from the sidebar. Users can click them right now and see garbage content written by an AI that was using the wrong API. Fix these first.

1. docs/getting_started/ecosystem.md

Current state:  Wrong API. Wrong function names, wrong package imports. **What it should be:** A practical page showing how to use qkrylov output downstream.

This is a **plain markdown** page (Mode 1), no notebook needed.

Show each of these with a minimal working code snippet: - Convert eigenvector to a numpy array - Export the Hamiltonian as a `scipy.sparse.csr_matrix` - Wrap the Hamiltonian as a `scipy.sparse.linalg.LinearOperator` for use with `eigsh` - Export an eigenvector as a `torch.Tensor` for downstream ML tasks

Use the 1D Heisenberg model ($L=10$, $S_z=0$ sector) as the example throughout. Reference: [Python API docs](#): look for the ecosystem and LinearOperator sections.

2. docs/getting_started/sciml_julia.md

Current state:  Wrong API. Uses `SpinBasis`, `EigenProblem`: these do not exist. **What it should be:** A Julia-focused page showing the SciML `solve(prob, alg)` pattern.

This is also a **plain markdown** page (Mode 1). Cover: - `GroundStateProblem + solve(prob, Lanczos())` - `ExcitedStatesProblem + solve(prob, Davidson(n_eig=3))` - `ThermalProblem + solve(prob, FTLM())` for finite temperature observables - The decoupled workflow: `ftlm_sample + ftlm_evaluate_sweep` (fast multi-temperature sweeps)

Reference: [Julia API docs](#)

3. docs/api/index.md

Current state:  Stub. Just says “Under Construction.” **What it should be:** A short (~200 word) overview explaining the four API layers (pure C++, C ABI, Python, Julia) and when to use each one. Finish with a table linking to the four dedicated API reference pages. This is a quick win: good first task if you want to warm up.

● Medium Priority: Pages Are Written but Need Vetting

These pages were written by AI and contain plausible-looking physics prose, but have NOT been reviewed by a human physicist. Check the code and physics against the API references and fix anything wrong.

4. docs/theory/matrix_free_spmv.md

Explains why we never build the full Hamiltonian matrix. Check the math (2^L memory growth for spin systems) and the Python code examples.

5. docs/theory/lanczos_memory.md

Explains Single-Pass (high memory, stores full Krylov subspace $O(k \times N)$) vs Two-Pass (recomputes Krylov vectors on the fly, $O(N)$ memory, much slower per-run but GPU-friendly). Verify the memory formulas are correct.

6. docs/theory/precision_stability.md

Explains FP32 vs FP64 in Lanczos: ghost states, loss of orthogonality, convergence speed. **Key physics to verify:** FP32 is faster per iteration but pollutes the Krylov subspace (“ghost states”), requiring more iterations. FP64 is slower per iteration but maintains clean orthogonality, so it needs fewer iterations. With early stopping, FP64 often wins in wall-clock time.

7. docs/replications.md

Contains paper replications (Bethe Ansatz for 1D Heisenberg, etc.). Check that the Python code examples use the correct `qkrylov` API function names. Fix any wrong function calls.

● New Tutorials to Write (Not Started)

These do not exist yet. For each one, create the folder and files, then wire the folder into `zensical.toml`'s navigation and `[extra]` blocks as explained in Mode 2.

All of these use **Mode 2 (Polyglot Tutorial)**: you create `main.ipynb`, `main.jl`, and `main.cpp`.

8. Tutorial: FTLM (Finite Temperature Lanczos Method)

Create: `docs/getting_started/tutorial_ftlm/`

Storyline: 1. Build the Heisenberg Hamiltonian briefly (you can reference the Hamiltonian tutorial). 2. Use `FTLM.sample()` to generate R random Krylov samples (the heavy step: SpMV). 3. Use `FTLM.evaluate_sweep(betas)` with a temperature grid to

compute observables. 4. Plot $E(\beta)$, $C_v(\beta)$, $S(\beta)$ vs temperature. 5. Show the **decoupled workflow advantage**: generate samples once, then call `evaluate_sweep` for many different temperature grids instantly at zero SpMV cost.

Reference: `Python FTLM.sample`, `FTLM.evaluate_sweep`, `FTLMSamples` in [Python API](#).

9. Tutorial: Quantum Dynamics & Spectral Functions

Create: `docs/getting_started/tutorial_dynamics/`

Storyline: 1. Build Hamiltonian, get ground state. 2. Evolve in time using `time_evolve`: compute survival probability $|\langle \psi_0 | \psi(t) \rangle|^2$. 3. Compute a dynamical spin structure factor $S(q, \omega)$ using `dynamical_correlator`. 4. Show a simple spectral function plot.

Reference: Dynamics section of [Python API](#).

10. Tutorial: Excited States (Davidson Solver)

Create: `docs/getting_started/tutorial_excited_states/`

Storyline: 1. Build Hamiltonian. 2. Use `Davidson(n_eig=5)` to get the 5 lowest energy eigenstates. 3. Explain why Davidson is better than running Lanczos 5 separate times for multiple eigenstates. 4. Show an energy level spectrum plot.

Writing Style Guide

- **Use the Heisenberg model everywhere.** It is the canonical example: $L=10$ sites, $J=1.0$, $S_z=0$ sector.
- **Never invent function names.** Copy them exactly from the API reference files.
- **Write prose, not bullets.** The tutorials should read like a lab notebook: explain *why*, not just *how*.
- **Math is encouraged.** Use LaTeX: inline $\dots$ and display $\dots$ freely. Zensical renders it beautifully.
- **Tabs for polyglot.** For short snippets that have Python, Julia, and C++ versions, use the tab syntax directly in your markdown:

```
=== "Python"
    ``python
    # your code here
    ```

=== "Julia"
 ``julia
 # your code here
    ```

=== "C++"
    ``cpp
```

```
// your code here  
``
```

- **Do NOT edit files in docs/api/:** those are Pritipriya's authoritative API references. Leave them alone.
- **Do NOT create index.md inside a tutorial folder:** the build system generates this automatically. If you create one manually, it will be overwritten and you will be confused.