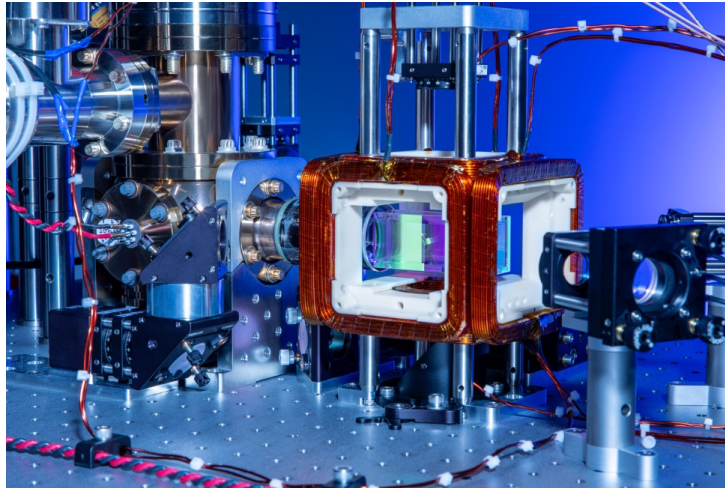


HIGH PERFORMANCE CAPABILITIES

Recap

- Focused on **Bloqade-to-Hardware Pipeline**

$$\frac{H}{\hbar} = \sum_i \frac{\Omega(t)}{2} (e^{i\phi(t)} |g_i\rangle\langle r_i| + e^{-i\phi(t)} |r_i\rangle\langle g_i|) - \sum_i \Delta_i(t) n_i + \sum_{i<j} V_{ij} n_i n_j$$

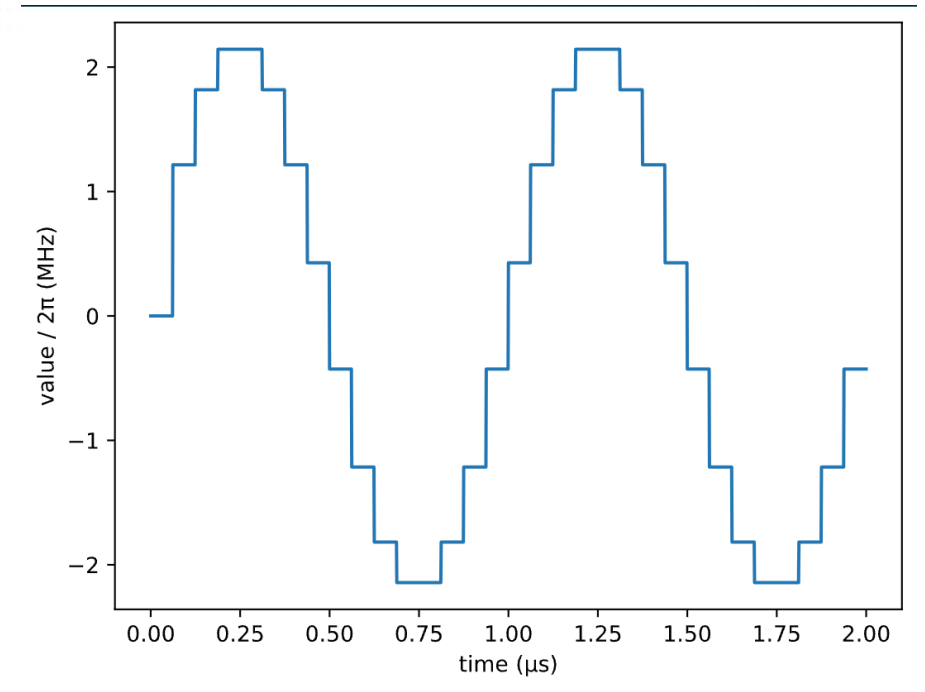
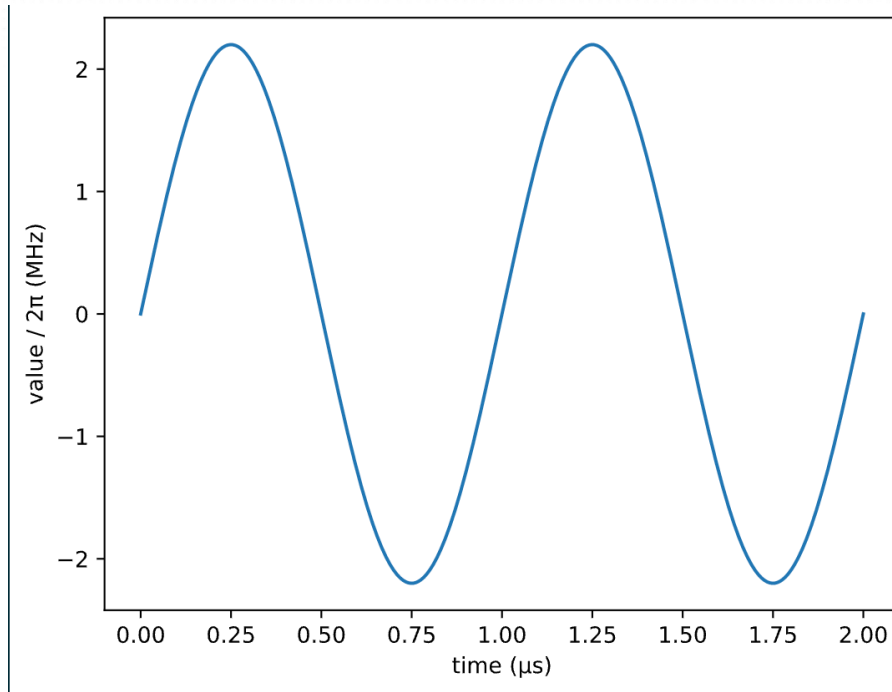


Recap (Cont.)

- 1. **Define** your problem
 - If you can, perform emulation!
- 2. **Transform** to Hardware
- 3. **Validate**
- 4. **Tweak** problem (if necessary)
- 5. **Submit!**
- 6. **Retrieve** data

Recap (Cont.)

- Also learned about various Hardware Constraints and how to work around them



Learning Objectives

- **Explain** how Bloqade's tools for high performance work
- **Execute** dynamics simulations using Blockade Subspace, Multithreading, and GPU support
- **Critique** the benefits and drawbacks of each method

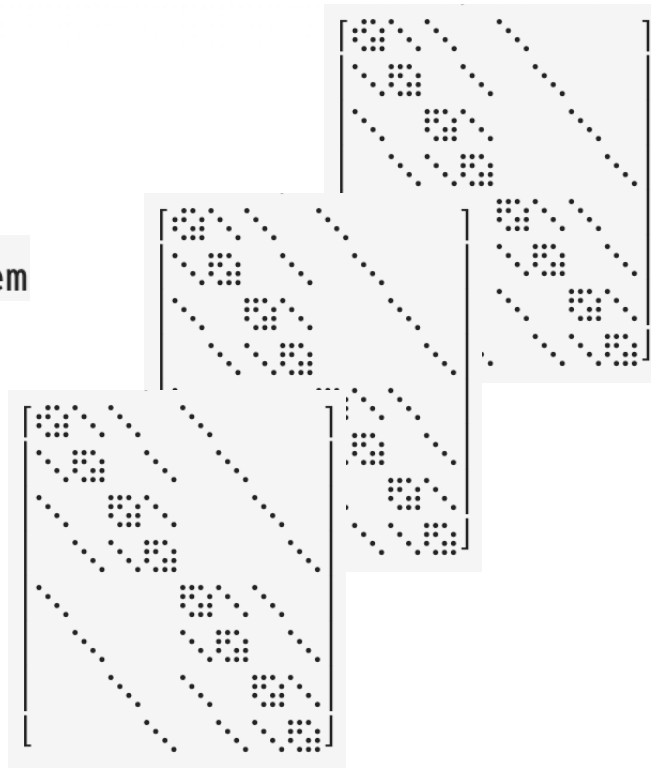
Bird's Eye View

- Let's look under the hood



$$i\hbar \frac{\partial}{\partial t} |\psi\rangle = \hat{\mathcal{H}}(t) |\psi\rangle$$

SchrodingerProblem



$$y_{n+1} = y_n + h \sum_{i=1}^s b_i k_i$$

$$k_1 = f(t_n, y_n),$$

$$k_2 = f(t_n + c_2 h, y_n + h(a_{21} k_1)),$$

$$k_3 = f(t_n + c_3 h, y_n + h(a_{31} k_1 + a_{32} k_2)),$$

\vdots

$$k_i = f\left(t_n + c_i h, y_n + h \sum_{j=1}^{i-1} a_{ij} k_j\right).$$

Can We Optimize This?

- *Memory Efficiency*
 - **Blockade Subspace**
- *Execution Efficiency*
 - **Multithreading**
 - **GPU support**

Blockade Subspace

- Key Idea: entire 2^n state space unnecessary
- If two atoms blockade each other, probability of reaching double Rydberg Excitation state impossible
- Can get rid of these states but should take care in potentially dropping non-trivial contributions to dynamics

Multiple Dispatch for Performance

- Targeting repeated solver calls (lots of Matrix-Vector multiplication)
- Reduce memory footprint as much as possible with "sparsest" possible matrix formats
- Take advantage of *Multiple Dispatch* to **define Matrix-Vector multiplication optimized for different matrix types!**
- Also lets us perform **Matrix-Vector multiplication with different backends** (Custom CPU approaches and simple GPU support)

```
= PermMatrix([2, 1, 3], T[1, 1, 0])  
= PermMatrix([1, 3, 2], T[0, 1, 1])  
  
= Diagonal(T[1, -1, 0])  
= Diagonal(T[0, 1, -1])  
  
= sparse([2], [2], T[1], 3, 3)  
= sparse([3], [3], T[1], 3, 3)
```

Multithreading

- Handled by **BloqadeExpr**
- Two different backends to be aware of
 - **ThreadedSparseCSR** – Naïve Sparse-Matrix Vector multiplication, good for balanced matrices
 - Little to no overhead (other than threads) but could have longer run time
 - **ParallelMergeCSR** – Based on Merrill and Garland's algorithm (10.1109/SC.2016.57), good for unbalanced matrices
 - Pay some upfront cost but ensure optimally balanced load across threads
- Balance – number of nonzero entries per row of Matrix
 - Can be unbalanced if you add phase or use subspace
 - Documentation can be found here: <https://queracomputing.github.io/Bloqade.jl/dev/multithreading/>

GPU Support

- Take advantage of **CUDA.jl**
- Additional step of loading and unloading from the GPU
- Keep in mind:
 - Now restricted to GPU memory
 - Loading and Unloading incurs some overhead

Conclusion

- Recap Learning Objectives:
 - **Understand** how Bloqade's tools for high performance work
 - **Compute** dynamics using Blockade Subspace, Multithreading, and GPU support
 - **Observe** the benefits and drawbacks of each method