Hands-on advanced numerical methods for quantum many-body dynamics

Université de Strasbourg

cnrs dépasser les frontières

CESQ CENTRE EUROPÉEN DE SCIENCES QUANTIQUES

www.cqmt.org

www.cesq.eu

Johannes Schachenmayer: **schachenmayer@unistra.fr**

Computational quantum many-body theory group

PhDs:
Ruben Daraban
Réka Schwengelbeck
Maxence Pandini
Sakshi Velvenkar
Devashish Tiwari
Sinara Dourado (visiting from UFSCar)

Post-docs:
Raphaël Menu
Yiwen Han
Tatiana Bespalova

Former:
Pedro Rosario (visiting PhD UFSCar)
Luiz Solak (visiting PhD UFSCar)
Guillermo Preisser (PhD)
David Wellnitz (PhD)
Thomas Botzung (Post-doc)
Stefan Schütz (Post-doc)

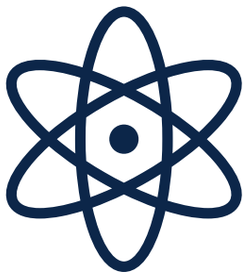
www.cqmt.org

Johannes Schachenmayer: **schachenmayer@unistra.fr**

European Research Council
Established by the European Commission

Motivation - Exploring the macroscopic quantum world numerically

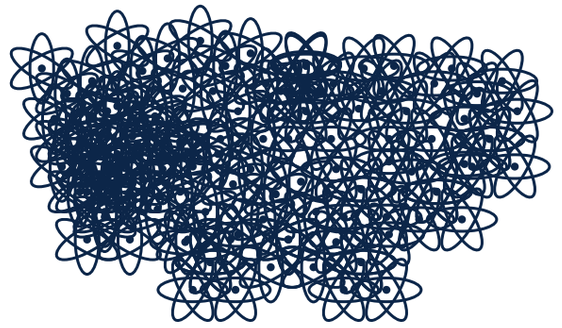
Microscopic world



Quantum physics

$$\frac{d}{dt}|\psi\rangle = -\frac{\mathrm{i}}{\hbar}\hat{H}|\psi\rangle$$

Macroscopic world

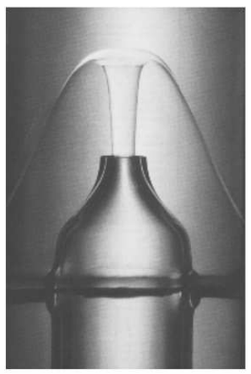


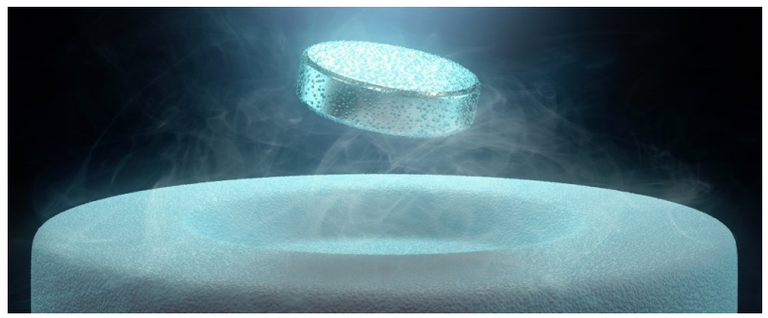
Classical physics

$$\boldsymbol{F} = m\boldsymbol{a}$$

(decoherence)

Macroscopic quantum effects?





very rare, low temperatures

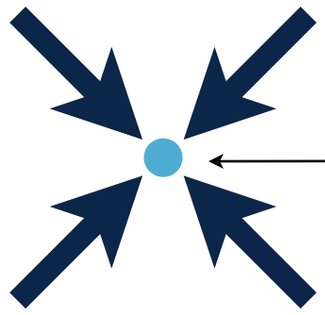
Motivation - Macroscopic quantum world

Gigantic Hilbert space $\dim(\mathcal{H}) \sim \exp(N)$ (N particles)

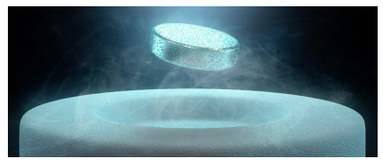
$\mathcal{H}$

Reasons:

- Noise
- Disorder
- Interaction range



Physics we know
(needs very few states)



What else is out there?

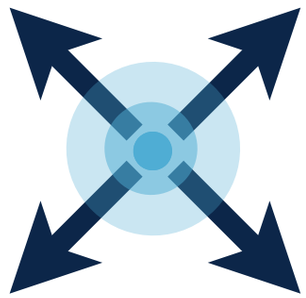
Motivation - Macroscopic quantum world

Gigantic Hilbert space $\dim(\mathcal{H}) \sim \exp(N)$ (N particles)

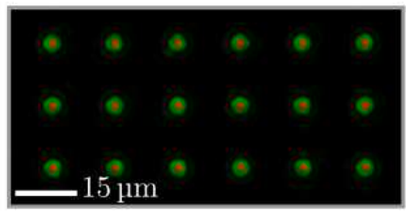
$\mathcal{H}$

Quantum Computers/Simulators fight:

- Noise
- Disorder
- Interaction range



Coherent control:



- **Fundamental question:** Does quantum physics hold on the large scale?
- **Applications:** Engineering material properties, enhanced sensing, computing … ?

Classical simulations of the macroscopic quantum world

$$\frac{d}{dt}|\psi\rangle = -\mathrm{i}\hat{H}|\psi\rangle$$

$$\frac{d}{dt}\hat{\rho} = -\mathrm{i}[\hat{H},\hat{\rho}] + \sum_i \mathcal{L}^{[i]}\hat{\rho}$$

(time-evolution & steady states)

$\hbar \equiv 1$

Motivation:

- Model experiments & benchmark the status of quantum platforms (quantum advantage?)
- Find new emergent macroscopic phenomena
- Fundamentally understand quantum many-body physics from a classical complexity perspective

Hands-on advanced numerical methods for quantum many-body dynamics

A tour through numerical methods for simulating large-scale quantum physics (classically)
Motto: Do it from scratch to better understand quantum physics and classical limitations
Structure: Theory lecture part + tutorial-style

- Language used: Julia, **https://julialang.org/** (open source, easy, fast linear algebra)

- **Outline** (may vary)

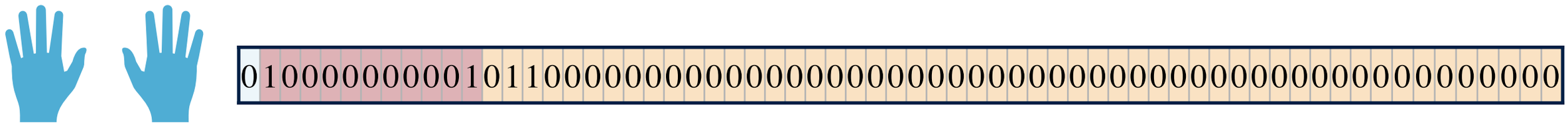
1. **20/10:** Basic concepts: Numbers on computers, basic exact diagonalization (ED)
   Tutorial: ED on a simple spin model
2. **21/10:** A better ED, sparse matrices, Krylov space. Open systems.
   Tutorial: Spin-model simulations using Krylov space
3. **22/10**: Mean-field, Runge-Kutta
   Tutorial: A mean-field simulation of the transverse Ising model
4. **23/10:** How to go beyond: Matrix product states

$$h \equiv 1$$

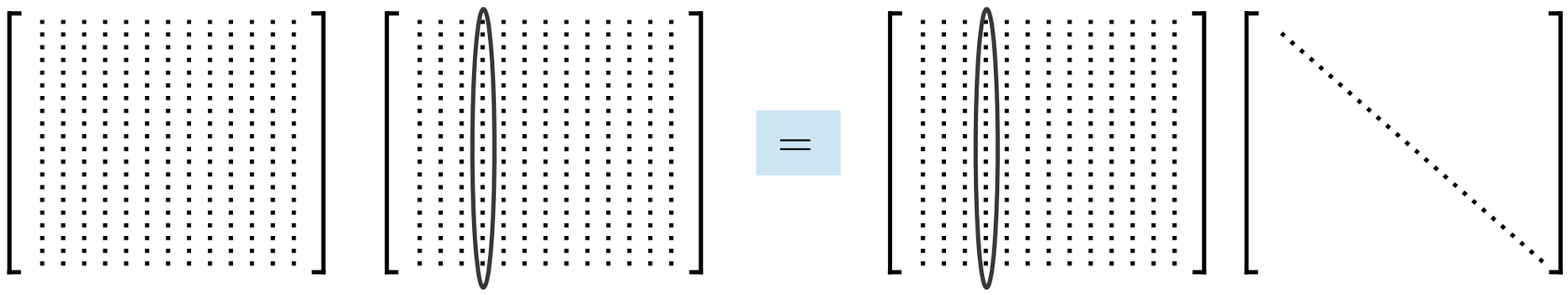
… always!

Plan for today

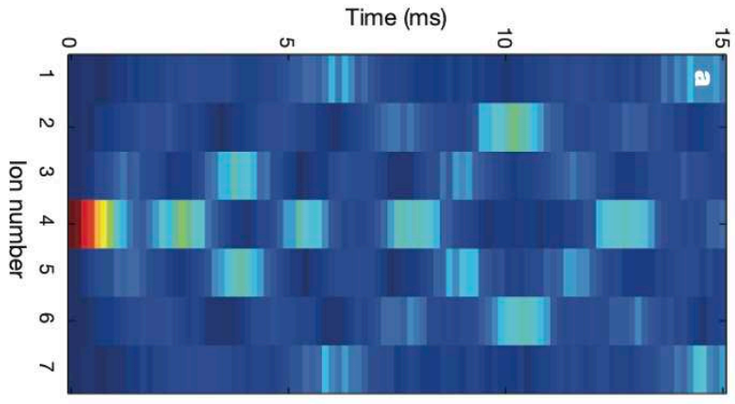
- **Part 1:** Brief summary about numbers in digital memory and the linear algebra of quantum mechanics



- **Part 2:** Exact diagonalization



- **Tutorial:** ED on a simple spin-model

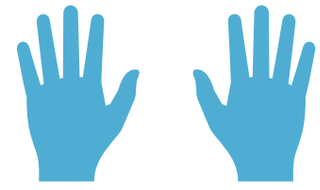


$$\cdots \otimes \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \otimes \begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix} \otimes \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \otimes \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \otimes \begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix} \otimes \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \otimes \cdots$$

spin #i *spin #j*

Integers on a computer

Latin: digitus = finger



- Computers are digital, they work with bit (true/false) representations of numbers
- Integers (signed, example with 4 bits)

$m=+0$	$m=+1$	$m=+2$	$m=+3$	$m=+4$	$m=+5$	$m=+6$	$m=+7$
0 0 0 0	0 0 0 1	0 0 1 0	0 0 1 1	0 1 0 0	0 1 0 1	0 1 1 0	0 1 1 1

Sign bit *Bit vector:* d_i $d_2\,d_1\,d_0$

$$m = d_2 2^2 + d_1 2^1 + d_0 2^0 = \sum_{i=0}^{2} d_i 2^i$$

$m=-0$	$m=-1$	$m=-2$	$m=-3$	$m=-4$	$m=-5$	$m=-6$	$m=-7$
1 0 0 0	1 0 0 1	1 0 1 0	1 0 1 1	1 1 0 0	1 1 0 1	1 1 1 0	1 1 1 1

- To not store the double zeros, usual convention is:

$m=-8$	$m=-7$	$m=-6$	$m=-5$	$m=-4$	$m=-3$	$m=-2$	$m=-1$
1 0 0 0	1 0 0 1	1 0 1 0	1 0 1 1	1 1 0 0	1 1 0 1	1 1 1 0	1 1 1 1

Representable range of numbers (n bits): $(-2^{(n-1)}), \ldots, (2^{(n-1)} - 1) \sim -10^{18}, \ldots, 10^{18}$ *(n=64)*

```
julia> m = 1
```

```
julia> typeof(m)
Int64
```

```
julia> m = 2^63-1
9223372036854775807
```

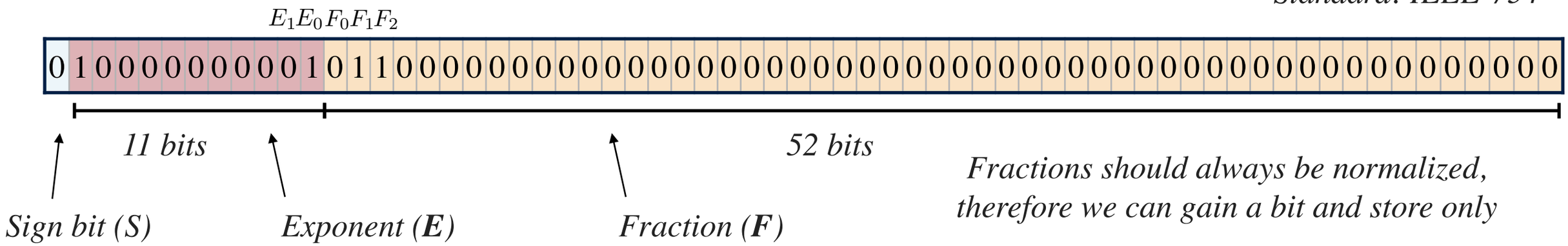
```
julia> m + 1
-9223372036854775808
```

- **Tip:** When really pushing code performance it can sometimes help to use bitwise operations

Floating point numbers on a computer

- Floating point numbers: $\pm 1.\text{fraction} \times 10^{\text{exponent}}$ … using **64 bits** (“double precision”)

Standard: IEEE-754



Fractions should always be normalized, therefore we can gain a bit and store only

$1.F$

- Then, in binary represented numbers are

$$\textit{fraction:} \quad f = 1 + F_0\ 2^{-1} + F_1\ 2^{-2} + F_2\ 2^{-3} + \cdots = 1 + \sum_{i=0}^{51} F_i\ 2^{-i-1}$$

$$\textit{exponent convention:} \quad \mathcal{E} = E - 1023 = \left(\sum_{i=1}^{11} E_i\ 2^i \right) - 1023$$

$$r = (-1)^S \times f \times 2^{\mathcal{E}}$$

- **Example:** $\mathcal{E} = 2^{10} + 2^0 - (2^{10} - 1) = 2 \qquad \mathcal{F} = 2^0 + 2^{-2} + 2^{-3} = 1.375$

(above)

$$r = (-1)^0 \times 1.375 \times 2^2 = +5.5$$

- **Important:** Relative precision $1.0 \approx 1.0 \pm 2^{-53} \approx 1.0 \pm 10^{-16}$

- This means in practice: **Keep units normalized and consider everything below 1e-16 zero** $\hbar \equiv 1$ *definitely!*

In practice, pick a time unit by normalizing an energy

Linear algebra of quantum mechanics

- Hilbert space of dimension D

State = Vector

$$\begin{bmatrix} \vdots \\ \psi_i \\ \vdots \end{bmatrix} \quad D \times 1$$

In general: Complex elements

$$D \times (2 \times 64)\,\text{Bits} = D \times 16\,\text{Bytes}$$

Operators = Matrix

$$\begin{bmatrix} \vdots & & \vdots \\ \cdots & h_{i,j} & \cdots \\ \vdots & & \vdots \end{bmatrix} \quad D \times D$$

Hamiltonian, Gate, Observables, Time-evolution operator

- Let's define a state-vector as a general concept (not limited to linear quantum mechanics)

$$\mathbf{y}(t) \begin{bmatrix} \vdots \end{bmatrix}$$

Ordinary differential equation

$$\dot{\mathbf{y}}(t) = f(t, \mathbf{y}(t))$$

Schrödinger equation

$$\frac{d}{dt}|\psi\rangle = -\mathrm{i}\hat{H}|\psi\rangle$$

$$\mathbf{y} = |\psi\rangle \qquad f(t, \mathbf{y}(t)) = \hat{H}\,|\psi\rangle \qquad \textit{(linear)}$$

- The state-vector can be anything:

Linearized density matrix

$$\mathbf{y} = [\rho_{1,1}, \rho_{1,2}, \rho_{2,1}, \rho_{2,2}]^T$$

Two classical particles

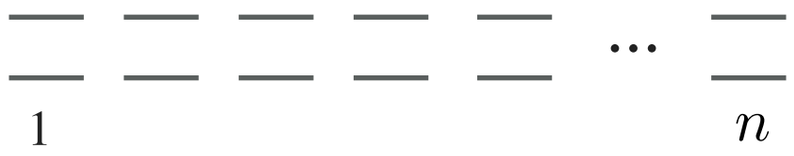
$$\mathbf{y} = [x_1, p_1, x_2, p_2]^T$$

$\ldots$

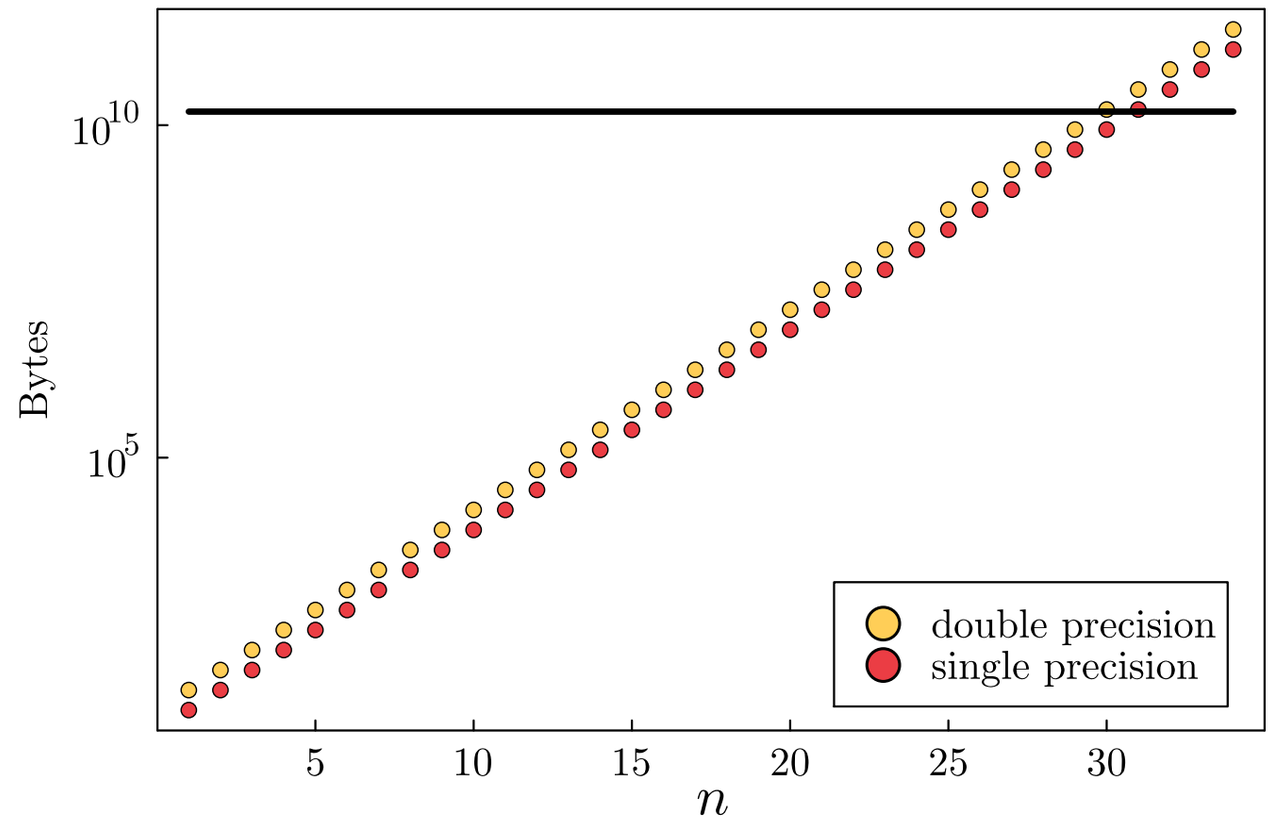
Linear algebra of quantum mechanics

- On a classical computer… how far can we go? *Double precision* $D \times (2 \times 64)\,\text{Bits} = D \times 16\,\text{Bytes}$
- Ultimately this will mean a memory limitation *Single precision* $D \times 8\,\text{Bytes}$

System of spins/qubits



$D = 2^n$



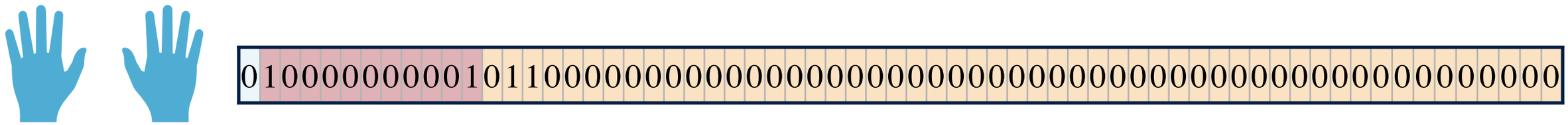
$n \approx 30$

doable on current hardware

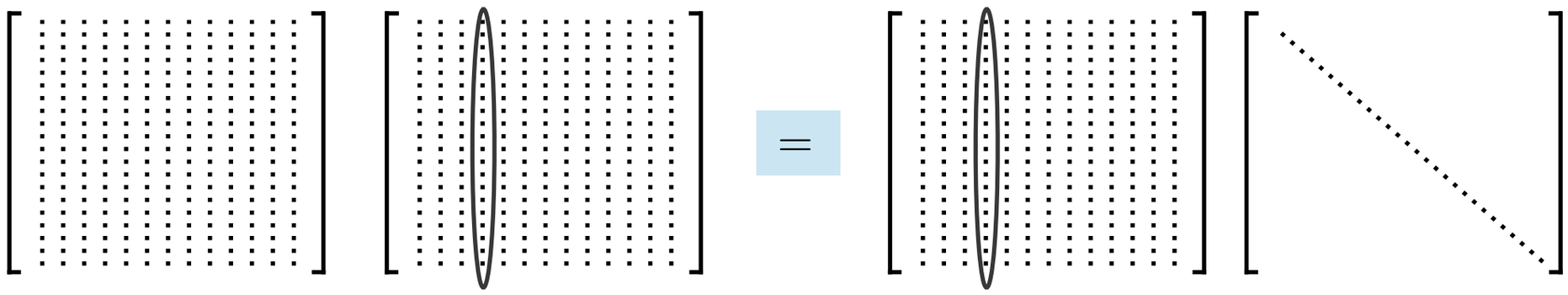
- **Remark:** Of course, a generic Hamiltonian would need memory $\sim D^2$, but we never need to store $\sim D^2$ elements for typical few-body Hamiltonians… when e.g. using Krylov space methods (see below)
- **Remark:** This is only the limit of exact state vector simulations, in practice with advanced methods, much larger systems are easily simulable (e.g. matrix product states, later)

Plan for today

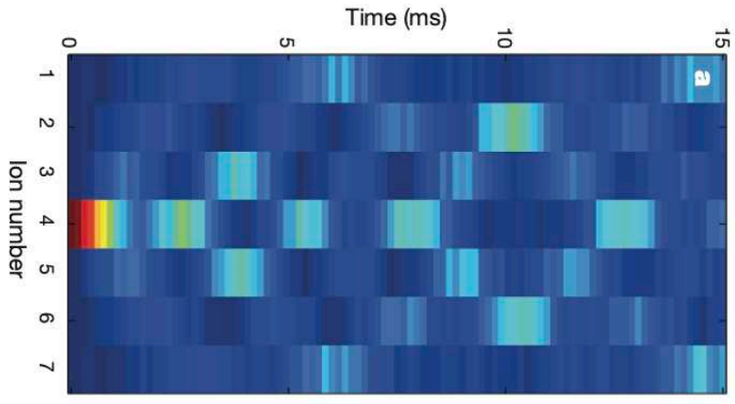
- **Part 1:** Brief summary about numbers in digital memory and the linear algebra of quantum mechanics



- **Part 2:** Exact diagonalization



- **Tutorial:** ED on a simple spin-model



$$\cdots \otimes \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \otimes \begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix} \otimes \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \otimes \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \otimes \begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix} \otimes \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \otimes \cdots$$

spin #i *spin #j*

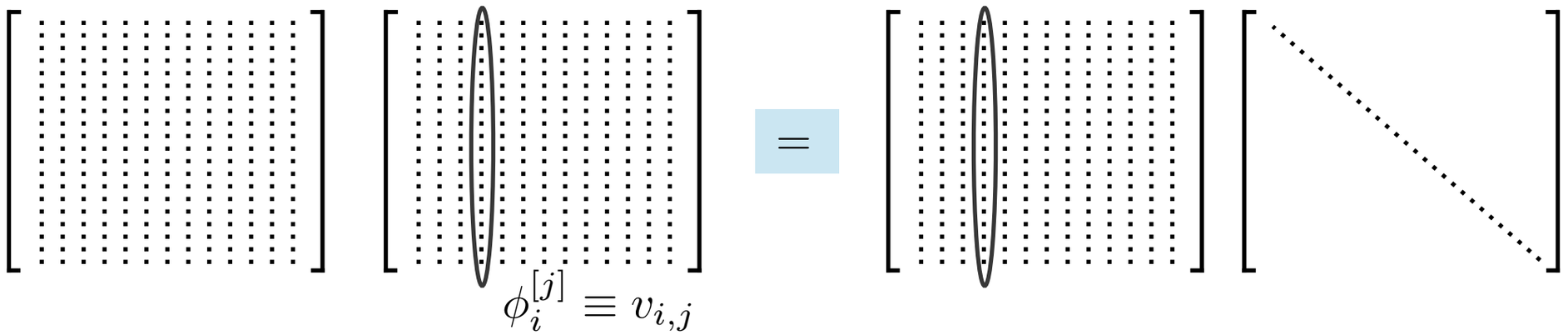
Exact Diagonalization

- Exact simulations of quantum mechanics often loosely described as: **Exact Diagonalization (ED)**

- **Summary:** *Hamiltonian is Hermitian:* $\hat{H}^\dagger = \hat{H}$ $D \times D$ *matrix* *…has real eigenvalues*

There exists a unitary matrix: $\hat{V}^\dagger \hat{V} = \hat{V}\hat{V}^\dagger = \mathbb{1}$, $\hat{V}^\dagger = \hat{V}^{-1}$ *s.t.* $\hat{V}^\dagger \hat{H} \hat{V} = \hat{E} \Leftrightarrow \hat{H}\hat{V} = \hat{V}\hat{E} \Leftrightarrow \hat{H} = \hat{V}\hat{E}\hat{V}^\dagger$

$(\hat{H})_{i,j} \equiv h_{i,j}$ $(\hat{V})_{i,j} \equiv v_{i,j}$ $(\hat{V})_{i,j} \equiv v_{i,j}$ $(\hat{E})_{i,j} \equiv e_{i,j}$



$\phi_i^{[j]} \equiv v_{i,j}$

Diagonal matrix

$e_{i,j} \equiv \delta_{i,j} E_j$

- The j-th column of the matrix *v* is the eigenvector corresponding to the j-th eigenvalue:

$$\sum_k h_{i,k} v_{k,j} = \sum_k v_{i,k} e_{k,j} = E_j v_{i,j} \qquad \phi_i^{[j]} \equiv v_{i,j} \qquad \sum_k h_{i,k} \phi_k^{[j]} = E_j \phi_i^{[j]}$$

- The columns of V are the "eigenkets" $(|\phi_j\rangle)_i \equiv v_{i,j}$ $\hat{H}\,|\phi_j\rangle = E_j\,|\phi_j\rangle$

- … the rows are the "eigenbras" (complex conjugated eigenvectors), which follows from $\hat{V}^\dagger \hat{H} = \hat{E}\hat{V}^\dagger$

Exact Diagonalization

- Exact simulations of quantum mechanics often loosely described as: **Exact Diagonalization (ED)**
- **Summary:** *Hamiltonian is Hermitian:* $\hat{H}^\dagger = \hat{H}$ $D \times D$ *matrix* *…has real eigenvalues*

There exists a unitary matrix: $\hat{V}^\dagger \hat{V} = \hat{V}\hat{V}^\dagger = \mathbb{1}$, $\hat{V}^\dagger = \hat{V}^{-1}$ *s.t.* $\hat{V}^\dagger \hat{H} \hat{V} = \hat{E} \Leftrightarrow \hat{H}\hat{V} = \hat{V}\hat{E} \Leftrightarrow \hat{H} = \hat{V}\hat{E}\hat{V}^\dagger$

- A diagonalization gives us a full spectral decomposition: $\hat{H} = \sum_k E_k \left|\phi_k\right\rangle \left\langle\phi_k\right|$

- With this we can solve Schrödinger equation time-evolution

$$\frac{d}{dt}|\psi\rangle = -\mathrm{i}\hat{H}|\psi\rangle \qquad |\psi(t)\rangle = \mathrm{e}^{-\mathrm{i}t\hat{H}} |\psi(t=0)\rangle$$

$$\hat{V}^\dagger\hat{V} = \mathbb{1} \Leftrightarrow \langle\phi_k|\phi_l\rangle = \delta_{k,l}$$

$$\left(|\phi_k\rangle\langle\phi_k|\right)^n = |\phi_k\rangle\langle\phi_k|$$

Time-evolution operator (matrix exponential)

$$\mathrm{e}^{-\mathrm{i}\hat{H}t} = \sum_{n=0} \frac{(-\mathrm{i}t\hat{H})^n}{n!} = \sum_{n=0} \frac{(-\mathrm{i}t\sum_k E_k |\phi_k\rangle\langle\phi_k|)^n}{n!} = \sum_k \mathrm{e}^{-\mathrm{i}tE_k} |\phi_k\rangle\langle\phi_k|$$

- Diagonalization allows to compute the matrix exponential, and to compute exact evolution (no time-stepping needed)

$$|\psi(t)\rangle = \sum_k \mathrm{e}^{-\mathrm{i}tE_k} |\phi_k\rangle\langle\phi_k|\psi(t=0)\rangle$$

Exact Diagonalization

- Create random Hamiltonian

```
julia> H = rand(ComplexF64, 100, 100); H += H'
```

$$\hat{V}^\dagger \hat{H} \hat{V} = \hat{E} \quad \Leftrightarrow \quad \hat{H}\hat{V} = \hat{V}\hat{E} \quad \Leftrightarrow \quad \hat{H} = \hat{V}\hat{E}\hat{V}^\dagger$$

- Compute V and E

```
julia> using LinearAlgebra

julia> E, V = eigen(H);
```

E comes out as vector

- Check equations:

```
julia> norm(H*V .- V*Diagonal(E))
2.3421977872625636e-13
```

```
julia> norm(H .- V*Diagonal(E)*V')
7.79771749285727e-13
```

Precision of diagonalization algorithm (not quite double, but enough)

- Time evolution with matrix exponential

$$|\psi(t)\rangle = \sum_k \mathrm{e}^{-\mathrm{i}tE_k} |E_k\rangle \langle E_k|\psi(t=0)\rangle$$

```
psi0 = zeros(ComplexF64, D)
psi0[1] = 1.0
psit = zeros(ComplexF64, D)
for kk = 1:D
    ovl = V[:, kk]' * psi0
    psit += exp(-1im * t * E[kk]) * ovl .* V[:, kk]
end
```

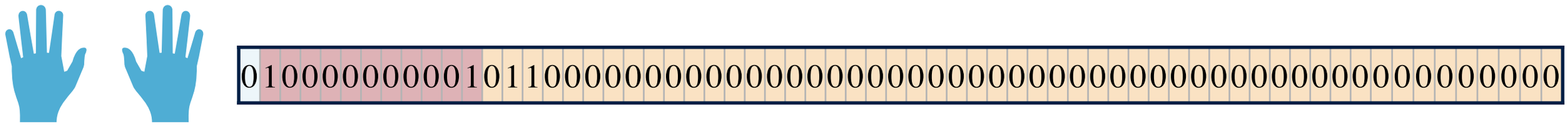
- Also possible, build matrix exponential directly with built-in routines:

```
julia> U = exp(-1im .* t .* H); psit = U*psi0
100-element Vector{ComplexF64}:
```

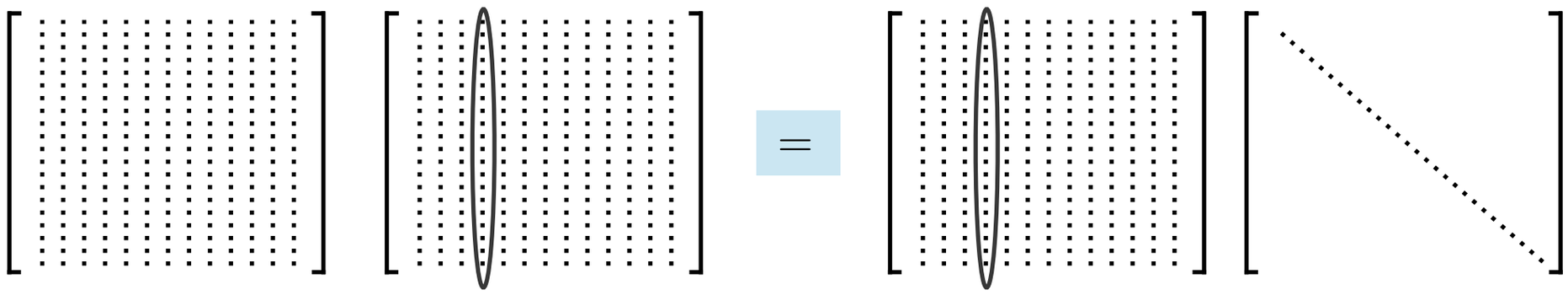
Warning: Make sure exp is not element-wise exponential!

Plan for today

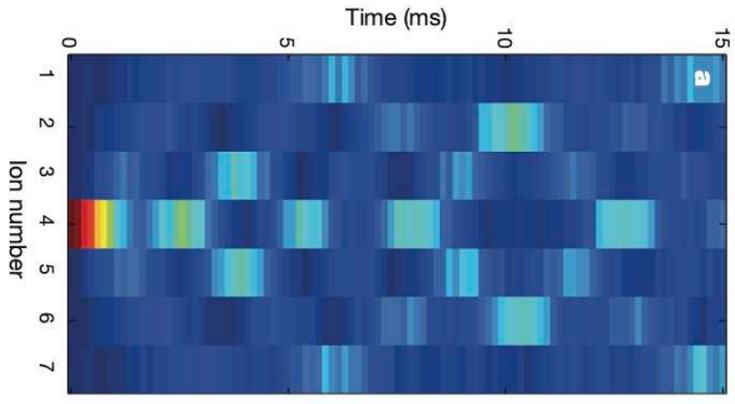
- **Part 1:** Brief summary about numbers in digital memory and the linear algebra of quantum mechanics



- **Part 2:** Exact diagonalization



- **Tutorial:** ED on a simple spin-model



$$\cdots \otimes \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \otimes \begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix} \otimes \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \otimes \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \otimes \begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix} \otimes \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \otimes \cdots$$

spin #i *spin #j*

Tutorial: Motivation — Analog spin-model quantum simulation

- Experiments implementing long-range spin-models effectively (e.g. with trapped ions)

$$J_{ij} = \frac{J}{|i-j|^{\alpha}}$$

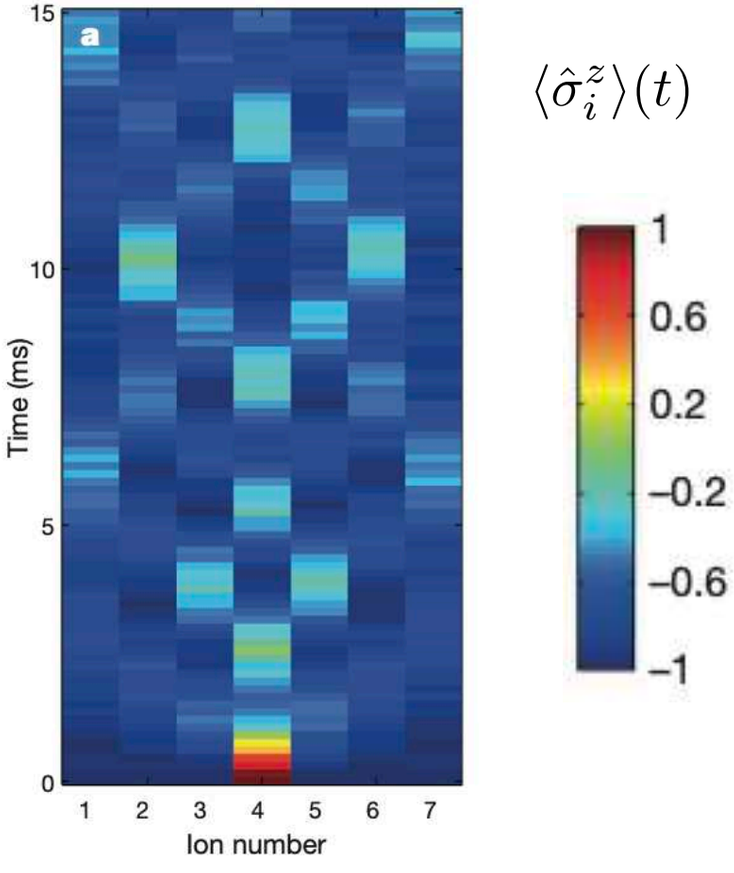
D. Porras & I. J. Cirac, Phys. Rev. Lett. 92, 207901 (2004)

$$\hat{H}_{\mathrm{TI}} = \sum_{i<j} J_{i,j}\hat{\sigma}_i^z\hat{\sigma}_j^z + h_x\sum_i \hat{\sigma}_i^x$$

Long-range transverse Ising model

P. Jurcevic, et al. Nature 511, 202 (2014)
Innsbruck

$$|\psi(t=0)\rangle = |\downarrow\ldots\downarrow\uparrow\downarrow\ldots\downarrow\rangle$$



$$\langle\hat{\sigma}_i^z\rangle(t)$$

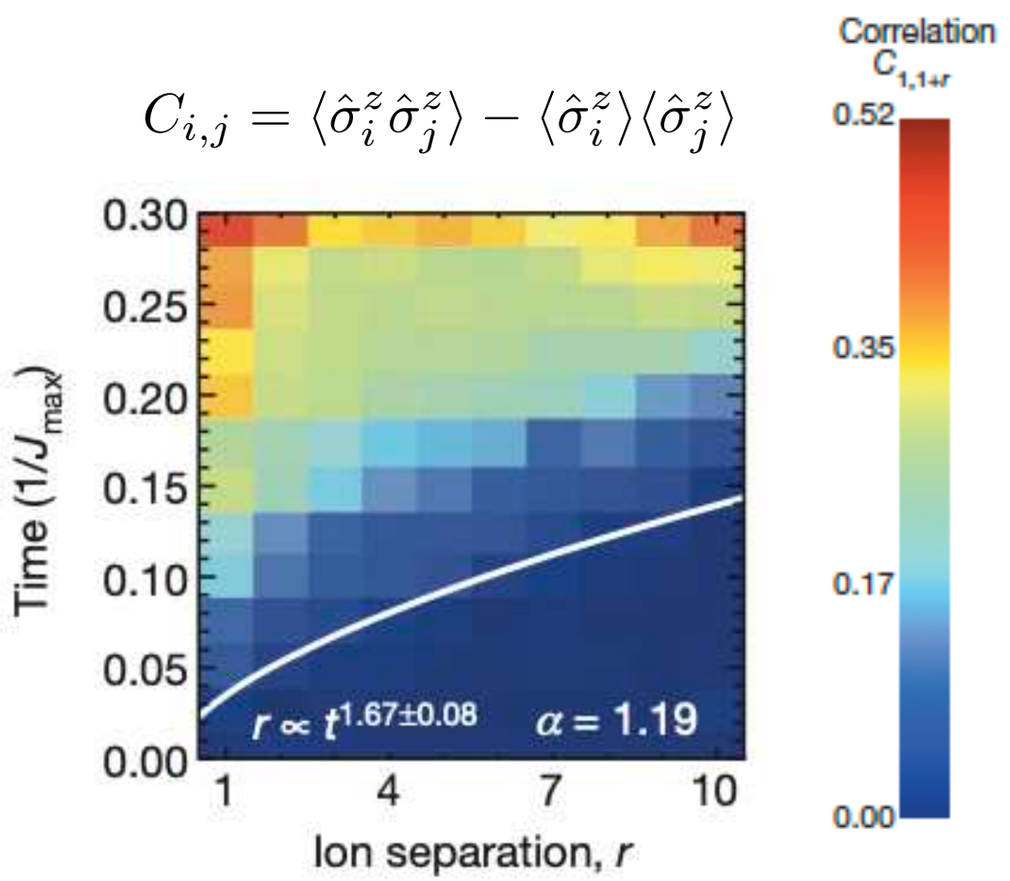
$$\hat{H}_{\mathrm{XY}} = \sum_{i<j} J_{i,j}\left(\hat{\sigma}_i^x\hat{\sigma}_j^x + \hat{\sigma}_i^y\hat{\sigma}_j^y\right) = 2\sum_{i<j} J_{i,j}\left(\hat{\sigma}_i^+\hat{\sigma}_j^- + \hat{\sigma}_i^-\hat{\sigma}_j^+\right)$$

Long-range XY model

P. Richerme, et al. Nature 511, 198 (2014)
JQI Maryland

$$|\psi(t=0)\rangle = |\downarrow\downarrow\ldots\downarrow\rangle$$

$$C_{i,j} = \langle\hat{\sigma}_i^z\hat{\sigma}_j^z\rangle - \langle\hat{\sigma}_i^z\rangle\langle\hat{\sigma}_j^z\rangle$$



Tutorial: Constructing a Hamiltonian and simulating dynamics

1. ***Construct the many-body spin operators***
2. *Construct the Hamiltonian*
3. *Simulate dynamics from an initial non-equilibrium state*
4. *Compute and plot observable dynamics*

- In a many-body system, how do we construct Hamiltonian matrices for such spin model?

$$\hat{H}_{\mathrm{TI}} = \sum_{i<j} J_{i,j} \hat{\sigma}_i^z \hat{\sigma}_j^z + h_x \sum_i \hat{\sigma}_i^x \qquad \textit{Transverse Ising model}$$

- What Hamiltonian terms actually mean: All terms are matrices: $2^N \times 2^N$

$$\hat{\sigma}_i^x = \cdots \otimes \mathbb{1} \otimes \underset{\textit{spin \#i}}{\hat{\sigma}^x} \otimes \mathbb{1} \otimes \cdots = \cdots \otimes \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \otimes \underset{\textit{spin \#i}}{\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}} \otimes \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \otimes \cdots$$

"tensor product" = "Kronecker product"

$$\hat{\sigma}_i^z \hat{\sigma}_j^z = \cdots \otimes \mathbb{1} \otimes \hat{\sigma}^z \otimes \mathbb{1} \otimes \mathbb{1} \otimes \hat{\sigma}^z \otimes \mathbb{1} \otimes \cdots =$$

$$\cdots \otimes \underset{\textit{spin \#i}}{\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}} \otimes \begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix} \otimes \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \otimes \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \otimes \underset{\textit{spin \#j}}{\begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix}} \otimes \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \otimes \cdots$$

Tutorial: Constructing a Hamiltonian and simulating dynamics

1. ***Construct the many-body spin operators***
2. *Construct the Hamiltonian*
3. *Simulate dynamics from an initial non-equilibrium state*
4. *Compute and plot observable dynamics*

$$\hat{H}_{\mathrm{TI}} = \sum_{i<j} J_{i,j} \hat{\sigma}_i^z \hat{\sigma}_j^z + h_x \sum_i \hat{\sigma}_i^x \qquad \textit{Transverse Ising model}$$

- All spin-operators can be fully constructed from spin-lowering operators only

$$\hat{\sigma}^+ = (\hat{\sigma}^-)^\dagger \qquad \hat{\sigma}_i^x = \hat{\sigma}_i^- + \hat{\sigma}_i^+ \qquad \hat{\sigma}_i^z = \hat{\sigma}_i^+ \hat{\sigma}_i^- - \hat{\sigma}_i^- \hat{\sigma}_i^+$$

$$|\downarrow\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \qquad \hat{\sigma}^- = \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix} \qquad \hat{\sigma}_i^- = \underbrace{\begin{pmatrix} 1 & & \\ & 1 & \\ & & \ddots \end{pmatrix}}_{2^{i-1} \times 2^{i-1}} \otimes \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix} \otimes \underbrace{\begin{pmatrix} 1 & & \\ & 1 & \\ & & \ddots \end{pmatrix}}_{2^{N-i} \times 2^{N-i}}$$

$$|\uparrow\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

Tutorial: Constructing a Hamiltonian and simulating dynamics

1. ***Construct the many-body spin operators***
2. *Construct the Hamiltonian*
3. *Simulate dynamics from an initial non-equilibrium state*
4. *Compute and plot observable dynamics*

```julia
using LinearAlgebra

sm = [0 1; 0 0]
# lowering operator for a singe spin:
#    |0> = [1;0]
#    |1> = [0;1]

N = 10  # number of spins
```

```
julia> sm
2×2 Matrix{Int64}:
 0  1
 0  0

julia> id = Matrix(I, 2, 2)
2×2 Matrix{Bool}:
 1  0
 0  1

julia> kron(sm, id)
4×4 Matrix{Int64}:
 0  0  1  0
 0  0  0  1
 0  0  0  0
 0  0  0  0
```

```
julia> sm'*sm - sm*sm'
2×2 Matrix{Int64}:
 -1  0
  0  1
```

```
julia> sm + sm'
2×2 Matrix{Int64}:
 0  1
 1  0
```

```
julia> kron(sm, id) * [0;0;1;0]
4-element Vector{Int64}:
 1
 0
 0
 0
```

$$|10\rangle \rightarrow |00\rangle$$

Tutorial: Constructing a Hamiltonian and simulating dynamics

1. ***Construct the many-body spin operators***
2. *Construct the Hamiltonian*
3. *Simulate dynamics from an initial non-equilibrium state*
4. *Compute and plot observable dynamics*

```julia
# Step 1: Buld a series of lowering operators
function build_lowering_ops(sm, N)

    sms = Vector{}()
    # this is an empty vector
    # ... it will be filled with matrices as elements

    for ii = 1:N
        ldim = 2 ^ (ii - 1) # the Hilbert space dimention left of spin ii
        rdim = 2 ^ (N - ii) # the Hilbert space dimention right of spin ii
        left_id = Matrix(I, ldim, ldim) # identity matrix for left Hilbert space
        right_id = Matrix(I, rdim, rdim) # identity matrix for right Hilbert space
        push!(sms, kron(left_id, sm, right_id))
    end

    return sms

end
```

```
julia> sms = build_lowering_ops(sm, 2)
2-element Vector{Any}:
 [0 0 1 0; 0 0 0 1; 0 0 0 0; 0 0 0 0]
 [0 1 0 0; 0 0 0 0; 0 0 0 1; 0 0 0 0]
```

```
julia> sms[2]
4×4 Matrix{Int64}:
 0  1  0  0
 0  0  0  0
 0  0  0  1
 0  0  0  0
```

Structured matrices, that could be build more efficiently, but this is safe.

Tutorial: Constructing a Hamiltonian and simulating dynamics

1. ***Construct the many-body spin operators***
2. *Construct the Hamiltonian*
3. *Simulate dynamics from an initial non-equilibrium state*
4. *Compute and plot observable dynamics*

```julia
# Build spin operators
function build_spin_ops_bad(sms)

    sxs = Vector{}()
    szs = Vector{}()
    for ii = 1:length(sms)
        push!(sxs, sms[ii] + sms[ii]')
        push!(szs, sms[ii]' * sms[ii] - sms[ii] * sms[ii]')
    end

    return sxs, szs
end
```

Julia: Arrays passed by reference

```
julia> sms = build_lowering_ops(sm, 10)
```

```
julia> @time sxs, szs = build_spin_ops_bad(sms);
  2.881166 seconds (159 allocations: 320.005 MiB, 1.87% gc time)
```

Tutorial: Constructing a Hamiltonian and simulating dynamics

1. ***Construct the many-body spin operators***
2. *Construct the Hamiltonian*
3. *Simulate dynamics from an initial non-equilibrium state*
4. *Compute and plot observable dynamics*

```julia
# Step 1: Buld the necessary many-body spin operators
function build_spin_ops(sm, N)

    sxs = Vector{}()
    szs = Vector{}()

    for ii = 1:N
        ldim = 2 ^ (ii - 1) # the Hilbert space dimention left of spin ii
        rdim = 2 ^ (N - ii) # the Hilbert space dimention right of spin ii
        left_id = Matrix(I, ldim, ldim) # identity matrix for left Hilbert space
        right_id = Matrix(I, rdim, rdim) # identity matrix for right Hilbert space
        push!(sxs, kron(left_id, sm + sm', right_id))
        push!(szs, kron(left_id, sm'*sm - sm*sm', right_id))
    end

    return sxs, szs

end
```

```
julia> @time sxs, szs = build_spin_ops(sm, N);
  0.023566 seconds (251 allocations: 182.010 MiB, 13.41% gc time)
```

Tutorial: Constructing a Hamiltonian and simulating dynamics

1. *Construct the many-body spin operators*
2. ***Construct the Hamiltonian***
3. *Simulate dynamics from an initial non-equilibrium state*
4. *Compute and plot observable dynamics*

$$\hat{H}_{\mathrm{TI}} = \sum_{i<j} J_{i,j} \hat{\sigma}_i^z \hat{\sigma}_j^z + h_x \sum_i \hat{\sigma}_i^x$$

long-range Ising model

$$J_{ij} = \frac{J}{|i-j|^{\alpha}}$$

Tutorial: Constructing a Hamiltonian and simulating dynamics

1. *Construct the many-body spin operators*
2. ***Construct the Hamiltonian***
3. *Simulate dynamics from an initial non-equilibrium state*
4. *Compute and plot observable dynamics*

$$\hat{H}_{\mathrm{TI}} = \sum_{i<j} J_{i,j}\hat{\sigma}_i^z\hat{\sigma}_j^z + h_x \sum_i \hat{\sigma}_i^x \qquad \textit{long-range Ising model} \qquad J_{ij} = \frac{J}{|i-j|^\alpha}$$

- Hamiltonian can be fully constructed from spin-operators

```julia
# Step 2: Build Hamiltonian
function build_hamiltonian(J, alpha, hx, sxs, szs)

    N = length(sxs)
    H = zeros(Float64, 2^N, 2^N)

    for ii = 1:N
        H += hx .* sxs[ii]
        for jj = (ii+1):N
            H += (J / (jj-ii)^alpha) .* szs[ii] * szs[jj]
        end
    end

    return H

end
```

```
julia> H = build_hamiltonian(1, 1, 1, sxs, szs)
1024×1024 Matrix{Float64}:
 19.2897   1.0      1.0     0.0     1.0    0.0
  1.0     13.6317   0.0     1.0     0.0    1.0
  1.0      0.0     11.854   1.0     0.0    0.0
  0.0      1.0      1.0    10.196   0.0    0.0
  1.0      0.0      0.0     0.0    11.104  1.0
  0.0      1.0      0.0     0.0     1.0    7.44603
```

```
julia> H = build_hamiltonian(1, 1, 2, sxs, szs)
4×4 Matrix{Float64}:
 1.0   2.0   2.0  0.0
 2.0  -1.0   0.0  2.0
 2.0   0.0  -1.0  2.0
 0.0   2.0   2.0  1.0
```

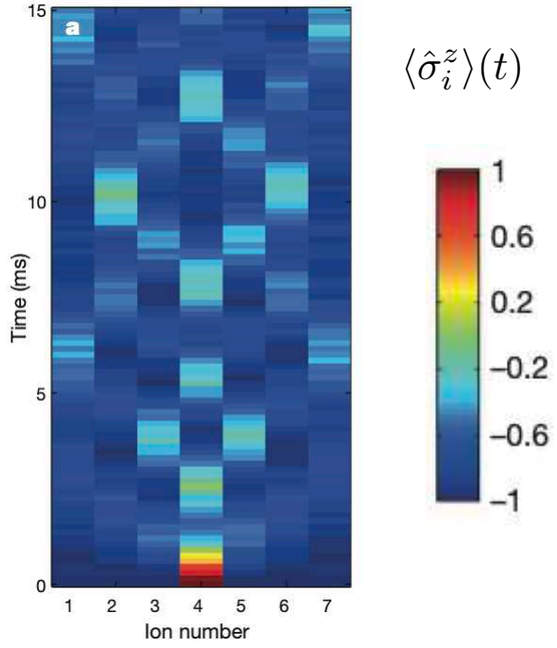
Tutorial: Constructing a Hamiltonian and simulating dynamics

1. *Construct the many-body spin operators*
2. *Construct the Hamiltonian*
3. ***Simulate dynamics from an initial non-equilibrium state***
4. *Compute and plot observable dynamics*

$$\hat{H}_{\mathrm{TI}} = \sum_{i<j} J_{i,j} \hat{\sigma}_i^z \hat{\sigma}_j^z + h_x \sum_i \hat{\sigma}_i^x \qquad \textit{long-range Ising model} \qquad J_{ij} = \frac{J}{|i-j|^\alpha}$$

- We have the Hamiltonian, so now we need to simulate the dynamics with the Hamiltonian

$$|\psi(t=0)\rangle = |\downarrow \ldots \downarrow \uparrow \downarrow \ldots \downarrow\rangle$$



Obtain initial state with $|\psi(t=0)\rangle = \hat{\sigma}_{N/2}^x |\downarrow \ldots \downarrow\rangle$

To evaluate at specific times, introduce a stroboscopic evolution

$$\hat{U}(\Delta t) = \mathrm{e}^{-\mathrm{i}\Delta t \hat{H}}$$

(time step is pure convenience, no approximation)

Tutorial: Constructing a Hamiltonian and simulating dynamics

1. *Construct the many-body spin operators*
2. *Construct the Hamiltonian*
3. ***Simulate dynamics from an initial non-equilibrium state***
4. *Compute and plot observable dynamics*

```
function ti_simulation(N, J, alpha, hx, dt, steps)

    sm = [0 1; 0 0]
    sxs, szs = build_spin_ops(sm, N)

    H = build_hamiltonian(J, alpha, hx, sxs, szs)

    # initial state
    psi = zeros(ComplexF64, 2^N)
    psi[1] = 1.0 # the all zero state
    psi = sxs[ div(N,2) + 1 ] * psi # flip spin in center
```

```
    # build evolution operator for a single time-step
    U = exp(-1im .* dt .* H)

    for tt = 1:steps
        println("Step $tt/$steps - norm(psi) = $(norm(psi))")
        psi = U * psi
    end

    return nothing

end
```

```
julia> ti_simulation(10, 1, 1, 1, 0.1, 10)
Step 1/10 - norm(psi) = 1.0
Step 2/10 - norm(psi) = 0.9999999999999998
Step 3/10 - norm(psi) = 0.9999999999999998
Step 4/10 - norm(psi) = 0.9999999999999993
Step 5/10 - norm(psi) = 1.0000000000000004
```

Tutorial: Constructing a Hamiltonian and simulating dynamics

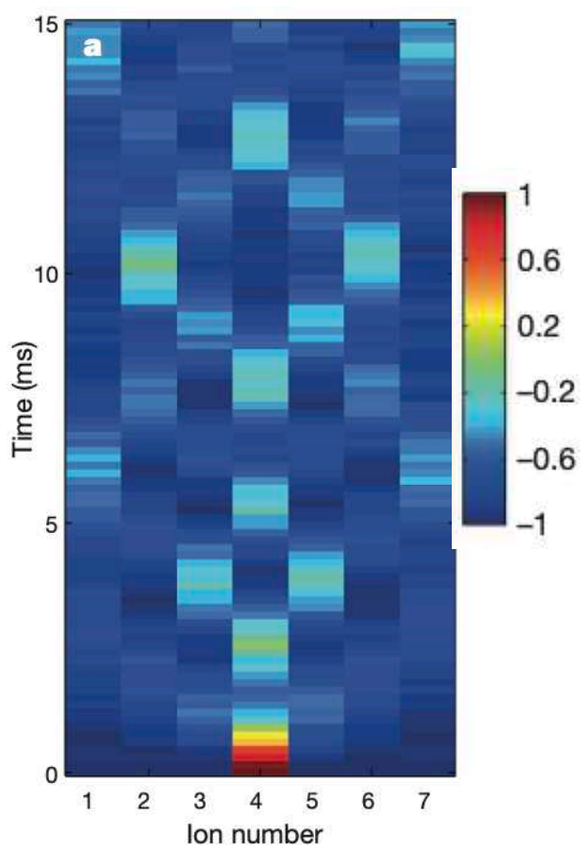
1. Construct the many-body spin operators

2. Construct the Hamiltonian

3. Simulate dynamics from an initial non-equilibrium state

4. Compute and plot observable dynamics

- Produce the plot for the spin-z evolution



Use simply: $\langle\psi(t)|\,\hat{\sigma}_i^z\,|\psi(t)\rangle$

Tutorial: Constructing a Hamiltonian and simulating dynamics

1. *Construct the many-body spin operators*
2. *Construct the Hamiltonian*
3. *Simulate dynamics from an initial non-equilibrium state*
4. ***Compute and plot observable dynamics***

```
    out_sz = zeros(Float64, steps, N)

    for tt = 1:steps
        println("Step $tt/$steps - norm(psi) = $(norm(psi))")

        for ii = 1:N
            out_sz[tt, ii] = psi' * szs[ii] * psi
        end

        psi = U * psi
    end

    return out_sz
```

```
using Plots
```

$$N = 11$$

```
julia> out_sz = ti_simulation(11, 1, 10, 1, 0.1, 101)
```

```
julia> heatmap(out_sz)
```

```
julia> heatmap(1:11, 0:dt:10,  out_sz)
```

Tutorial: Constructing a Hamiltonian and simulating dynamics

1. *Construct the many-body spin operators*
2. *Construct the Hamiltonian*
3. *Simulate dynamics from an initial non-equilibrium state*
4. ***Compute and plot observable dynamics***

```
function main()

    N = 7 # number of spins

    J = 1 # defines energy/time units
    alpha = 1.36 # interaction range
    hx = 1 # transverse field

    dt = 0.1 # time step for plotting
    steps = 101

    out_sz = ti_simulation(N, J, alpha, hx, dt, steps)
```

```
using LaTeXStrings

# nice plot
cmap = cgrad(:RdBu)
default(
    tickfontsize = 10,
    labelfontsize = 12,
    fontfamily="times",
    colorbar_ticks=-1:0.5:1,
    color = reverse(cmap),
    aspect_ratio=1.2,
    dpi=200)

    heatmap(1:N, 0:dt:((steps-1) * dt),  out_sz)
    xlims!((0.5, N+0.5))
    xlabel!(L"$i$")
    ylabel!(L"$tJ$")

    #display(h)
    savefig("ti_ising.png")

    return nothing

end
```

Tutorial: Constructing a Hamiltonian and simulating dynamics

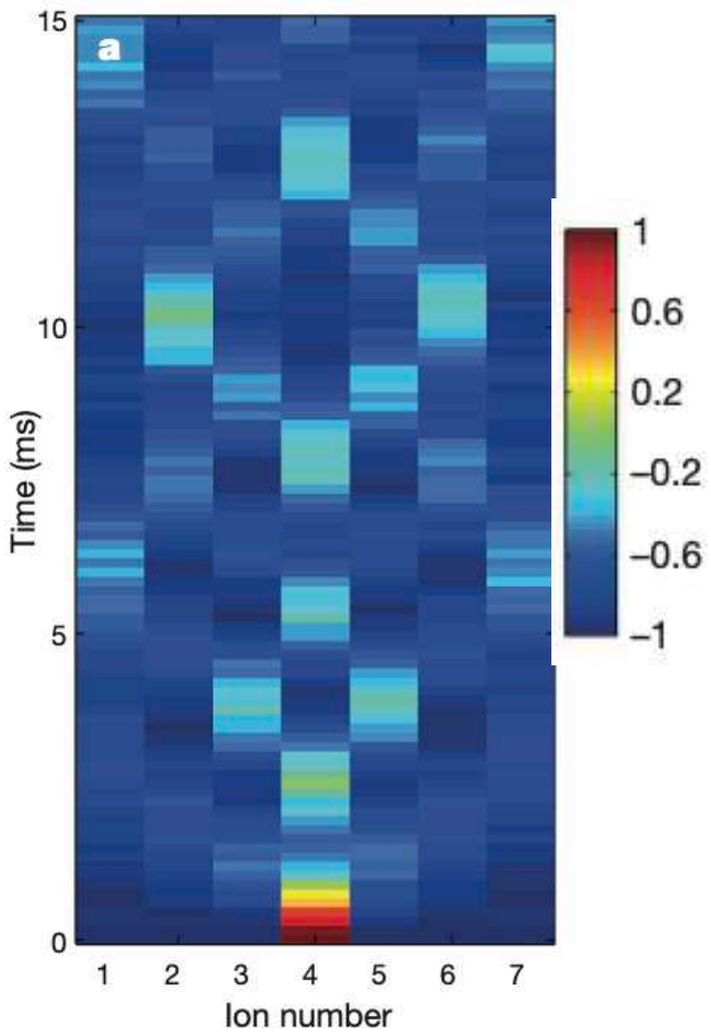
1. Construct the many-body spin operators

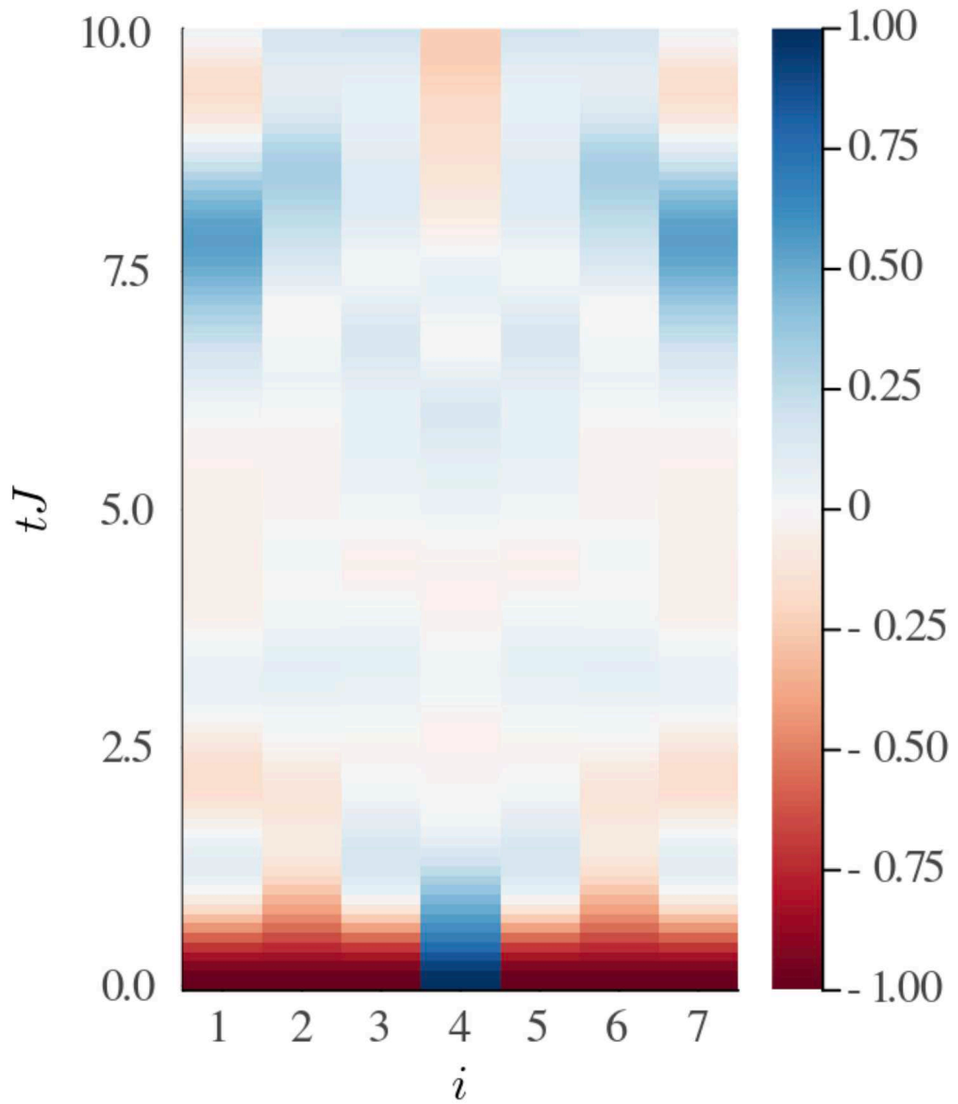
2. Construct the Hamiltonian

3. Simulate dynamics from an initial non-equilibrium state

4. Compute and plot observable dynamics

$$\alpha = 1.36$$





Tutorial: Constructing a Hamiltonian and simulating dynamics

- **Problem:** We are currently wasting memory

$$N = 10$$

Even for a full coupling matrix, the Hamiltonian is very sparse!

Most elements are zero!

```
julia> sxs, szs = build_spin_ops(sm, N); H = build_hamiltonian(1, 3, 1, sxs, szs)
1024×1024 Matrix{Float64}:
 10.4256  1.0      1.0      0.0      1.0      0.0      0.0      0.0      …  0.0      0.0
  1.0     8.03249  0.0      1.0      0.0      1.0      0.0      0.0         0.0      0.0
  1.0     0.0      6.03523  1.0      0.0      0.0      1.0      0.0         0.0      0.0
  0.0     1.0      1.0      7.64217  0.0      0.0      0.0      1.0         0.0      0.0
  1.0     0.0      0.0      0.0      5.78914  1.0      1.0      0.0         0.0      0.0
  0.0     1.0      0.0      0.0      1.0      3.89607  0.0      1.0      …  0.0      0.0
  0.0     0.0      1.0      0.0      1.0      0.0      5.39882  1.0         0.0      0.0
  0.0     0.0      0.0      1.0      0.0      1.0      1.0      7.50575     0.0      0.0
  1.0     0.0      0.0      0.0      0.0      0.0      0.0      0.0         0.0      0.0
  0.0     1.0      0.0      0.0      0.0      0.0      0.0      0.0         0.0      0.0
  0.0     0.0      1.0      0.0      0.0      0.0      0.0      0.0      …  0.0      0.0
  0.0     0.0      0.0      1.0      0.0      0.0      0.0      0.0         0.0      0.0
  ⋮                                          ⋮                             ⋱
  0.0     0.0      0.0      0.0      0.0      0.0      0.0      0.0         0.0      0.0
  0.0     0.0      0.0      0.0      0.0      0.0      0.0      0.0         0.0      0.0
  0.0     0.0      0.0      0.0      0.0      0.0      0.0      0.0         0.0      0.0
  0.0     0.0      0.0      0.0      0.0      0.0      0.0      0.0      …  0.0      0.0
  0.0     0.0      0.0      0.0      0.0      0.0      0.0      0.0         1.0      1.0
  0.0     0.0      0.0      0.0      0.0      0.0      0.0      0.0         5.39882  0.0
  0.0     0.0      0.0      0.0      0.0      0.0      0.0      0.0         0.0      3.89607
```

- Elements in matrix: 2^{2N}
- Non-zero elements: $\mathcal{O}(2^N)$

This example:

```
julia> sum(abs.(H) .> 1e-16)
11264
```

```
julia> sum(abs.(H) .> 1e-16)/2^20
0.0107421875
```

Only 1% filled … let's use **Sparse Matrices!**

- We should be using **sparse matrices**
  = a data format where we only store the non-zero elements (with their index positions)

Tutorial: Constructing a Hamiltonian and simulating dynamics

- With our construction, changing to sparse matrices is trivial!

```
using SparseArrays

# sm = [0 1; 0 0]
 sm = sparse([0 1; 0 0])
```

```
julia> sm
2×2 SparseMatrixCSC{Int64, Int64} with 1 stored entry:
 ·  1
 ·  ·
```

```
# Step 1: Build the necessary sparse many-body spin operators
function build_sparse_spin_ops(sm, N)

    sxs = Vector{}()
    szs = Vector{}()

    for ii = 1:N
        ldim = 2 ^ (ii - 1) # the Hilbert space dimention left of spin ii
        rdim = 2 ^ (N - ii) # the Hilbert space dimention right of spin ii
        #left_id = Matrix(I, ldim, ldim) # identity matrix for left Hilbert space
        left_id = sparse(I, ldim, ldim) # identity matrix for left Hilbert space
        #right_id = Matrix(I, rdim, rdim) # identity matrix for right Hilbert space
        right_id = sparse(I, rdim, rdim) # identity matrix for right Hilbert space
        push!(sxs, kron(left_id, sm + sm', right_id))
        push!(szs, kron(left_id, sm'*sm - sm*sm', right_id))
    end

    return sxs, szs

end
```

```
# Step 2: Build sparse Hamiltonian
function build_sparse_hamiltonian(J, alpha, hx, sxs, szs)

    N = length(sxs)
    #H = zeros(Float64, 2^N, 2^N)
    H = spzeros(Float64, 2^N, 2^N)
```

Tutorial: Constructing a Hamiltonian and simulating dynamics

Full version

```
julia> @time begin; sm = [0 1; 0 0]; sxs, szs = build_spin_ops(sm, 10); H = build_hamiltonian(1, 3, 1, sxs, szs); end;
  0.830853 seconds (821 allocations: 1.397 GiB, 48.33% gc time)
```

Sparse version

```
julia> @time begin; sxs, szs = build_sparse_spin_ops(sparse([0 1; 0 0]), 10); H = build_sparse_hamiltonian(1, 3, 1, sxs, szs); end;
  0.007831 seconds (2.78 k allocations: 10.354 MiB, 59.62% gc time)
```

- Much more memory and time-efficient to construct the Hamiltonian!
- **Problem:** A full diagonalization does not make use of the sparsity!

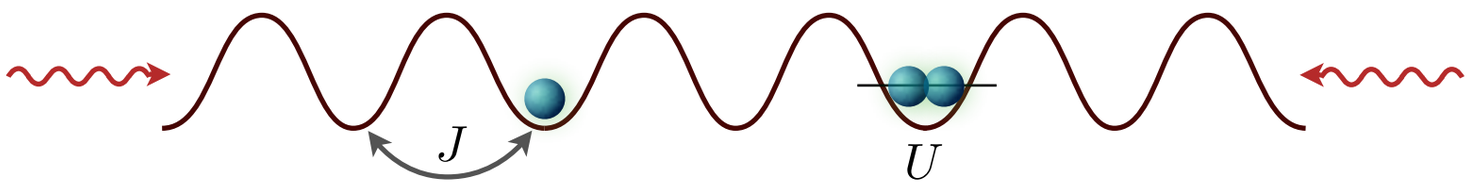
$$\hat{V}^\dagger \hat{H} \hat{V} = \hat{E} \quad \Leftrightarrow \quad \hat{H}\hat{V} = \hat{V}\hat{E} \quad \Leftrightarrow \quad \hat{H} = \hat{V}\hat{E}\hat{V}^\dagger$$

V would be a full matrix!

We have to be smarter to make use of sparse matrices in the time-evolution!

Tutorial: Constructing a Hamiltonian and simulating dynamics

- **Remark:** For a system of bosons in a lattice, very similar construction possible



$$\hat{H} = -J \sum_i (\hat{b}_i \hat{b}_{i+1}^\dagger + \hat{b}_i^\dagger \hat{b}_{i+1}) + \frac{U}{2} \sum_i \hat{b}_i^\dagger \hat{b}_i^\dagger \hat{b}_i \hat{b}_i$$

Introduce cutoff of Fock space basis on each site

But: *Number not conserved!*

$$D = n_{\max}^N$$

Define bosonic field operators and use Kroneckers as before

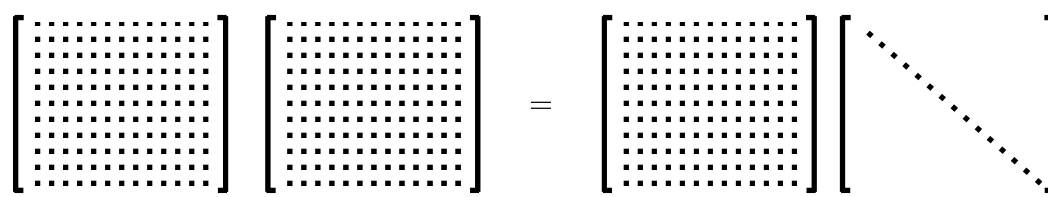
```
# |n=0> = [1;0;0;0; ...]
# |n=2> = [0;0;1;0; ...]
# b |n=2> = sqrt(n) * [0;1;0;0; ...]
nm = 5 # cutoff for max. nm-1 bosons)
b = sparse(1:(nm-1), 2:nm, sqrt.(1:(nm-1)), nm, nm)
```

Recap

- We discussed how to represent numbers as bits, and associated relative machine precisions ~1e-16

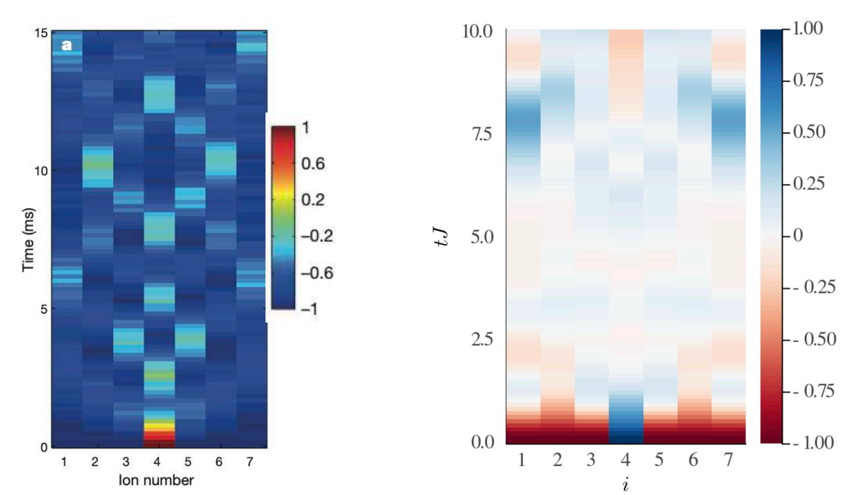
0	1 0 0 0 0 0 0 0 0 0 1	0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

- We describe systems with state-vectors (not only in QM). If the problem is linear (QM), an exact diagonalization of the Hamiltonian allows to simulate arbitrary time-evolution



- In tutorial style we have seen how to construct many-body spin (or other) operators using Kronecker product functions. We used this to simulate dynamics in a transverse Ising spin model (which can e.g. be realized with trapped ions)

$$\hat{\sigma}_i^- = \begin{pmatrix} 1 & & \\ & 1 & \\ & & \ddots \end{pmatrix} \otimes \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix} \otimes \begin{pmatrix} 1 & & \\ & 1 & \\ & & \ddots \end{pmatrix}$$



- It's important to use sparse matrices.