

Speeding up Quantum Mechanics - Matrix Types

Carsten Bauer

August 5, 2019

0.1 Free Fermions on a Chain

$$\mathcal{H} = -t \sum_{\langle i,j \rangle} c_i^\dagger c_j + \mu \sum_i n_i$$

Here, t is the hopping amplitude, μ is the chemical potential, and c, c^\dagger are creation and annihilation operators.

For the sake of the argument of this tutorial, we'll consider **open boundary conditions**.

Since the fermions are *not* interacting, we can work in the *single particle basis* and do not have to worry about how to construct a basis for the many-body Fock space.

We use the canonical cartesian basis in which one uses 0s to indicate empty sites and a 1 for the particle's site, i.e. $|00100\rangle$ represents the basis state which has the particle exclusively on the 3rd site.

If you aren't familiar with second quantization just think of \mathcal{H} as any quantum mechanical operator that can be represented as a matrix.

Let's build the Hamiltonian matrix.

```
using LinearAlgebra # makes Julia speak linear algebra fluently
```

```
N = 100 # number of sites
t = 1
μ = -0.5
```

```
H = diagm(0 => fill(μ, N), 1 => fill(-t, N-1), -1 => fill(-t, N-1))
```

```
100×100 Array{Float64,2}:
```

```
-0.5 -1.0  0.0  0.0  0.0  0.0  ...  0.0  0.0  0.0  0.0  0.0  0.0
-1.0 -0.5 -1.0  0.0  0.0  0.0      0.0  0.0  0.0  0.0  0.0  0.0
 0.0 -1.0 -0.5 -1.0  0.0  0.0      0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0 -1.0 -0.5 -1.0  0.0      0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0 -1.0 -0.5 -1.0      0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0 -1.0 -0.5  ...  0.0  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
 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  0.0  0.0  0.0  0.0  0.0
```

```

0.0  0.0  0.0  0.0  0.0  0.0  -0.5 -1.0  0.0  0.0  0.0  0.0
0.0  0.0  0.0  0.0  0.0  0.0  ... -1.0 -0.5 -1.0  0.0  0.0  0.0
0.0  0.0  0.0  0.0  0.0  0.0  0.0 -1.0 -0.5 -1.0  0.0  0.0
0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0 -1.0 -0.5 -1.0  0.0
0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0 -1.0 -0.5 -1.0
0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0 -1.0 -0.5

```

A typical thing one does in quantum mechanics is calculate the expectation value of an operator, like \mathcal{H} , with respect to some quantum mechanical state ψ .

```

ψ = normalize(rand(N)); # some state

"""ev(0, ψ)Calculates the quantum mechanical expectation value `<ψ|0|ψ>`, where `0` is
    an operator and `ψ` is a state."""
ev(0, ψ) = ψ'*0*ψ;

ev(H, ψ) # calculate the energy

-1.8924233111013842

```

Let's see how long this calculation takes.

```

using BenchmarkTools

@btime ev($H, $ψ);

10.101 μs (1 allocation: 896 bytes)

```

0.1.1 Utilizing the matrix structure

Since `typeof(H) == Matrix{Float64}`, we have so far dispatched to generic matrix multiplications (and transposition) in the `ev` function.

What we should do is exploit the *structure* of our Hamiltonian and use specialized, faster implementations of these operations. After all, most entries in \mathcal{H} are equal to zero and multiplications involving those zeros are clearly unnecessary.

To speed up our expectation value computation, we can tell Julia about the *sparsity* of our Hamiltonian. The way we do this, is by changing H 's type.

```

using SparseArrays
Hsparse = sparse(H)

100×100 SparseArrays.SparseMatrixCSC{Float64,Int64} with 298 stored entries
:
 [1 , 1] = -0.5
 [2 , 1] = -1.0
 [1 , 2] = -1.0
 [2 , 2] = -0.5
 [3 , 2] = -1.0
 [2 , 3] = -1.0
 [3 , 3] = -0.5
 [4 , 3] = -1.0
 [3 , 4] = -1.0
 ⋮
 [98 , 97] = -1.0
 [97 , 98] = -1.0
 [98 , 98] = -0.5

```

```
typeof(Hsparse)
```

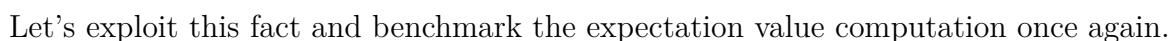
As we can see, `Hsparse` is now a sparse matrix. Let's see how long our computation takes after this simple type change.

```
350.948 ns (2 allocations: 912 bytes)
```

Inspecting our Hamiltonian more closely, we note that it is not just sparse but actually *tridiagonal*.

```
using UnicodePlots
```

Sparsity Pattern



```
Htri = Tridiagonal(H)
```

