An Intro to HOQST - closed-system simulation

Huo Chen

November 10, 2020

0.1 Close System Examples

This notebook will get you started with HOQST by introducing you to the functionality for solving colsed-system equations.

0.1.1 Define the Hamiltonin

The first step is to define an Hamiltonian. In this tutorial we focus on a 2-level system with the following Hamiltonian

$$H(s) = -\sigma_z$$

where $s = t/t_f$ is the dimensionless time and t_f is the total evolution time. We use a constant Hamiltonian so the simulation results can be trivially confirmed. The syntax is the same for time dependent Hamiltonians. Let's first define the Hamiltonian by:

```
using QuantumAnnealingTools, OrdinaryDiffEq, Plots
# define the Hamiltonian
H = DenseHamiltonian([(s)->1.0], [-σz], unit=:ħ)
```

```
DenseHamiltonian with Complex{Float64}
with size: (2, 2)
```

In this example, we use the `DenseHamiltonian` object. The syntax is the same for other type of Hamiltonians.

The closed-system evolution is completely specified by the Hamiltonian and the initial state. We can combine them into a single `Annealing` object by:

```
# define the initial state by PauliVec[k][j],
# which are the jth eigenvector of the
# Pauli matrix σ_k
u0 = PauliVec[1][1]
# define total evolution time in (ns)
tf = 10
# combine H and u0 into an Annealing object
annealing = Annealing(H, u0)
```

```
Annealing with hType QTBase.DenseHamiltonian{Complex{Float64}} and uType Ar
ray{Complex{Float64},1}
u0 with size: (2,)
```

The initial state in above code block is $|\phi(0)\rangle = |+\rangle$.

We will consider three variants of the closed-system equations in this tutorial.

0.1.2 Schrodinger equation

We start with the Schrodinger equation \begin{equation} \lvert \dot{\phi} \rangle = -i t_f H(s) \lvert \phi \rangle \ . \end{equation}

To solve the this differential equation, we need to choose a proper algortihm. HOQST relies on `OrdinaryDiffEq.jl` as the low level solver, which support a large collection of algorithms. We do not guarantee compatibilities to every solver in this list. Users can try specific algorithms if they are interested. We provide a list of algorithms we tested and recommended here:

1. The default Tsitouras 5/4 Runge-Kutta method(Tsit5()).

   This is the default method in `OrdinaryDiffEq` and works well in most cases.

2. A second order A-B-L-S-stable one-step ESDIRK method(TRBDF2()).

   This is the method widely used in large scale classical circuit simulations. Because this method has order of 2, it is recommended to use smaller error tolerance comparing with other higher order methods.

3. A simple linear exponential method(LinearExponential()).

   This method simply discretize the Hamiltonian and do matrix exponential for each interval.

4. Adaptive exponential Rosenbrock methods(Exprb32()/Exprb43()).

   This method belongs to the adaptive exponential Runge-Kutta method family.

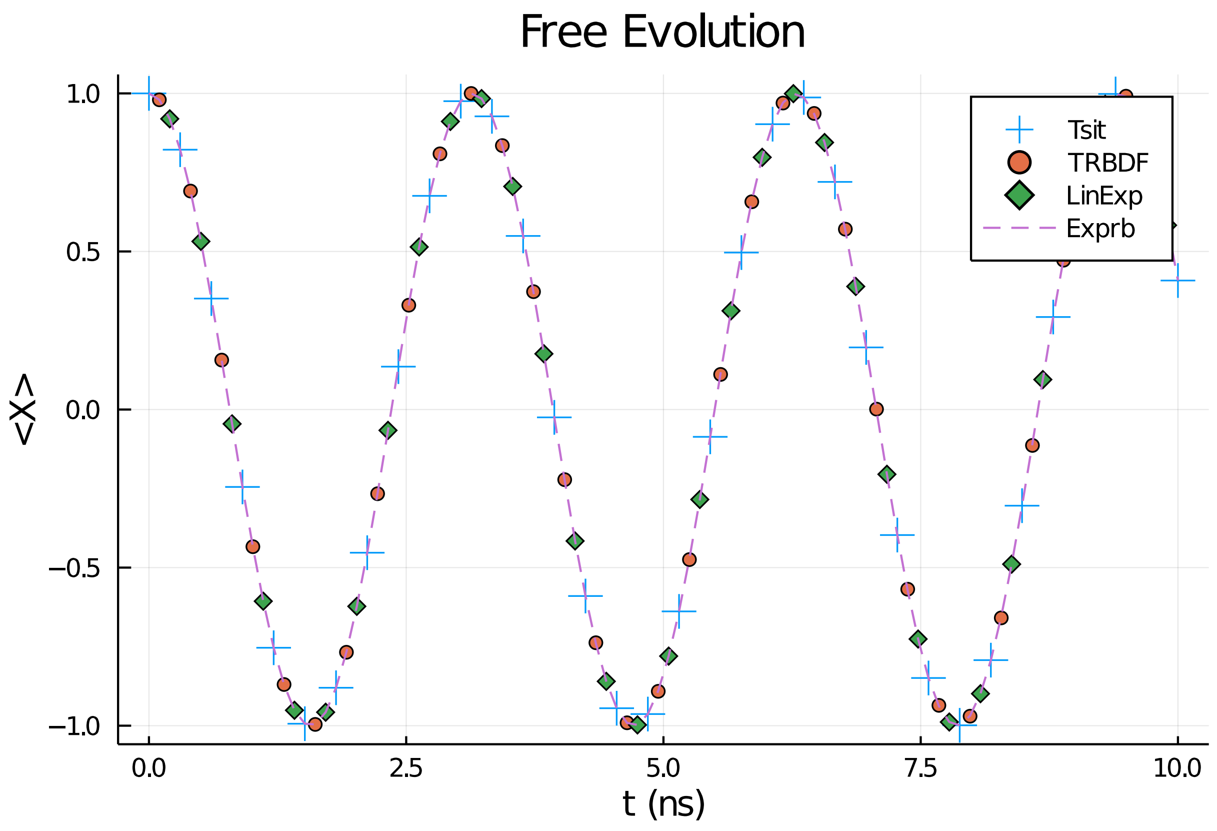
It is important to notice that, method 3 and 4 are exponential methods which would preserve the norm of the state vectors. To solve our the Schrodinger equation we use the function `solve_schrodinger`.

```
sol_tsit = solve_schrodinger(annealing, tf, alg=Tsit5(), abstol=1e-6, reltol=1e-6);
sol_trbdf = solve_schrodinger(annealing, tf, alg=TRBDF2(), abstol=1e-6, reltol=1e-6);
# LinearExponential is a fixed step size method, user need to specify the time steps
using keyword argument `tstops`.
sol_linexp = solve_schrodinger(annealing, tf, alg=LinearExponential(), abstol=1e-6,
reltol=1e-6, tstops=range(0,tf,length=100));
# Even though Exprb method is an adaptive method, it tends to jump a lot of middle
points. So if you want accurate evolution in the middle,
# it is better to manually add more points for the algorithm.
sol_exprb32 = solve_schrodinger(annealing, tf, alg=Exprb32(),
tstops=range(0,tf,length=100));
```

We plot the observable $\langle X \rangle$ during the evolution.

```
t_list = range(0,tf,length=100)
tsit = []
trbdf = []
linexp = []
```

```
exprb32 = []
for s in t_list
    push!(tsit, real(sol_tsit(s)'*σx*sol_tsit(s)))
    push!(trbdf, real(sol_trbdf(s)'*σx*sol_trbdf(s)))
    push!(linexp, real(sol_linexp(s)'*σx*sol_linexp(s)))
    push!(exprb32, real(sol_exprb32(s)'*σx*sol_exprb32(s)))
end
scatter(t_list[1:3:end], tsit[1:3:end], label="Tsit", marker=:+, markersize=8)
scatter!(t_list[2:3:end], trbdf[2:3:end], label="TRBDF")
scatter!(t_list[3:3:end], linexp[3:3:end], label="LinExp", marker=:d)
plot!(t_list, exprb32, label="Exprb", linestyle=:dash)
xlabel!("t (ns)")
ylabel!("<X>")
title!("Free Evolution")
```



0.1.3 Other close system equations

The package also contains several other closed-system solvers.

Von Neumann equation Von Neumann equation is the "Schrodinger" equation for density matrices

$$\dot{\rho} = -it_f[H(s), \rho] \ .$$

Even though Von Neumann equation is equivalent to the Schrodinger equation, it is sometimes numerically more stable than the Schrodinger equation. Users is encouraged to try to solve it using different algorithms.

```
annealing = Annealing(H, u0)
```

```
sol_tsit = solve_von_neumann(annealing, tf, alg=Tsit5(), abstol=1e-6, reltol=1e-6)
```

```
retcode: Success
Interpolation: specialized 4th order "free" interpolation
t: 78-element Array{Float64,1}:
  0.0
  0.025416471135423512
  0.07162657822452011
  0.12691325854682373
  0.19490751014915958
  0.2726355853793798
  0.3605272018981224
  0.4567317871059021
  0.5605609353632346
  0.6707029681719032


⋮@*(8.939120309934459.079043385421589.2189664738821449.3588895753161469.4988126804568449.6387357985709
78-element Array(*@{Array{Complex{Float64},2},1}:
 [0.4999999999999999 + 0.0im 0.4999999999999999 + 0.0im; 0.4999999999999999
 + 0.0im 0.4999999999999999 + 0.0im]
 [0.4999999999999999 + 0.0im 0.4993541420867104 + 0.025405526573483335im; 0
.4993541420867104 - 0.025405526573483335im 0.4999999999999999 + 0.0im]
 [0.4999999999999999 + 0.0im 0.49487840083247414 + 0.07138184906644161im; 0
.49487840083247414 - 0.07138184906644161im 0.4999999999999999 + 0.0im]
 [0.4999999999999999 + 0.0im 0.4839793174578459 + 0.12555484945742712im; 0.
4839793174578459 - 0.12555484945742712im 0.4999999999999999 + 0.0im]
 [0.4999999999999999 + 0.0im 0.46248968537194657 + 0.19000865966633518im; 0
.46248968537194657 - 0.19000865966633518im 0.4999999999999999 + 0.0im]
 [0.4999999999999999 + 0.0im 0.427493339167494 + 0.25932497858582826im; 0.4
27493339167494 - 0.25932497858582826im 0.4999999999999999 + 0.0im]
 [0.4999999999999999 + 0.0im 0.37555502540351415 + 0.3300885061538326im; 0.
37555502540351415 - 0.3300885061538326im 0.4999999999999999 + 0.0im]
 [0.4999999999999999 + 0.0im 0.30550378150848995 + 0.39581237672432157im; 0
.30550378150848995 - 0.39581237672432157im 0.4999999999999999 + 0.0im]
 [0.4999999999999999 + 0.0im 0.2173361827963109 + 0.450294326336122im; 0.21
73361827963109 - 0.450294326336122im 0.4999999999999999 + 0.0im]
 [0.4999999999999999 + 0.0im 0.1136919542765893 + 0.48690259218729237im; 0.
1136919542765893 - 0.48690259218729237im 0.4999999999999999 + 0.0im]
 ⋮@*([0.4999999999999999 + 0.0im 0.28210754285167167 - 0.4128134549525751im;
0.28210754285167167 + 0.4128134549525751im 0.4999999999999999 + 0.0im][0.4999999999999999
+ 0.0im 0.3851552754690854 - 0.3188338459667725im; 0.3851552754690854 +
0.3188338459667725im 0.4999999999999999 + 0.0im][0.4999999999999999 + 0.0im
0.45823633996555174 - 0.20004763813288137im; 0.45823633996555174 + 0.20004763813288137im
0.4999999999999999 + 0.0im][0.4999999999999999 + 0.0im 0.4956647185696421 -
0.06569688565511934im; 0.4956647185696421 + 0.06569688565511934im 0.4999999999999999 +
0.0im][0.4999999999999999 + 0.0im 0.49452831997779884 + 0.07376535766452154im;
0.49452831997779884 - 0.07376535766452154im 0.4999999999999999 +
0.0im][0.4999999999999999 + 0.0im 0.4549155549065749 + 0.207488357233397im;
0.4549155549065749 - 0.207488357233397im 0.4999999999999999 + 0.0im][0.4999999999999999 +
0.0im 0.37990846483481877 + 0.32506788838972583im; 0.37990846483481877 -
0.32506788838972583im 0.4999999999999999 + 0.0im][0.4999999999999999 + 0.0im
0.2753429188830746 + 0.4173557762487389im; 0.2753429188830746 - 0.4173557762487389im
0.4999999999999999 + 0.0im][0.4999999999999999 + 0.0im 0.20403994993813357 +
0.4564726338136446im; 0.20403994993813357 - 0.4564726338136446im 0.4999999999999999 +
0.0im]
```

As shown below, the solution given by the solver is the density matrix instead of state vector:

```
sol_tsit(0.5)
```

```
2×@*(2 Array(*@{Complex{Float64},2}:
      0.5+0.0im       0.270151+0.420736im
 0.270151-0.420736im       0.5+0.0im
```

Recommended algorithm Only explicit methods are supported for solving equations w.r.t. density matrices. Vectorization is needed for implicit methods. This can be done by setting `vectorize` keyword argument to be true. For example, in the following code block, we solve the Von Neumann equation with TRBDF2 method:

```
sol_bdf = solve_von_neumann(annealing, tf, alg=TRBDF2(), reltol=1e-6, vectorize=true)
sol_bdf(0.5 * tf)
```

```
4-element Array{Complex{Float64},1}:
  0.4999999999999999 + 0.0im
 -0.4214558175232074 + 0.26894060674187487im
 -0.4214558175232074 - 0.26894060674187487im
  0.4999999999999999 + 0.0im
```

As shown above, the solution given by the solver becomes a vectorized version of the density matrix.

Side note: `TRBDF2` can actually work without vectorizing the Von Neumann equation. However, this is not generally true for other algorithms/solvers. For example, we will run into error for `LinearExponential` method if no vectorization is performed:

```
sol_linexp = solve_von_neumann(annealing, tf, alg=LinearExponential(),
tstops=range(0,tf,length=100));
sol_linexp = solve_von_neumann(annealing, tf, alg=LinearExponential(),
tstops=range(0,tf,length=100), vectorize=true);
```

```
Error: MethodError: no method matching Array{T,2} where T(::QuantumAnnealin
gTools.var"#34#38")
Closest candidates are:
  Array{T,2} where T(!Matched::LinearAlgebra.SymTridiagonal{T,V} where V<:A
bstractArray{T,1}) where T at D:\buildbot\worker\package_win64\build\usr\sh
are\julia\stdlib\v1.5\LinearAlgebra\src\tridiag.jl:141
  Array{T,2} where T(!Matched::LinearAlgebra.Tridiagonal{T,V} where V<:Abst
ractArray{T,1}) where T at D:\buildbot\worker\package_win64\build\usr\share
\julia\stdlib\v1.5\LinearAlgebra\src\tridiag.jl:582
  Array{T,2} where T(!Matched::LinearAlgebra.LowerTriangular{T,S} where S<:
AbstractArray{T,2}) where T at D:\buildbot\worker\package_win64\build\usr\s
hare\julia\stdlib\v1.5\LinearAlgebra\src\triangular.jl:34
  ...
```

We can again plot the $\langle X \rangle$ for different methods

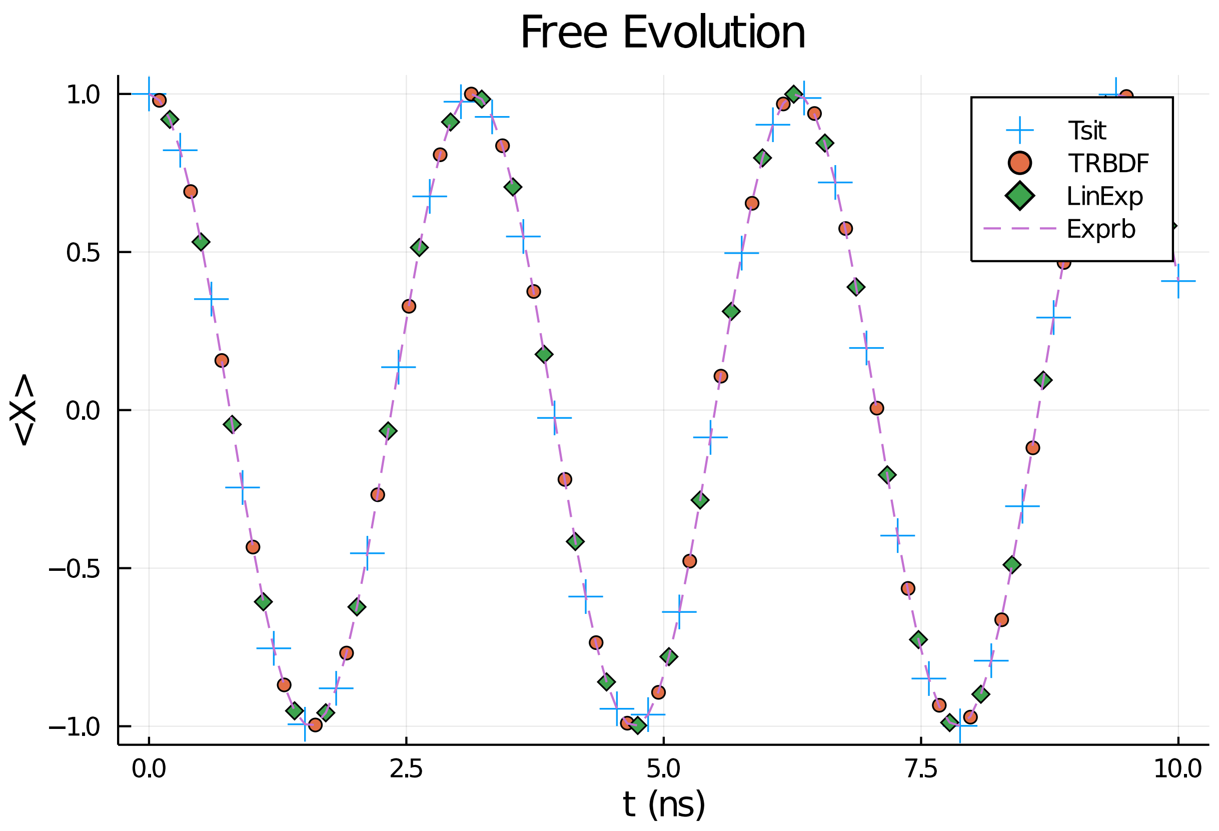
```
sol_tsit = solve_von_neumann(annealing, tf, alg=Tsit5(), reltol=1e-6);
sol_trbdf = solve_von_neumann(annealing, tf, alg=TRBDF2(), reltol=1e-6);

sol_linexp = solve_von_neumann(annealing, tf, alg=LinearExponential(),
tstops=range(0,tf,length=100), vectorize=true);

sol_exprb32 = solve_von_neumann(annealing, tf, alg=Exprb32(),
tstops=range(0,tf,length=100), vectorize=true);

t_list = range(0,tf,length=100)
tsit = []
trbdf = []
```

```
linexp = []
exprb32 = []
for s in t_list
    push!(tsit, real(tr(sol_tsit(s)*σx)))
    push!(trbdf, real(tr(sol_trbdf(s)*σx)))
    push!(linexp, real(tr(σx*reshape(sol_linexp(s),2,2))))
    push!(exprb32, real(tr(σx*reshape(sol_exprb32(s),2,2))))
end
scatter(t_list[1:3:end], tsit[1:3:end], label="Tsit", marker=:+, markersize=8)
scatter!(t_list[2:3:end], trbdf[2:3:end], label="TRBDF")
scatter!(t_list[3:3:end], linexp[3:3:end], label="LinExp", marker=:d)
plot!(t_list, exprb32, label="Exprb", linestyle=:dash)
xlabel!("t (ns)")
ylabel!("<X>")
title!("Free Evolution")
```



Unitary Lastly, we can also solve the unitary

$$U(s) = T_+ \exp\left\{ -it_f \int_0^s H(s')\mathrm{d}s' \right\}$$

using `solve_unitary`. The ODE form of the problem is

$$\dot{U} = -it_f H(s) U \ .$$

Again, although this is in principle equivalent to Schrondinger/Von Neumann equation, the unitary becomes handy in certain cases, e.g. when solving the Redfeild equation.

```
annealing = Annealing(H, u0)
sol_tsit = solve_unitary(annealing, tf, alg=Tsit5(),abstol=1e-6, reltol=1e-6)
```

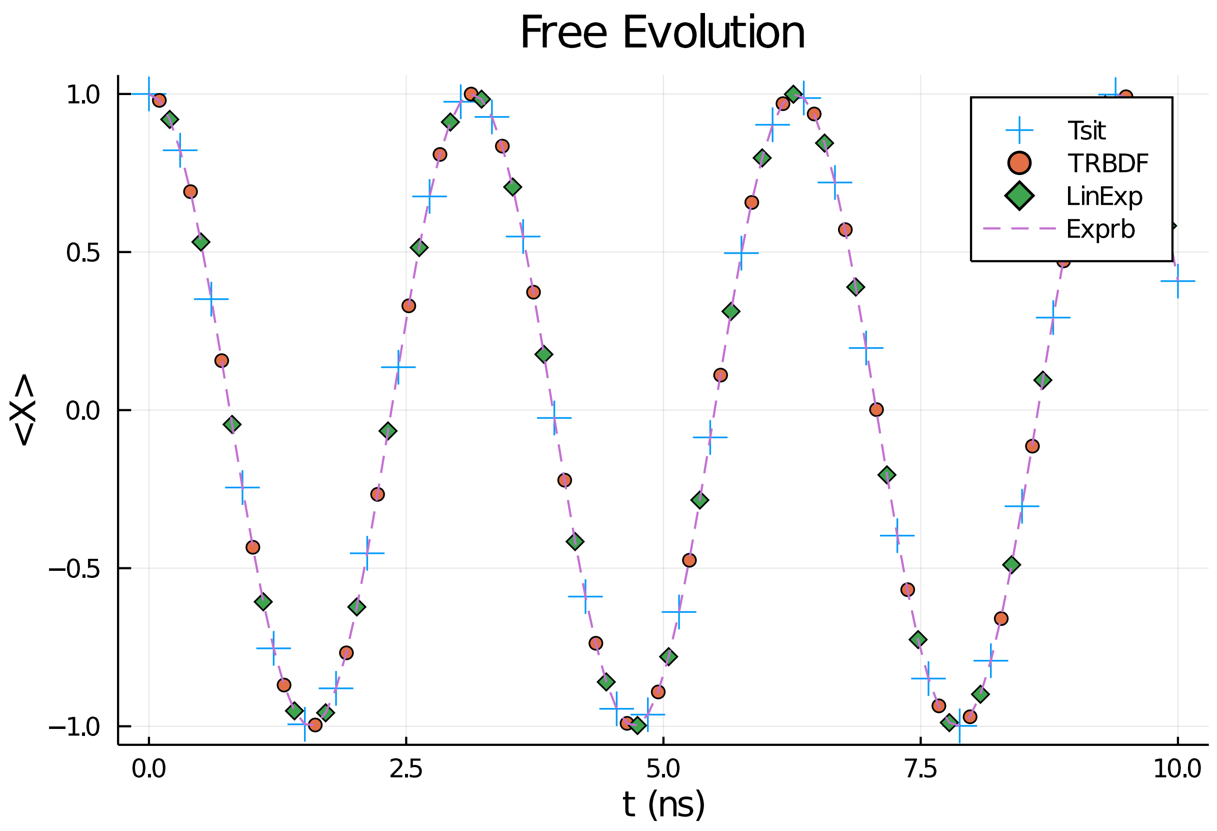
```
sol_tsit(0.5 * tf)
```

```
2×@*(2 Array(*@{Complex{Float64},2}:
 0.283662-0.958924im       0.0+0.0im
      0.0+0.0im       0.283662+0.958924im
```

Again we plot the $\langle X \rangle$ obtained by multiplying the unitary with the initial state.

```
sol_tsit = solve_unitary(annealing, tf, alg=Tsit5(), reltol=1e-6);
sol_trbdf = solve_unitary(annealing, tf, alg=TRBDF2(), reltol=1e-6, vectorize=true);
# LinearExponential is a fixed step size method, user need to specify the time steps
using keyword argument `tstops`.
sol_linexp = solve_unitary(annealing, tf, alg=LinearExponential(),
tstops=range(0,tf,length=100), vectorize=true);
# Even though Exprb method is an adaptive method, it tends to jump a lot of middle
points. So if you want accurate evolution in the middle,
# it is better to manually add more points for the algorithm.
sol_exprb32 = solve_unitary(annealing, tf, alg=Exprb32(), tstops=range(0,tf,length=100),
vectorize=true);

t_list = range(0,tf,length=100)
tsit = []
trbdf = []
linexp = []
exprb32 = []
for s in t_list
    state_tsit = sol_tsit(s) * u0
    state_trbdf = reshape(sol_trbdf(s), 2, 2) * u0
    state_linexp = reshape(sol_linexp(s), 2, 2) * u0
    state_exprb32 = reshape(sol_exprb32(s), 2, 2) * u0
    push!(tsit, real(state_tsit' * σx * state_tsit))
    push!(trbdf, real(state_trbdf' * σx * state_trbdf))
    push!(linexp, real(state_linexp' * σx * state_linexp))
    push!(exprb32, real(state_exprb32' * σx * state_exprb32))
end
scatter(t_list[1:3:end], tsit[1:3:end], label="Tsit", marker=:+, markersize=8)
scatter!(t_list[2:3:end], trbdf[2:3:end], label="TRBDF")
scatter!(t_list[3:3:end], linexp[3:3:end], label="LinExp", marker=:d)
plot!(t_list, exprb32, label="Exprb", linestyle=:dash)
xlabel!("t (ns)")
ylabel!("<X>")
title!("Free Evolution")
```


Free Evolution
<X>
t (ns)
1.0
0.5
0.0
−0.5
−1.0
0.0
2.5
5.0
7.5
10.0
Tsit
TRBDF
LinExp
Exprb