An Intro to HOQST - closed-system simulation

Huo Chen

April 18, 2021

0.1 Closed-system Examples

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

0.1.1 Define the Hamiltonian

The first step is to define a 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 that the simulation results can be trivially confirmed. The syntax is the same for time-dependent Hamiltonians. Let's first define the Hamiltonian by:

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

```
DenseHamiltonian with ComplexF64
with size: (2, 2)
```

In this example, we use the `DenseHamiltonian` type. It means that the underlying data structure of this Hamiltonian type is Julia array. There exists a different Hamiltonian type named `SparseHamiltonian` that relies on sparse array as its internal data structure. Sparsity can only provide performance improvement when the system size is large. So, as a rule of thumb, users should only consider using `SparseHamiltonian` when the system size is larger than 10 qubits.

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 is 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 OpenQuantumBase.DenseHamiltonian{ComplexF64} and u0 Vector{C
omplexF64}
u0 size: (2,)
```

The initial state in the 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 this differential equation, we need to choose a proper algorithm. HOQST relies on `OrdinaryDiffEq.jl` as the low-level solver, which supports 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 recommend a list of algorithms we tested as follows:

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 is a second-order method, we recommended using a smaller error tolerance than with other higher-order methods.

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

   This method discretizes the Hamiltonian and does the matrix exponential for each interval.

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

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

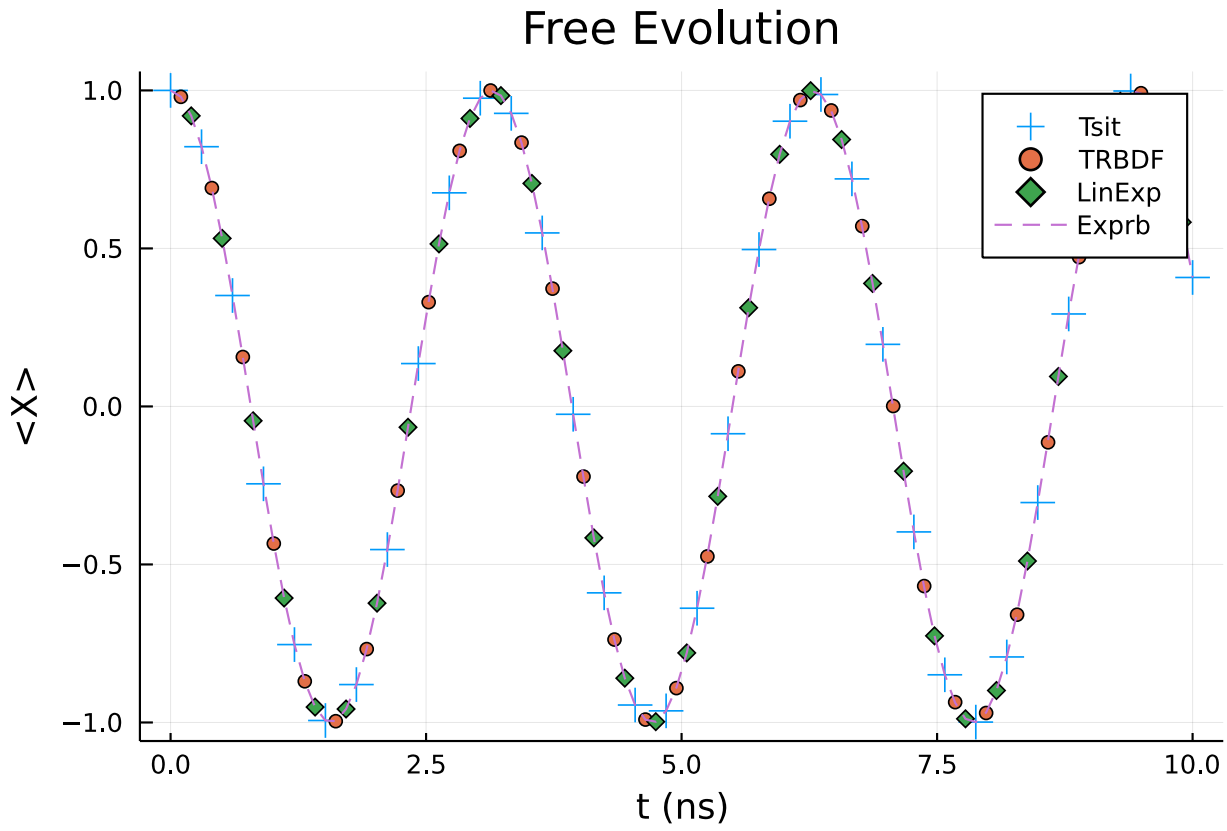
It is important to note that methods 3 and 4 are exponential methods that are supposed to preserve the state vectors' norm. To solve the Schrodinger equation, we use the function `solve_schrodinger`:

```
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-stepsize method, the user needs to specify the time steps
using the 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 is an adaptive method, it tends to skip a lot of middle points. So
if you want an accurate solution in the middle,
# it is better to manually add more points.
sol_exprb32 = solve_schrodinger(annealing, tf, alg=Exprb32(),
tstops=range(0,tf,length=100));
```

In the above code block, the keyword arguments `abstol` and `reltol` are the absolute and relative error tolerances for stepsize control in adaptive stepsize ODE algorithms. They are usually chosen by trial-and-error in real applications.

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

```julia
# this code block shows how to plot the expectation value of X
t_list = range(0,tf,length=100)
tsit = []
trbdf = []
linexp = []
exprb32 = []
for s in t_list
    # sol_tsit(s)'*σx*sol_tsit(s) calculates the
    # expectation value <ψ(s)|X|ψ(s)>
    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 closed-system equations

The package also contains several other closed-system solvers.

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

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

Even though the Von Neumann equation is equivalent to the Schrodinger equation, it is sometimes numerically more stable than the Schrodinger equation. Users are 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 Vector{Float64}:
  0.0
  0.025416471135423512
  0.07162657822452011
  0.12691325854682373
  0.19490751014915958
  0.2726355853793798
  0.3605272018981224
  0.4567317871059021
  0.5605609353632346
  0.6707029681719032
  ⋮
  8.93912030993445
  9.07904338542158
  9.218966473882144
  9.358889575316146
  9.498812680456844
  9.63873579857098
  9.778658911125072
  9.91858201811912
 10.0
u: 78-element Vector{Matrix{ComplexF64}}:
 [0.4999999999999999 + 0.0im 0.4999999999999999 + 0.0im; 0.4999999999999999
 + 0.0im 0.4999999999999999 + 0.0im]
 [0.4999999999999999 + 0.0im 0.49935414208671036 + 0.025405526573483335im;
0.49935414208671036 - 0.025405526573483335im 0.4999999999999999 + 0.0im]
 [0.4999999999999999 + 0.0im 0.4948784008324741 + 0.07138184906644161im; 0.
4948784008324741 - 0.07138184906644161im 0.4999999999999999 + 0.0im]
 [0.4999999999999999 + 0.0im 0.48397931745784584 + 0.12555484945742712im; 0
.48397931745784584 - 0.12555484945742712im 0.4999999999999999 + 0.0im]
 [0.4999999999999999 + 0.0im 0.4624896853719465 + 0.19000865966633518im; 0.
4624896853719465 - 0.19000865966633518im 0.4999999999999999 + 0.0im]
 [0.4999999999999999 + 0.0im 0.42749333916749394 + 0.2593249785858282im; 0.
42749333916749394 - 0.2593249785858282im 0.4999999999999999 + 0.0im]
 [0.4999999999999999 + 0.0im 0.37555502540351415 + 0.3300885061538325im; 0.
37555502540351415 - 0.3300885061538325im 0.4999999999999999 + 0.0im]
 [0.4999999999999999 + 0.0im 0.30550378150848995 + 0.3958123767243214im; 0.
30550378150848995 - 0.3958123767243214im 0.4999999999999999 + 0.0im]
 [0.4999999999999999 + 0.0im 0.2173361827963109 + 0.45029432633612176im; 0.
2173361827963109 - 0.45029432633612176im 0.4999999999999999 + 0.0im]
 [0.4999999999999999 + 0.0im 0.11369195427658926 + 0.4869025921872921im; 0.
11369195427658926 - 0.4869025921872921im 0.4999999999999999 + 0.0im]
 ⋮
 [0.4999999999999999 + 0.0im 0.28210754285167106 - 0.41281345495257504im; 0
.28210754285167106 + 0.41281345495257504im 0.4999999999999999 + 0.0im]
 [0.4999999999999999 + 0.0im 0.38515527546908473 - 0.3188338459667724im; 0.
```

```
38515527546908473 + 0.3188338459667724im 0.4999999999999999 + 0.0im]
 [0.4999999999999999 + 0.0im 0.458236339965551 - 0.2000476381328814im; 0.45
8236339965551 + 0.2000476381328814im 0.4999999999999999 + 0.0im]
 [0.4999999999999999 + 0.0im 0.49566471856964134 - 0.06569688565511958im; 0
.49566471856964134 + 0.06569688565511958im 0.4999999999999999 + 0.0im]
 [0.4999999999999999 + 0.0im 0.4945283199777981 + 0.0737653576645211im; 0.4
945283199777981 - 0.0737653576645211im 0.4999999999999999 + 0.0im]
 [0.4999999999999999 + 0.0im 0.4549155549065743 + 0.20748835723339634im; 0.
4549155549065743 - 0.20748835723339634im 0.4999999999999999 + 0.0im]
 [0.4999999999999999 + 0.0im 0.3799084648348185 + 0.325067888389725im; 0.37
99084648348185 - 0.325067888389725im 0.4999999999999999 + 0.0im]
 [0.4999999999999999 + 0.0im 0.2753429188830743 + 0.41735577624873776im; 0.
2753429188830743 - 0.41735577624873776im 0.4999999999999999 + 0.0im]
 [0.4999999999999999 + 0.0im 0.20403994993813346 + 0.4564726338136435im; 0.
20403994993813346 - 0.4564726338136435im 0.4999999999999999 + 0.0im]
```

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

```
sol_tsit(0.5)
```

```
2×2 Matrix{ComplexF64}:
      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 the `vectorize` keyword argument to be true. For example, in the following code block, we solve the Von Neumann equation with the TRBDF2 method:

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

```
4-element Vector{ComplexF64}:
   0.4999999999999999 + 0.0im
 -0.42145581752320743 + 0.2689406067418741im
 -0.42145581752320743 - 0.2689406067418741im
   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 errors 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 (Matrix{T} where T)(::OpenQuantumToo
ls.var"#34#38")
Closest candidates are:
  (Matrix{T} where T)(!Matched::Union{LinearAlgebra.QR, LinearAlgebra.QRCom
pactWY}) at C:\buildbot\worker\package_win64\build\usr\share\julia\stdlib\v
1.6\LinearAlgebra\src\qr.jl:400
  (Matrix{T} where T)(!Matched::LinearAlgebra.Hessenberg) at C:\buildbot\wo
```

```
rker\package_win64\build\usr\share\julia\stdlib\v1.6\LinearAlgebra\src\hess
enberg.jl:470
  (Matrix{T} where T)(!Matched::LinearAlgebra.LQ) at C:\buildbot\worker\pac
kage_win64\build\usr\share\julia\stdlib\v1.6\LinearAlgebra\src\lq.jl:120
  ...
```

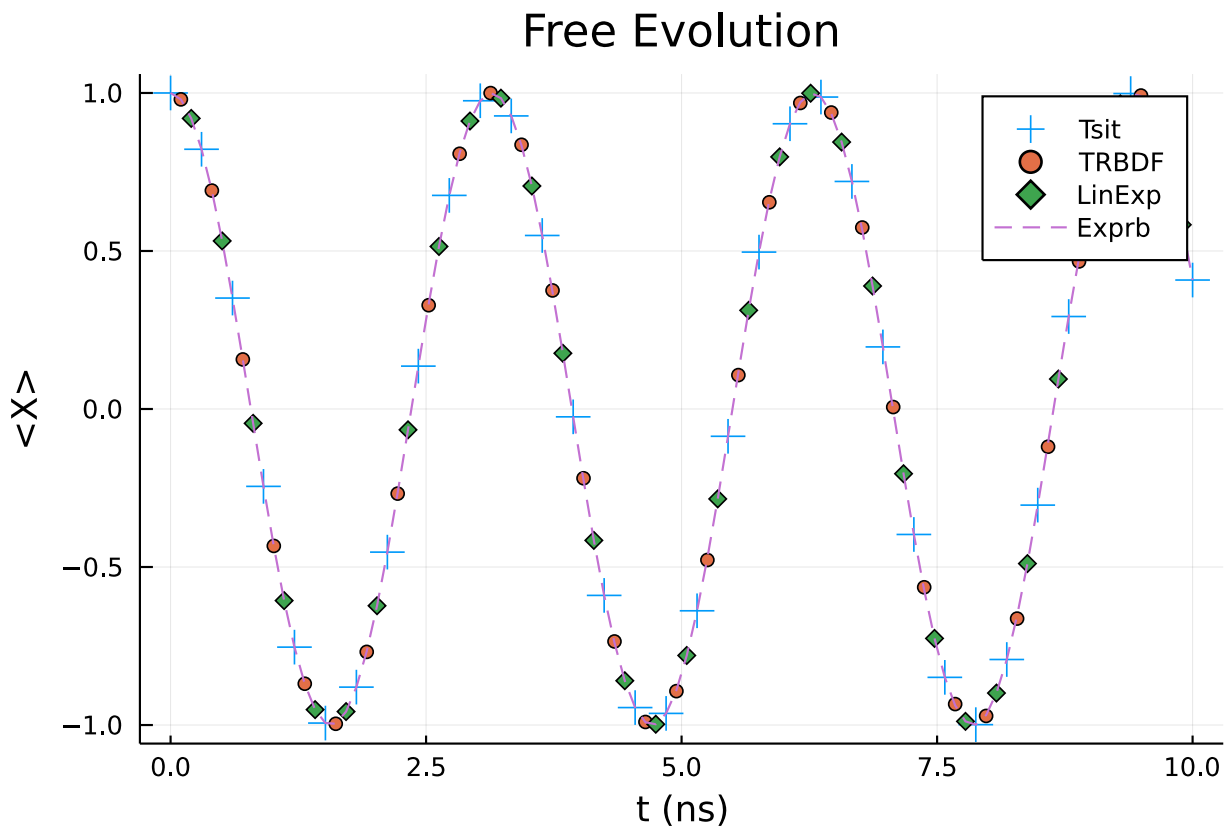
We can again plot the $\langle X \rangle$ for different methods:

```
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

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

Although this is in principle equivalent to the Schrodinger/Von Neumann equation, the unitary becomes handy in certain cases, e.g., when solving the Redfield equation.

```
annealing = Annealing(H, u0)
sol_tsit = solve_unitary(annealing, tf, alg=Tsit5(),abstol=1e-6, reltol=1e-6)
sol_tsit(0.5 * tf)
```

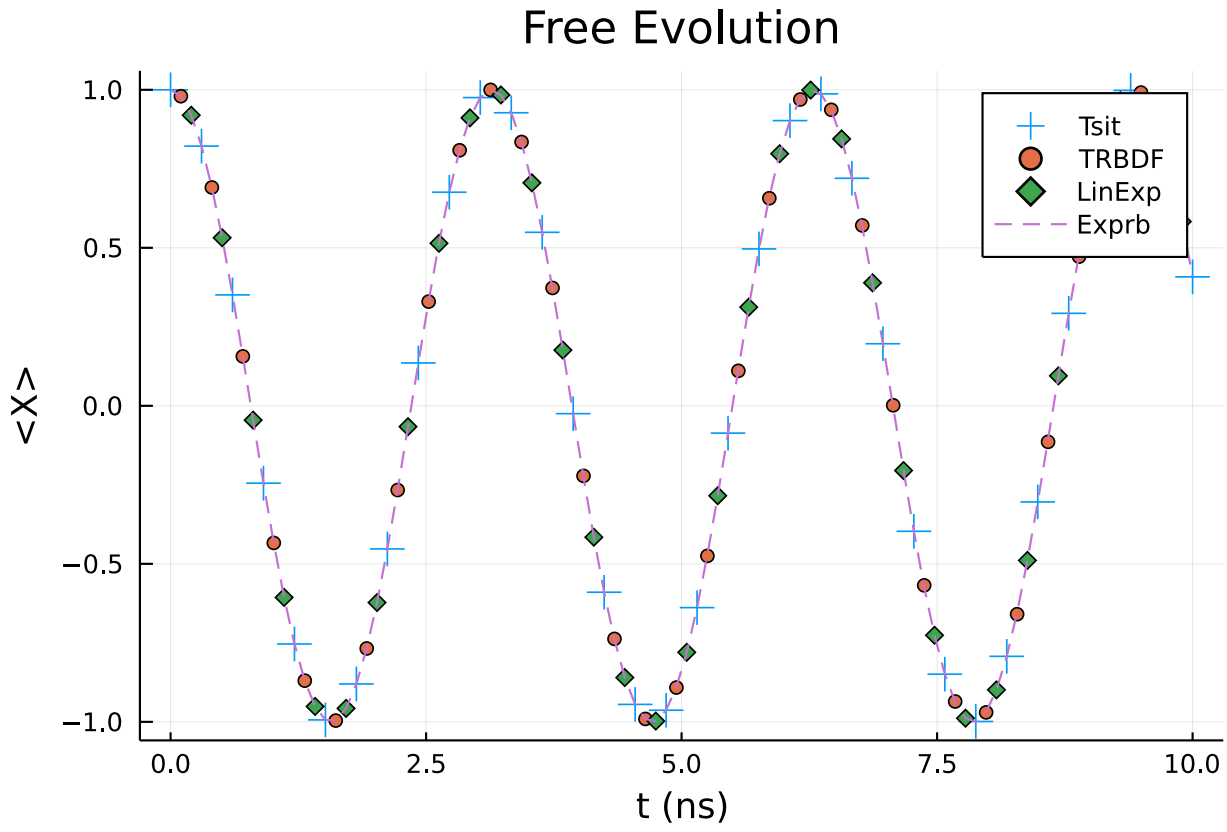
```
2×2 Matrix{ComplexF64}:
 0.283662-0.958924im       0.0+0.0im
      0.0+0.0im       0.283662+0.958924im
```

Again we plot the $\langle X \rangle$ obtained by multiplying the unitary by 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, users need to specify the time steps
using the 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 skip a lot of middle
points. So if you want an 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")
```



0.2 Appendix

This tutorial is part of the HOQSTTutorials.jl repository, found at: https://github.com/USCqserver/HOQ

To locally run this tutorial, do the following commands:

```
using HOQSTTutorials
HOQSTTutorials.weave_file("introduction","01-closed_system.jmd")
```

Computer Information:

```
Julia Version 1.6.0
Commit f9720dc2eb (2021-03-24 12:55 UTC)
Platform Info:
  OS: Windows (x86_64-w64-mingw32)
  CPU: Intel(R) Core(TM) i7-6700K CPU @ 4.00GHz
  WORD_SIZE: 64
  LIBM: libopenlibm
  LLVM: libLLVM-11.0.1 (ORCJIT, skylake)
```

Package Information:

```
Status `tutorials\introduction\Project.toml`
[2913bbd2-ae8a-5f71-8c99-4fb6c76f3a91] StatsBase 0.33.4
[1dea7af3-3e70-54e6-95c3-0bf5283fa5ed] OrdinaryDiffEq 5.52.2
[e429f160-8886-11e9-20cb-0dbe84e78965] OpenQuantumTools 0.6.2
[91a5bcdd-55d7-5caf-9e0b-520d859cae80] Plots 1.11.2
[b964fa9f-0449-5b57-a5c2-d3ea65f4040f] LaTeXStrings 1.2.1
[1fd47b50-473d-5c70-9696-f719f8f3bcdc] QuadGK 2.4.1
```