An Intro to HOQST - adiabatic master equation

Huo Chen

November 21, 2020

0.1 Single qubit annealing

This tutorial will recreate the single-qubit example in this paper: [1] Decoherence in adiabatic quantum computation.

The Hamiltonian of this example is

$$H(s) = -\frac{1}{2}(1-s)\sigma_x - \frac{1}{2}s\sigma_z \ ,$$

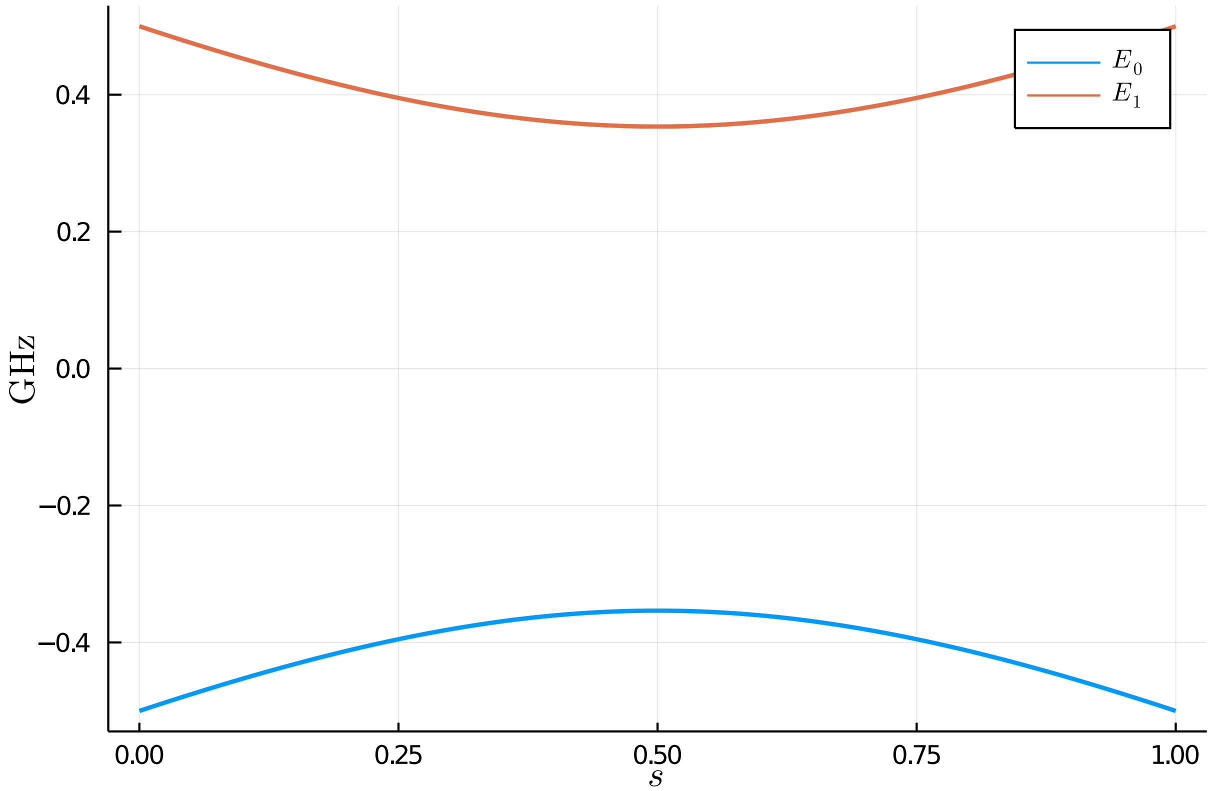
which can be constructed by the following code block

```
using OpenQuantumTools, OrdinaryDiffEq, Plots
H = DenseHamiltonian([(s)->1-s, (s)->s], -[σx, σz]/2)
```

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

This package directly interacts with Plots.jl by defining recipes. We can visualize the spectrum of the Hamiltonian by directly plotting the object:

```
# this plot recipe is for conveniently plotting the spectrum of the Hamiltonian
# the first 3 arguments are: the Hamiltonian, the grid `s` and the levels to keep
plot(H, 0:0.01:1, 2, linewidth=2)
```



0.1.1 Unit (h or $\hbar$)

A keyword argument `unit` whose default value is `:h` can be provided to any Hamiltonian type's constructor. This argument specifies the unit of other input arguments. For example, setting `unit` to `:h` means the other input arguments have the unit of GHz, while setting it to `:ħ` means the other input arguments have the unit of 2πGHz. To evaluate the Hamiltonian at a given time, the user should use the `evaluate` function instead of directly calling it. `evaluate` will always return the Hamiltonian value in the unit system of $h = 1$. The following code block shows the effects of different choices of `unit`:

```
H_h = DenseHamiltonian([(s)->1-s, (s)->s], -[σx, σz]/2, unit=:h)
H_ħ = DenseHamiltonian([(s)->1-s, (s)->s], -[σx, σz]/2, unit=:ħ)
println("Setting unit to :h")
@show evaluate(H_h, 0.5)
println("Setting unit to :ħ")
@show evaluate(H_ħ, 0.5);
```

```
Setting unit to :h
evaluate(H_h, 0.5) = Complex{Float64}[-0.25 + 0.0im -0.25 + 0.0im; -0.25 +
0.0im 0.25 + 0.0im]
Setting unit to :ħ@*(evaluate(H(*@_ħ@*(, 0.5) =
Complex(*@{Float64}[-0.039788735772973836 + 0.0im -0.0397
88735772973836 + 0.0im; -0.039788735772973836 + 0.0im 0.039788735772973836
+ 0.0im]
```

Internally, HOQST uses a unit system of $\hbar = 1$. If we call `H_h` directly, its value is scaled by 2π:

```
H_h(0.5)
```

```
2×@*(2 StaticArrays.MArray(*@{Tuple{2,2},Complex{Float64},2,4} with indices SOneT
o(2)×@*(SOneTo(2):-1.5708+0.0im -1.5708+0.0im-1.5708+0.0im 1.5708+0.0im
```

0.1.2 Annealing

The total Hamiltonian presented in [1] is

$$H(s) = H_{\mathrm{S}}(s) + gS \otimes B + H_{\mathrm{B}} \ .$$

We denote S the coupling and $\{gB, H_{\mathrm{B}}\}$ the bath.

Coupling For constant coupling operators, we can use the constructor `ConstantCouplings`. Like Hamiltonian's case, there is a keyword argument `unit` to specify the input unit.

```
coupling = ConstantCouplings(["Z"])
```

```
ConstantCouplings with AbstractArray{T,2} where T
and string representation: ["Z"]
```

Bath A bath instance can be any object which implements the following three methods:

1. Correlation function: `correlation(τ, bath)`
2. Spectral density: `γ(ω, bath)`
3. Lamb shift: `S(ω, bath)`

Redfield/Adiabatic ME solvers require those three methods. Currently, we have built-in support for the Ohmic bath. An Ohmic bath object can be created by :

```
η = 1e-4
fc = 4
T = 16
bath = Ohmic(η, fc, T)
```

```
Ohmic bath instance:
η@*( (unitless): 0.0001(*@ω@*(c (GHz): 4.0T (mK): 16.0
```

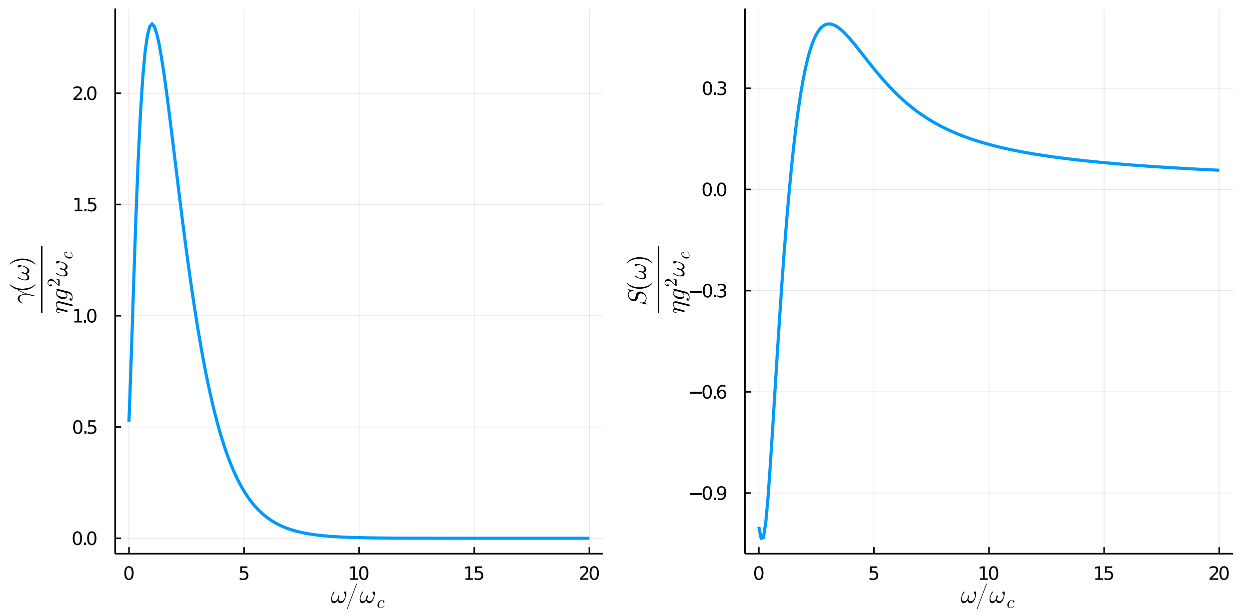
`info_freq` is a convenient function to convert each quantity into the same unit.

```
info_freq(bath)
```

```
ω@*(c (GHz): 4.0T (GHz): 0.33338579560200365
```

We can also directly plot the spectral density of Ohmic bath:

```
p1 = plot(bath, :γ, range(0,20,length=200), label="", size=(800, 400), linewidth=2)
p2 = plot(bath, :S, range(0,20,length=200), label="", size=(800, 400), linewidth=2)
plot(p1, p2, layout=(1,2), left_margin=3Plots.Measures.mm)
```



Annealing object Finally, we can assemble the annealing object by

```
# Hamiltonian
H = DenseHamiltonian([(s)->1-s, (s)->s], -[σx, σz]/2, unit=:ħ)
# initial state
u0 = PauliVec[1][1]
# coupling
coupling = ConstantCouplings(["Z"], unit=:ħ)
# bath
bath = Ohmic(1e-4, 4, 16)
annealing = Annealing(H, u0; coupling=coupling, bath=bath)
```

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

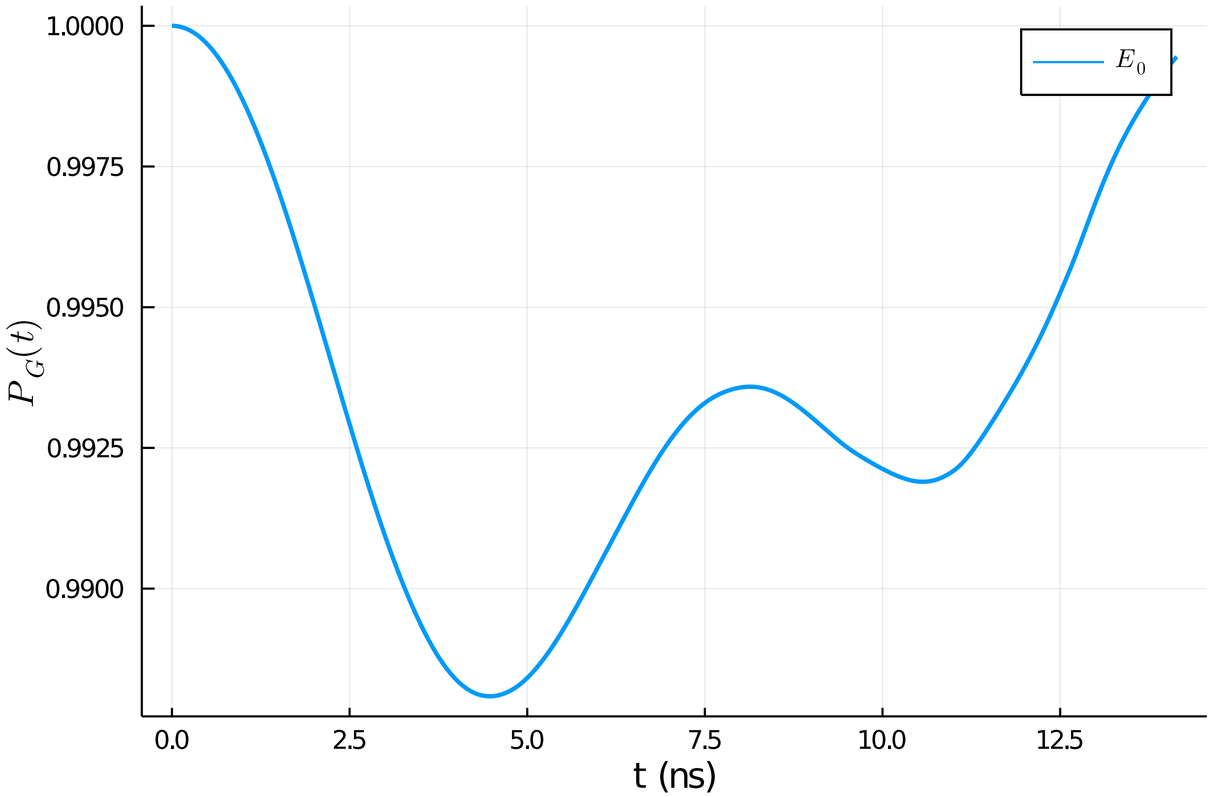
Because we want to compare our results to the [1], we need to set the unit to $\hbar = 1$.

0.1.3 Closed system

There are several interfaces in HOQST that might be handy. The first one is the Schrodinger equation solver:

```
tf = 10*sqrt(2)
@time sol = solve_schrodinger(annealing, tf, alg=Tsit5(), retol=1e-4)
# The following line of code is a convenient recipe to plot the instantaneous population
during the evolution.
# It currently only supports Hamiltonian with annealing parameter s = t/tf from 0 to 1.
# The third argument can be either a list or a number. When it is a list, it specifies
the energy levels to plot (starting from 0); when it is a number, it specifies the total
number of levels to plot.
plot(sol, H, [0], 0:0.01:tf, linewidth=2, xlabel = "t (ns)", ylabel="\$P_G(t)\$")
```

```
0.003923 seconds (2.42 k allocations: 147.051 KiB)
```



The solution is an `ODESolution` object in `DifferentialEquations.jl` package. More details for the interface can be found here. The state vector's value at a given time can be obtained by directly calling the `ODESolution` object.

```
sol(0.5)
```

```
2-element Array{Complex{Float64},1}:
 0.6856253144209079 + 0.1750041214939618im
 0.6861430138705714 + 0.1688172359560306im
```

Other interfaces include

```
# You need to solve the unitary first before trying to solve Redfield equation
@time U = solve_unitary(annealing, tf, alg=Tsit5(), abstol=1e-8, retol=1e-8);
@time solve_von_neumann(annealing, tf, alg=Tsit5(), abstol=1e-8, retol=1e-8);
```

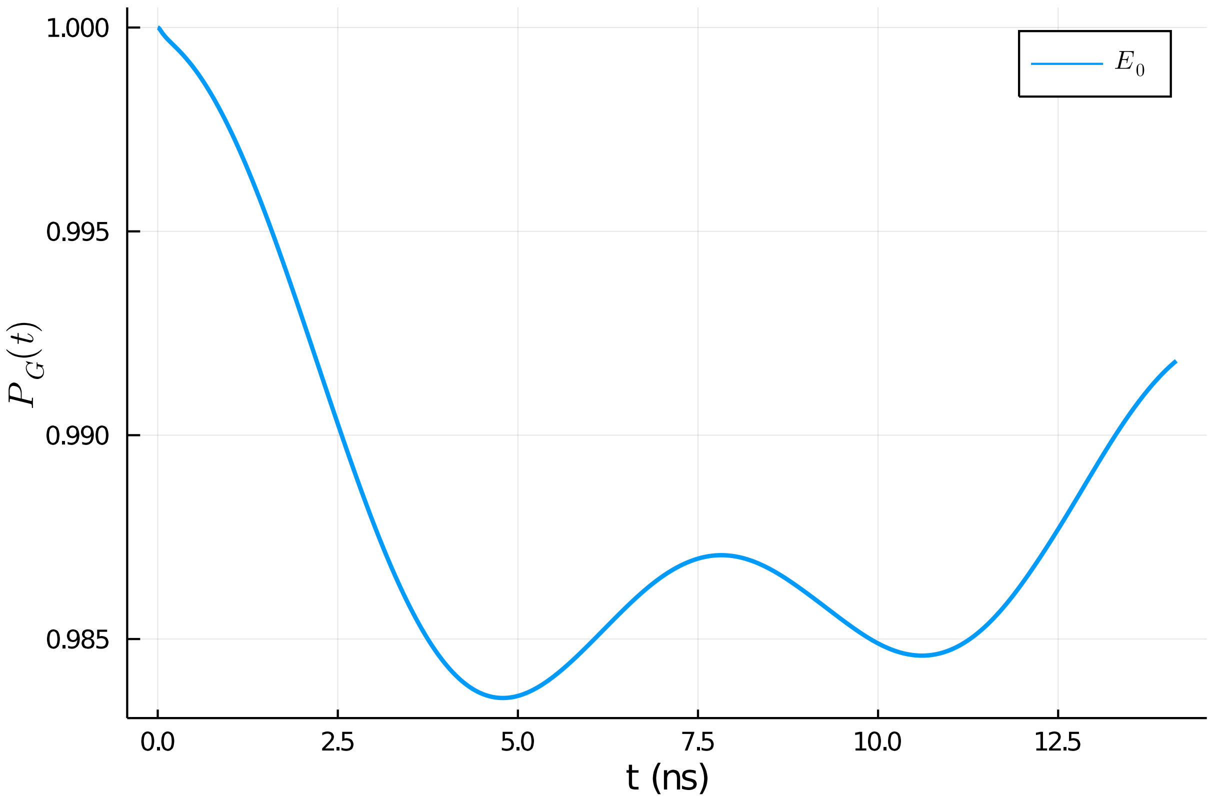
0.1.4 Open System

Time-dependent Redfield equation The time-dependent Redfield equation solver needs

1. Annealing object
2. Total annealing time
3. Pre-calculated unitary

to work. The following code block illustrates how to supply the above three objects to the Redfield solver. In addition all the other keyword arguments in DifferentialEquations.jl are supported.

```
tf = 10*sqrt(2)
U = solve_unitary(annealing, tf, alg=Tsit5(), abstol=1e-8, retol=1e-8);
sol = solve_redfield(annealing, tf, U; alg=Tsit5(), abstol=1e-8, retol=1e-8)
plot(sol, H, [0], 0:0.01:tf, linewidth=2, xlabel="t (ns)", ylabel="\$P_G(t)\$")
```



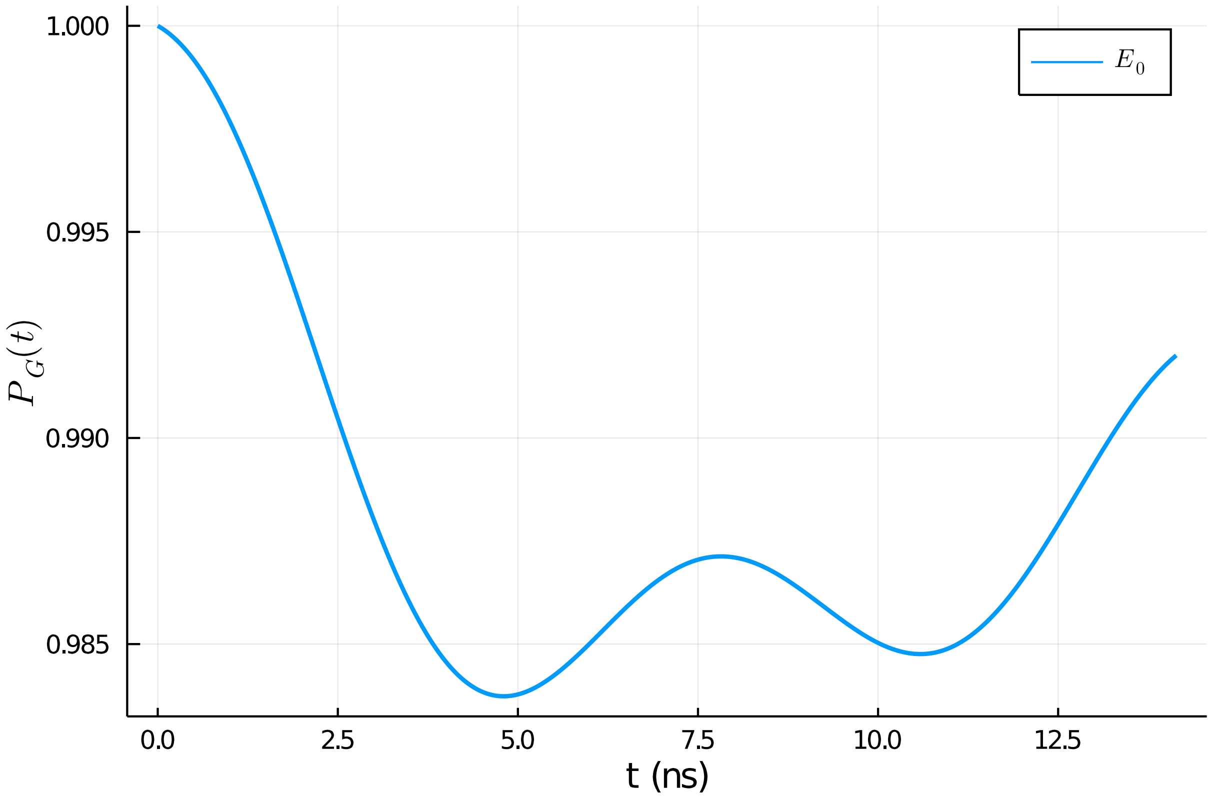
Adiabatic master equation The adiabatic master equation solver needs

1. Annealing object
2. Total Annealing time

Besides other keyword arguments supported in DifferentialEquations.jl, it is highly recommended to add the ω`_hint` keyword argument. By doing this, the solver will pre-compute the quantity $S(\omega)$ in Lambshift within the range specified by ω`_hint` to speed up the computation.

```
tf = 10*sqrt(2)
@time sol = solve_ame(annealing, tf; alg=Tsit5(), ω_hint=range(-6, 6, length=100),
reltol=1e-4)
plot(sol, H, [0], 0:0.01:tf, linewidth=2, xlabel="t (ns)", ylabel="\$P_G(t)\$")
```

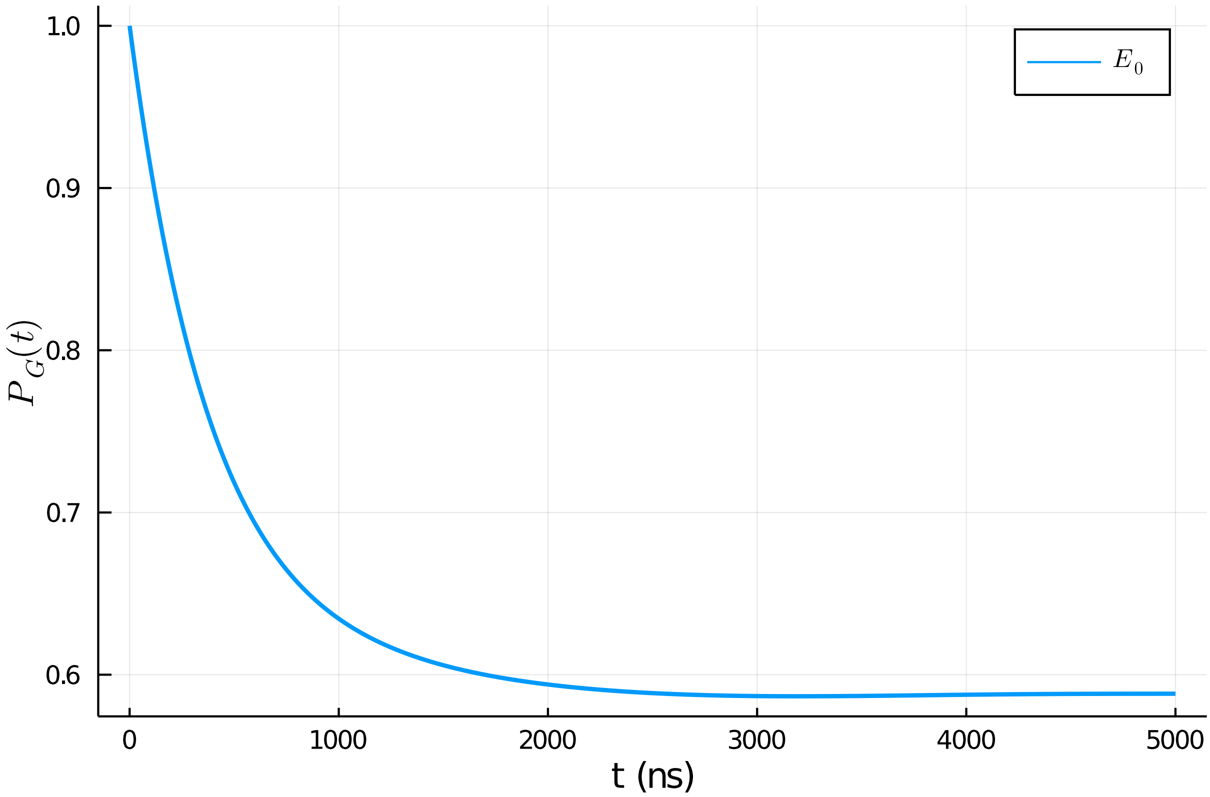
```
0.010433 seconds (193.16 k allocations: 4.270 MiB)
```



We can also solve the AME for a longer annealing time:

```
tf = 5000
@time sol_ame = solve_ame(annealing, tf; alg=Tsit5(), ω_hint=range(-6, 6, length=100),
reltol=1e-6)
plot(sol_ame, H, [0], 0:1:tf, linewidth=2, xlabel="t (ns)", ylabel="\$P_G(t)\$")
```

```
0.093767 seconds (1.37 M allocations: 70.515 MiB)
```



The above results agree with Fig 2 of reference [1].

Trajectory method for adiabatic master equation The package also supports the trajectory method for AME. More details of this method can be found in [2] Quantum trajectories for time-dependent adiabatic master equations. The basic workflow is to create an ODE EnsembleProblem via `build_ensembles` interface. Then, the resulting `EnsembleProblem` object can be solved by the native Parallel Ensemble Simulations interface of `DifferentialEquations.jl`. The following code block solves the same annealing dynamics described above ($t_f = 5000(ns)$) using multithreading. To keep the running time reasonably short, we simulate only 3000 trajectories in this example. The result may not converge to the true solution yet. The user is encouraged to try more trajectories and see how the result converges.

The codes can also be deployed on high-performance clusters using Julia's native distributed computing module.

```
tf = 5000
# total number of trajectories
num_trajectories = 3000
# construct the `EnsembleProblem`
# `safetycopy` needs to be true because the current trajectories implementation is not
thread-safe.
prob = build_ensembles(annealing, tf, :ame, ω_hint=range(-6, 6, length=200),
safetycopy=true)
# to use multi-threads, you need to start Julia kernel with multiple threads
# julia --threads 8
sol = solve(prob, Tsit5(), EnsembleThreads(), trajectories=num_trajectories,
reltol=1e-6, saveat=range(0,tf,length=100))

t_axis = range(0,tf,length=100)
dataset = []
```

```julia
for t in t_axis
    w, v = eigen_decomp(H, t/tf)
    push!(dataset, [abs2(normalize(so(t))' * v[:, 1]) for so in sol])
end

# the following codes calculate the instantaneous ground state population and its error
bar by averaging over all the trajectories

pop_mean = []
pop_sem = []
for data in dataset
    p_mean = sum(data) / num_trajectories
    p_sem = sqrt(sum((x)->(x-p_mean)^2, data)) / num_trajectories
    push!(pop_mean, p_mean)
    push!(pop_sem, p_sem)
end

scatter(t_axis, pop_mean, marker=:d, yerror=2*pop_sem, label="Trajectory", markersize=6)
plot!(sol_ame, H, [0], t_axis, linewidth=2, label="Non-trajectory")
xlabel!("t (ns)")
ylabel!("\$P_G(s)\$")
```