A Tutorial on Quantum Dynamics Simulations on Quantum Computers. Part III: The Generalized Quantum Master Equation

Xiaohan Dan,[†] Pouya Khazaei,[‡] Ningyi Lyu,[†] Ellen Mulvihill,[†] Yuchen Wang,[¶] Saurabh Baswaraj,[¶] Sabre Kais,[¶] Victor S. Batista,[†,§] and Eitan Geva[*,‡]

[†]*Department of Chemistry, Yale University, New Haven, CT 06520, USA*

[‡]*Department of Chemistry, University of Michigan, Ann Arbor, MI 48109, USA*

[¶]*Department of Chemistry, Department of Physics and Purdue Quantum Science and Engineering Institute, Purdue University, West Lafayette, IN 47907, USA*

[§]*Yale Quantum Institute, Yale University, New Haven, CT 06511, USA*

E-mail: ???@???

Contents

Abstract

Finite temperature simulations of quantum dynamics in complex chemical systems represent an outstanding and ongoing challenge. In practice, it is often advantageous to focus on the quantum dynamics of a quantity of interest, such as the reduced density matrix of a system coupled to a bath, which can be treated as an open quantum system. The Nakajima-Zwanzig generalized quantum master equation (GQME) provides a formally exact framework for simulating the non-Markovian quantum dynamics of such an open system. This article is Part III of a series of tutorials about simulating quantum dynamics on quantum computers. In this part, we focus on simulating the non-Markovian open quantum system dynamics described by a GQME. The quantum algorithm, which is based on the Sz.-Nagy dilation theorem, is demonstrated on the spin-boson model.

1 Introduction

We focus on molecular systems with an overall Hamiltonian of the following form, which is suitable and commonly used for modeling electronic energy and charge transfer reactions in chemical systems:

$$\hat{H} = \sum_{j=1}^{N_e} \hat{H}_j(\hat{\mathbf{R}}, \hat{\mathbf{P}}) \, |j\rangle\langle j| + \sum_{\substack{j,k=1 \\ k \neq j}}^{N_e} \hat{V}_{jk}(\hat{\mathbf{R}}) \, |j\rangle\langle k| \quad . \tag{1}$$

Here, $\hat{H}_j(\hat{\mathbf{R}}, \hat{\mathbf{P}}) = \hat{\mathbf{P}}^2/2 + V_j(\hat{\mathbf{R}})$ is the nuclear Hamiltonian when the system is in the *diabatic* electronic state $|j\rangle$, with the index j running over the N_e electronic states; $\{\hat{V}_{jk}(\hat{\mathbf{R}}) | j \neq k\}$ are coupling terms between electronic states; and $\hat{\mathbf{R}} = \{\hat{R}_1, \hat{R}_2, ..., \hat{R}_{N_n}\}$ and $\hat{\mathbf{P}} = \{\hat{P}_1, \hat{P}_2, ..., \hat{P}_{N_n}\}$ are the mass-weighted position and momentum operators of the N_n nuclear degrees of freedom (DOFs). Throughout this paper, boldfaced variables, e.g., $\mathbf{A}$, indicate vector quantities; a hat over a variable, e.g., $\hat{B}$, indicates an operator quantity; and calligraphic font, e.g., $\mathcal{L}$, indicates a superoperator.

Assuming that the overall system is closed, its dynamics can be described by the quantum Liouville equation[1,2]

$$\frac{\mathrm{d}}{\mathrm{d}t}\hat{\rho}(t) = -\frac{i}{\hbar}[\hat{H}, \hat{\rho}(t)] = -\frac{i}{\hbar}\mathcal{L}\hat{\rho}(t) \quad . \tag{2}$$

Here, $\hat{\rho}(t)$ is the density operator that describes the state of the overall system and $\mathcal{L}(\cdot) = [\hat{H}, \cdot]$ is the overall system Liuovillian superoperator with $\hat{H}$ as in Eq. (1). The computational cost of solving the quantum Liouville equation, Eq. (2), scales exponentially with the number of electronic and nuclear DOFs ($N_n + N_e$). A more cost-effective alternative approach focuses on the electronic DOFs (the so-called "system"), which are the quantity of interest when it comes to energy and charge transfer, and seeks a minimal and compact description of the effect of the nuclear DOFs (the so-called "bath") on them. In this case, the electronic DOFs constitute an open quantum system whose dynamics can be rigorously described by a GQME (see below).

A reasonable choice of overall system initial state is given by

$$\hat{\rho}(0) = \hat{\rho}_n(0) \otimes \hat{\sigma}(0) \quad . \tag{3}$$

Here, $\hat{\rho}_n(0) = \mathrm{Tr}_e\{\hat{\rho}(0)\}$ is the reduced density operator that describes the initial state of the nuclear DOFs, where $\mathrm{Tr}_e\{\cdot\}$ stands for partially tracing over the electronic Hilbert space. Similarly, $\hat{\sigma}(0)$ is the reduced density operator that describes the initial state of the electronic DOFs, as obtained by partially tracing over the nuclear Hilbert space:

$$\hat{\sigma}(0) = \mathrm{Tr}_n\{\hat{\rho}(0)\} = \sum_{j,k=1}^{N_e} \sigma_{jk}(0)|j\rangle\langle k| \quad . \tag{4}$$

Integrating the Eq. (2), the state of the overall system at a later time t is given by:

$$\hat{\rho}(t) = e^{-i\hat{H}t/\hbar}\hat{\rho}_n(0) \otimes \hat{\sigma}(0)e^{i\hat{H}t/\hbar} \equiv e^{-i\mathcal{L}t/\hbar}\hat{\rho}_n(0) \otimes \hat{\sigma}(0) \quad . \tag{5}$$

Here, $\hat{H}$ is the overall Hamiltonian, Eq. (1), and $\mathcal{L}(\cdot) = [\hat{H}, \cdot]$ is the corresponding Liouvillian. The electronic state at time t is given by the electronic reduced density operator:

$$\hat{\sigma}(t) \;=\; \mathrm{Tr}_n\{\hat{\rho}(t)\} = \sum_{j,k=1}^{N_e} \sigma_{jk}(t)|j\rangle\langle k| \quad . \tag{6}$$

Importantly, knowledge of $\hat{\sigma}(t)$ allows for the evaluation of both the electronic populations, $\{\sigma_{jj}(t) = \langle j|\hat{\sigma}(t)|j\rangle\}$, and coherences, $\{\sigma_{jk}(t) = \langle j|\hat{\sigma}(t)|k\rangle | j \neq k\}$.

In this tutorial, we focus on the GQME that treats $\hat{\sigma}(t)$ as the quantity of interest, which is given by:[3]

$$\frac{d}{dt}\hat{\sigma}(t) = -\frac{i}{\hbar}\langle\mathcal{L}\rangle_n^0\hat{\sigma}(t) - \int_0^t d\tau\, \mathcal{K}(\tau)\hat{\sigma}(t-\tau) \quad . \tag{7}$$

Here, $\langle\mathcal{L}\rangle_n^0$ is the projected Liouvillian averaged over the initial state of the nuclear DOF (resulting in a superoperator in the electronic Liouville-subspace), given by

$$\begin{aligned} \langle\mathcal{L}\rangle_n^0(\cdot) &\equiv \mathrm{Tr}_n\left\{\hat{\rho}_n(0)\mathcal{L}\right\}(\cdot) \\ &= \left[\sum_{j=1}^{N_e}\langle\hat{H}_j\rangle_n^0|j\rangle\langle j| + \sum_{\substack{j,k=1\\k\neq j}}^{N_e}\langle\hat{V}_{jk}\rangle_n^0|j\rangle\langle k|, \cdot\right] \quad , \end{aligned} \tag{8}$$

and $\mathcal{K}(\tau)$ is the *memory kernel* superoperator, given by

$$\mathcal{K}(\tau) = \frac{1}{\hbar^2}\mathrm{Tr}_n\Big\{\mathcal{L}e^{-i\mathcal{Q}\mathcal{L}\tau/\hbar}\mathcal{Q}\mathcal{L}\hat{\rho}_n(0)\Big\} \quad . \tag{9}$$

Here, $\mathcal{P}(\cdot) = \hat{\rho}_n(0)\otimes\mathrm{Tr}_n\{\cdot\}$ and $\mathcal{Q} = \mathcal{I} - \mathcal{P}$ are complementary projection superoperators ($\mathcal{I}$ is the unity superoperator). The forms and derivations of the above GQME, along with its $\langle\mathcal{L}\rangle_n^0$ and $\mathcal{K}(\tau)$, can be found in many previous studies.[4–13]

Recently, open quantum system dynamics based on the GQME has been simulated on NISQ computers based on the Sz.-Nagy dilation.[3] In this tutorial, we provide a step-by-step description of the implementation of this methodology on the spin-boson model.

2 The Spin-Boson Model

The spin-boson model is widely used for simulating electronic energy and charge transfer dynamics in chemical systems.[1,14] Used in this context, the two electronic states within this model ($N_e = 2$) correspond to the diabatic donor and acceptor states ($|D\rangle$ and $|A\rangle$, respectively). The nuclear Hamiltonians that correspond to the donor and acceptor states are assumed harmonic and identical except for a shift in equilibrium energy and optical geometry. Those assumptions give rise to a Hamiltonian of the following form:

$$\hat{H} = \epsilon\hat{\sigma}_z + \Gamma\hat{\sigma}_x + \sum_{i=1}^{N_n}\left[\frac{\hat{P}_i^2}{2} + \frac{1}{2}\omega_i^2\hat{R}_i^2 - c_i\hat{R}_i\hat{\sigma}_z\right] \quad . \tag{10}$$

Here, $\hat{\sigma}_z = |D\rangle\langle D| - |A\rangle\langle A|$, $\hat{\sigma}_x = |D\rangle\langle A| + |A\rangle\langle D|$, 2ϵ is the reaction energy and $\Gamma \equiv V_{DA}$ is the electronic coupling between the donor and acceptor states.

A spin-boson model Hamiltonian is often given in terms of the so-called spectral density underlying it, which is given by:

$$J(\omega) = \frac{\pi}{2}\sum_{k=1}^{N_n}\frac{c_k^2}{\omega_k}\delta(\omega - \omega_k). \tag{11}$$

For the sake of concreteness, we focus in this tutorial on the case of Ohmic spectral density with exponential cutoff (the methodology can easily accommodate other types of spectral densities):[15–17]

$$J(\omega) = \frac{\pi\hbar}{2}\xi\omega e^{-\omega/\omega_c}. \tag{12}$$

Here, ξ is the Kondo parameter which determines the coupling strength between the system and bath, and ω_c is the cutoff frequency. In what follows, we will also assume that the chemical system starts in the donor state in thermal equilibrium that corresponds to the nuclear Hamiltonian $(\hat{H}_D + \hat{H}_A)/2$, such that (the methodology can easily accommodate

other types initial states of the form of Eq. (3):

$$\hat{\rho}(0) = |D\rangle\langle D| \otimes \hat{\rho}_n(0) \tag{13}$$

$$\hat{\rho}_n(0) = \frac{\exp\left[-\beta \sum_{i=1}^{N_n} \frac{\hat{P}_i^2}{2} + \frac{1}{2}\omega_i^2 \hat{R}_i^2 \right]}{\mathrm{Tr}_n \left\{ \exp\left[-\beta \sum_{i=1}^{N_n} \frac{\hat{P}_i^2}{2} + \frac{1}{2}\omega_i^2 \hat{R}_i^2 \right] \right\}} \tag{14}$$

Here, $\beta = 1/k_B T$. where k_B is the Boltzmann constant and T is the absolute temperature. Thus, the electronic energy and charge transfer dynamics can be given in terms of five spin-boson model parameters: ϵ, Γ, β, ξ, and ω_c.

Script 2.1: Installing and importing dependencies

```python
import os
from google.colab import drive

# Mount Google Drive
drive.mount('/content/mydir')

# Define the path for the folder where you want to clone the repository
folder_path = '/content/mydir/MyDrive/GQME_Tutorial'

# Create the folder if it doesn't already exist
if not os.path.exists(folder_path):
    os.makedirs(folder_path)
    print(f"Created folder: {folder_path}")

# Change the current working directory to the folder where you want to clone the
↪  repository
os.chdir(folder_path)

# Clone the GitHub Repository into the specified folder
!git clone https://github.com/XiaohanDan97/CCI_PartIII_GQME .

import numpy as np
np.float = float
np.complex = complex
import time

#parameters in the simulation
from params import *
#read and write functions
import readwrite as wr

import matplotlib.pyplot as plt
```

Script 2.2: Spin-Boson Model parameters

```python
GAMMA_DA = 1 # diabatic coupling
EPSILON = 1
BETA = 5 # inverse finite temperature beta = 1 / (k_B * T)
XI = 0.1
OMEGA_C = 2
```

There are many numerically exact or approximate methods for the simulation of the spin-boson model,[17–25] here as an example, we show the result obtained from the numerically exact tensor-train thermofield dynamics (TT-TFD) method.[17]

Script 2.3: Using TT-TFD to simulate Spin-Boson Model

```
!pip install git+https://github.com/bcallen95/ttpy.git --quiet
import tt_tfd as tfd

#RDO: reduced density operator, contain the information of population and coherence
#initial_state=0: initial at Donor state
t, RDO_arr = tfd.tt_tfd(initial_state=0)

#TT-TFD is time-consuming, after running it once, you can read it from the file without
↪  running it again
#output to the file
wr.output_operator_array(t, RDO_arr, "TTTFD_Output/TFDSigma_")

#read and plot
t, RDO_arr = wr.read_operator_array("TTTFD_Output/TFDSigma_")
plt.figure(figsize=(6,2))
plt.plot(t, RDO_arr[:,0].real,'b-', label='TT-TFD')
plt.xlabel('$\Gamma t$',fontsize=15)
plt.ylabel('$\sigma_{DD}$(t)',fontsize=15)
plt.legend()
```

The result is shown in figure 1.

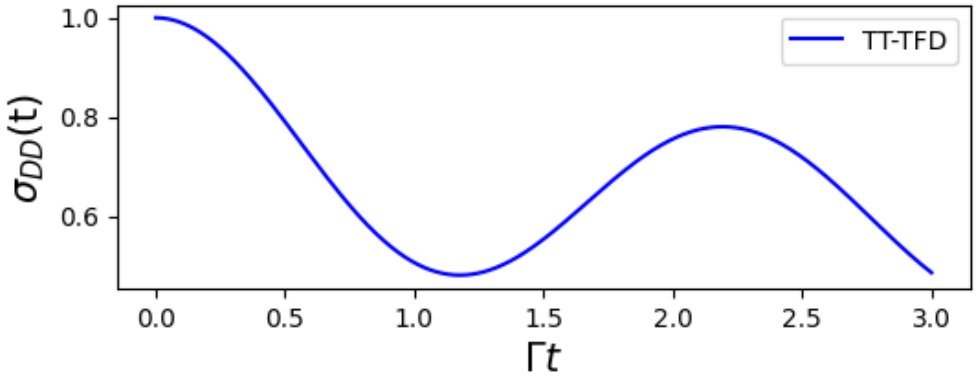

Figure 1: Population on the donor state $|D\rangle$ for the Spin-Boson model, obtained by TT-TFD method.

3 GQME for Spin-Boson Model

Since the spin-boson model is a two-state model ($N_e = 2$), the reduced electronic density operator is represented by a 2×2 matrix which can be given in terms of four matrix elements:

$\sigma_{ij}(t) = \langle i|\hat{\sigma}(t)|j\rangle$ with $i,j \in \{D,A\}$. Here, the diagonal elements $\sigma_{ii}(t)$, which are known as *populations*, correspond to the occupancies of the donor and acceptor states, while the off-diagonal terms, which are known as coherences, contain information about the coherent nature of the state.[1] Focusing on electronic energy and charge transfer, our focus would be on the dynamics of the populations of the donor and acceptor states, $\{\sigma_{DD}(t), \sigma_{AA}(t)\}$.

The electronic reduced density operator is represented in the code in its vectorized form:

$$\hat{\sigma}(t) \equiv [\sigma_{DD}(t), \sigma_{DA}(t), \sigma_{AD}(t), \sigma_{AA}(t)]^T \quad . \tag{15}$$

Thus, according to Eq. (13), the electronic initial state is $\hat{\sigma}(0) = |D\rangle\langle D| = [1,0,0,0]^T$. In this representation, the super-operators $\langle\mathcal{L}\rangle_n^0$ and $\mathcal{K}(t)$ are represented by 4×4 matrices.

3.1 The projected Liouvillian

We start with determining $\langle\mathcal{L}\rangle_n^0$ in Eq. (8). This electronic superoperator is defined by the way it acts on an arbitrary electronic operator $\hat{A}$:

$$\langle\mathcal{L}\rangle_n^0 \hat{A} = \mathrm{Tr}_n\left\{\left[\hat{H}, \hat{\rho}_n(0)\otimes\hat{A}\right]\right\} = \left[\epsilon\hat{\sigma}_z + \Gamma\hat{\sigma}_x, \hat{A}\right] \quad . \tag{16}$$

To get the second equality we used the property

$$\left[\frac{\hat{P}_i^2}{2} + \frac{1}{2}\omega_i^2\hat{R}_i^2, \hat{\rho}_n(0)\otimes\hat{A}\right] = \left[\frac{\hat{P}_i^2}{2} + \frac{1}{2}\omega_i^2\hat{R}_i^2, \hat{\rho}_n(0)\right]\otimes\hat{A} = 0 \quad ,$$

and

$$\mathrm{Tr}_n\left\{c_i\hat{R}_i\hat{\rho}_n(0)\right\} = 0 \quad ,$$

since the expectation value of the position operator in an unsifted harmonic oscillator at thermal equilibrium vanishes.

Using the vectorized form of the system subspace operator in Eq. (15), $\langle\mathcal{L}\rangle_n^0$ in Eq. (16) can be written as

$$\langle\mathcal{L}\rangle_n^0 = \begin{pmatrix} 0 & -\Gamma & \Gamma & 0 \\ -\Gamma & 2\epsilon & 0 & \Gamma \\ \Gamma & 0 & -2\epsilon & -\Gamma \\ 0 & \Gamma & -\Gamma & 0 \end{pmatrix} . \tag{17}$$

Script 3.1: projected Liouvillian

```
LN0 = np.zeros((DOF_E_SQ, DOF_E_SQ))
LN0[0][1] = LN0[1][0] = LN0[2][3] = LN0[3][2] = -GAMMA_DA
LN0[0][2] = LN0[2][0] = LN0[1][3] = LN0[3][1] = GAMMA_DA
LN0[1][1] = 2. * EPSILON
LN0[2][2] = -2. * EPSILON
```

The projected Liouvillian $\langle\mathcal{L}\rangle_n^0$ describes the time evolution that would be observed if the system was uncoupled from the bath [i.e. the memory kernel $\mathcal{K}(t)$ in Eq. (9) is zero since $\mathcal{P} = \mathcal{I}$ and $\mathcal{Q} = 0$], integrating Eq. (7) with $\mathcal{K}(t) = 0$ gives

$$\hat{\sigma}(t) = e^{-\frac{i}{\hbar}\langle\mathcal{L}\rangle_n^0 t}\hat{\sigma}(0) \quad . \tag{18}$$

Script 3.2: The dynamics with projected Liouvillian only

```
from scipy.linalg import expm
sigma_liou = np.zeros((TIME_STEPS, DOF_E_SQ), dtype=np.complex_)
time_arr = np.linspace(0,(TIME_STEPS-1)*DT,TIME_STEPS)
sigma_liou[0] = np.array([1.0,0,0,0],dtype=np.complex_)
for i in range(1,TIME_STEPS):
  sigma_liou[i] = expm(-1j*LN0*i*DT)@sigma_liou[0]

#read TT-TFD result and plot to compare
timeVec, sigma_tt_tfd = wr.read_operator_array("TTTFD_Output/TFDSigma_")
plt.figure(figsize=(6,2))
plt.plot(time_arr, sigma_liou[:,0].real,'b-', label='Liouvillian only')
plt.plot(timeVec,  sigma_tt_tfd[:,0].real,'ko', markersize=4,markevery=60,
  ↪  label='TT-TFD')
plt.xlabel('$\Gamma t$',fontsize=15)
plt.ylabel('$\sigma_{DD}$(t)',fontsize=15)
plt.legend(loc = 'upper right')
```

The result shown in Fig. 2. The system oscillates between the donor state $|D\rangle$ and the acceptor state $|A\rangle$, which corresponds to the time evolution of the pure system without the nuclear bath (i.e. dynamics of the two-level closed system[1]). Compared to the TT-TFD results, coupling to the bath brings in the dissipation effect, which makes the oscillation decay.

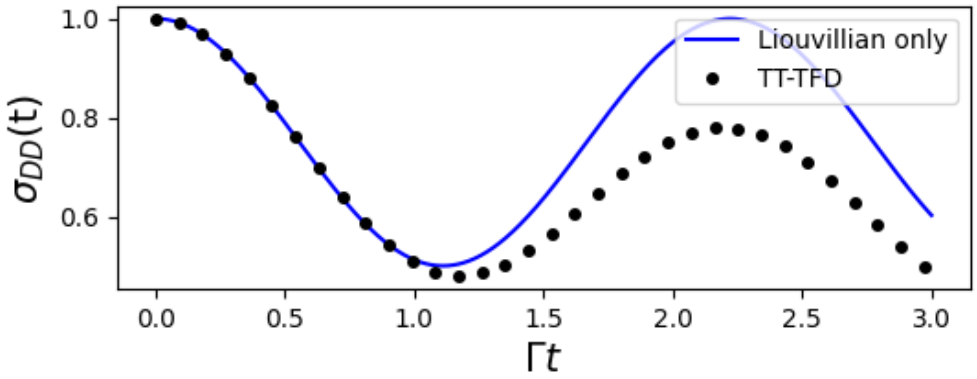

Figure 2: The population dynamics on the donor state $|D\rangle$. Here the dynamics correspond to the projected Liouvillian only. The numerically exact TT-TFD result is also shown for comparison.

4 The memory kernel

The memory kernel, Eq. (9), can be calculated by solving the following Volterra equation of the second kind:[20,26–29]

$$\mathcal{K}(t) = i\dot{\mathcal{F}}(t) - \frac{1}{\hbar}\mathcal{F}(t)\,\langle\mathcal{L}\rangle_n^0 + i\int_0^t \mathrm{d}\tau\,\mathcal{F}(t-\tau)\mathcal{K}(\tau) \quad . \tag{19}$$

Here, $\mathcal{F}(t)$ and $\dot{\mathcal{F}}(t)$, which are known as the projection-free inputs (PFIs), are given by:

$$\mathcal{F}(t) = \frac{1}{\hbar}\,\mathrm{Tr}_n[\mathcal{L}e^{-i\mathcal{L}t/\hbar}\hat{\rho}_n(0)] \tag{20}$$

$$\dot{\mathcal{F}}(t) = -\frac{i}{\hbar^2}\,\mathrm{Tr}_n[\mathcal{L}e^{-i\mathcal{L}t/\hbar}\mathcal{L}\hat{\rho}_n(0)] \quad . \tag{21}$$

Below we outline the procedures used for calculating the PFIs and then using them to obtain the memory kernel by solving Eq. (19).

4.1 Calculation of the Projection-Free Inputs

The PFIs can be obtained in multiple ways.[8,11,17,25] For the sake of concreteness, in this tutorial, we focus on obtaining them via TT-TFD. To this end, we note that $\mathcal{F}(t) = i\dot{\mathcal{U}}(t)$, where $\mathcal{U}(t)$ is the non-unitary time evolution superoperator, or *propagator*, of the system, which is defined by:

$$\hat{\sigma}(t) = \mathcal{U}(t)\hat{\sigma}(0) = \mathrm{Tr}_n\left[e^{-i\mathcal{L}t/\hbar}\hat{\rho}_n(0)\right]\hat{\sigma}(0) \quad , \tag{22}$$

Thus, $\mathcal{F}(t)$ and $\dot{\mathcal{F}}(t)$ can be obtained through taking time-derivatives of $\mathcal{U}(t)$. The propagator $\mathcal{U}(t)$, is a super-operator with the matrix element $\mathcal{U}_{jk,lm}(t)$ with $j,k,l,m \in \{D,A\}$, which can be defined by starting from initial state $|l\rangle\langle m| \otimes \hat{\rho}_n(0)$, and measuring $|j\rangle\langle k|$ at time t. The following code below shows obtaining $\{\mathcal{U}_{jk,lm}(t)\}$ via the TT-TFD method.[17]

Script 4.1: The propagator

```
def cal_U_tt_tfd():

    U = np.zeros((TIME_STEPS, DOF_E_SQ, DOF_E_SQ), dtype=np.complex_)

    # tt-tfd with initial state 0,1,2,3
    # initial state |0> means donor state |D>, |3> means acceptor state |A>
    # |1> is (|D> + |A>)/sqrt(2), |2> is (|D> + i|A>)/sqrt(2)
    t,U[:,:,0] = tfd.tt_tfd(0)
    t,U[:,:,1] = tfd.tt_tfd(1)
    t,U[:,:,2] = tfd.tt_tfd(2)
    t,U[:,:,3] = tfd.tt_tfd(3)

    U_final = U.copy()

    # the coherence elements that start at initial state |D><A| and |A><D|
    # is the linear combination of above U results
    # |D><A| = |1><1| + i * |2><2| - 1/2 * (1 + i) * (|0><0| + |3><3|)
    U_final[:,:,1] = U[:,:,1] + 1.j * U[:,:,2] - 0.5 * (1. + 1.j) * (U[:,:,0] +
    ↪  U[:,:,3])

    # |A><D| = |1><1| - i * |2><2| - 1/2 * (1 - i) * (|0><0| + |3><3|)
    U_final[:,:,2] = U[:,:,1] - 1.j * U[:,:,2] - 0.5 * (1. - 1.j) * (U[:,:,0] +
    ↪  U[:,:,3])

    #output U
    wr.output_superoper_array(t,U_final,"U_Output/U_")

    return 0
#The line below calculates all U elements with TT-TFD. The expected waiting time is 40
↪  minutes on Google Colab. To save time, the results are already pre-computed and
↪  saved, and this line is therefore commented out. The following code would still
↪  run normally. Please uncomment if one wishes to perform these calculations.
#cal_U_tt_tfd()
```

Once $\{\mathcal{U}_{jk,lm}(t)\}$ are obtained via TT-TFD, the PFIs $\mathcal{F}(t)$ and $\dot{\mathcal{F}}(t)$ can be obtained from it by taking time derivatives.

Script 4.2: Projection-Free Inputs $\mathcal{F}(\tau)$ and $\dot{\mathcal{F}}(\tau)$

```python
#the proj-free input from U data
def cal_F():
    #read the propagator data from files
    timeVec,U = wr.read_superoper_array("U_Output/U_")

    F = np.zeros((TIME_STEPS, DOF_E_SQ, DOF_E_SQ), dtype=np.complex_)
    Fdot = np.zeros((TIME_STEPS, DOF_E_SQ, DOF_E_SQ), dtype=np.complex_)

    for j in range(DOF_E_SQ):
        for k in range(DOF_E_SQ):
            # extracts real and imag parts of U element
            Ureal = U[:,j,k].copy().real
            Uimag = U[:,j,k].copy().imag

            # F = i * d/dt U so Re[F] = -1 * d/dt Im[U] and Im[F] = d/dt Re[U]
            Freal = -1. * np.gradient(Uimag.flatten(), DT, edge_order = 2)
            Fimag = np.gradient(Ureal.flatten(), DT, edge_order = 2)

            # Fdot = d/dt F so Re[Fdot] = d/dt Re[F] and Im[Fdot] = d/dt Im[F]
            Fdotreal = np.gradient(Freal, DT)
            Fdotimag = np.gradient(Fimag, DT)

            F[:,j,k] = Freal[:] + 1.j * Fimag[:]
            Fdot[:,j,k] = Fdotreal[:] + 1.j * Fdotimag[:]

    #write the result to the file
    wr.output_superoper_array(timeVec,F,"ProjFree_Output/F_")
    wr.output_superoper_array(timeVec,Fdot,"ProjFree_Output/Fdot_")

    return timeVec,F,Fdot
timeVec,F,Fdot = cal_F()
```

4.2 Calculation of the Memory Kernel

The memory kernel is obtained by solving Eq. (19). This is done by using the iterative algorithm outlined below. To this end, we put Eq. (19) in the following form

$$\mathcal{K}(t) = g(t) + \int_0^t f(t-\tau)\mathcal{K}(\tau)d\tau \tag{23}$$

where $g(t) = i\dot{\mathcal{F}}(t) - \frac{1}{\hbar}\mathcal{F}(t)\left\langle\mathcal{L}\right\rangle_n^0$ and $f(t-\tau) = \mathcal{F}(t-\tau)$.

Script 4.3: linear term $g(t)$

```
linearTerm = 1.j * Fdot.copy() # first term of the linear part
for l in range(TIME_STEPS):
    # subtracts second term of linear part
    linearTerm[l,:,:] -= 1./HBAR * F[l,:,:] @ LN0
```

Eq. (23) is solved via the iterative algorithm outlined next. To calculate $\mathcal{K}(n\Delta t)$ [where $n = 0, 1, 2, ..., N$ and $N\Delta t = t$], we start with an initial guess of $\mathcal{K}^0(n\Delta t) = g(n\Delta t)$ and iterate until convergence is accomplished:

$$\mathcal{K}^0(n\Delta t) = g(n\Delta t)$$
$$\mathcal{K}^1(n\Delta t) = g(n\Delta t) + \int_0^{n\Delta t} d\tau f(n\Delta t - \tau)\mathcal{K}^0(\tau)$$
$$\mathcal{K}^2(n\Delta t) = g(n\Delta t) + \int_0^{n\Delta t} d\tau f(n\Delta t - \tau)\mathcal{K}^1(\tau)$$
$$\vdots$$
$$\mathcal{K}^i(n\Delta t) = g(n\Delta t) + \int_0^{n\Delta t} d\tau f(n\Delta t - \tau)\mathcal{K}^{i-1}(\tau) \quad \text{where} \quad |\mathcal{K}^i(n\Delta t) - \mathcal{K}^{i-1}(n\Delta t)| \leq 10^{-10}$$

Within the code, the time integrals are calculated using the trapezoidal rule. Convergence is determined via the following criterion $|\mathcal{K}^{\text{sub},\,i}_{jk,lm}(n\Delta t) - \mathcal{K}^{\text{sub},\,i-1}_{jk,lm}(n\Delta t)| \leq 10^{-10}$ for all matrix elements j, k, l, m and time steps n.

Script 4.4: Memory Kernel - Volterra Algorithm

```
START_TIME = time.time() # starts timing
# sets initial guess to the linear part
prevKernel = linearTerm.copy()
kernel = linearTerm.copy()

# loop for iterations
for numIter in range(1, MAX_ITERS + 1):

    iterStartTime = time.time() # starts timing of iteration
    print("Iteration:", numIter)

    # calculates kernel using prevKernel and trapezoidal rule
    kernel = CalculateIntegral(DOF_E_SQ, F, linearTerm, prevKernel, kernel)

    numConv = 0 # parameter used to check convergence of entire kernel
    for i in range(DOF_E_SQ):
        for j in range(DOF_E_SQ):
            for n in range(TIME_STEPS):
                # if matrix element and time step of kernel is converged, adds 1
                if abs(kernel[n][i][j] - prevKernel[n][i][j]) <= CONVERGENCE_PARAM:
                    numConv += 1

                # if at max iters, prints which elements and time steps did not
                # converge and prevKernel and kernel values
                elif numIter == MAX_ITERS:
                    print("\tK time step and matrix element that didn't converge: %s,
                    ↪  %s%s"%(n,i,j))

    print("\tIteration time:", time.time() - iterStartTime)

    # enters if all times steps and matrix elements of kernel converged
    if numConv == TIME_STEPS * DOF_E_SQ * DOF_E_SQ:
        # prints number of iterations and time necessary for convergence
        print("Number of Iterations:", numIter, "\tVolterra time:", time.time() -
        ↪  START_TIME)

        # prints memory kernel to files
        wr.output_superoper_array(timeVec,kernel,"K_Output/K_")

        break # exits the iteration loop

    # if not converged, stores kernel as prevKernel, zeros the kernel, and then
    # sets kernel at t = 0 to linear part
    prevKernel = kernel.copy()
    kernel = linearTerm.copy()

    # if max iters reached, prints lack of convergence
    if numIter == MAX_ITERS:
        print("\tERROR: Did not converge for %s iterations"%MAX_ITERS)
        print("\tVolterra time:", print(time.time() - START_TIME))
```

The function *CalculateIntegral* calculates the integral part of the Volterra equation through the trapezoidal rule which approximates an integral on a uniform grid with N slices as:

$$\int_a^b f(t)dt \approx h\left[\frac{1}{2}f(a) + \frac{1}{2}f(b) + \sum_{k=1}^{N-1} f(a + k * h)\right], \tag{24}$$

where $h = (a - b) / N$.

Script 4.5: Function to Calculate Integral via Trapezoidal Rule

```
def CalculateIntegral(DOF_E_SQ, F, linearTerm, prevKernel, kernel):

    # time step loop starts at 1 because K is equal to linear part at t = 0
    for n in range(1, TIME_STEPS):
        kernel[n,:,:] = 0.

        # f(a) and f(b) terms
        kernel[n,:,:] += 0.5 * DT * F[n,:,:] @ kernel[0,:,:]
        kernel[n,:,:] += 0.5 * DT * F[0,:,:] @ prevKernel[n,:,:]

        # sum of f(a + kh) term
        for c in range(1, n):
            # since a new (supposed-to-be-better) guess for the
            # kernel has been calculated for previous time steps,
            # can use it rather than prevKernel
            kernel[n,:,:] += DT * F[n - c,:,:] @ kernel[c,:,:]

        # multiplies by i and adds the linear part
        kernel[n,:,:] = 1.j * kernel[n,:,:] + linearTerm[n,:,:]

    return kernel
```

As an illustration, Fig. 3 depicts two matrix elements $\mathcal{K}_{DD,DD}(t)$ and $\mathcal{K}_{DA,DD}(t)$, of the memory kernel. The memory kernel describes the influence of the bath on the system dynamics. The small amplitude of $\mathcal{K}_{DD,DD}(t)$ implies that the environmental effect does not directly impact the dynamics of σ_{DD} itself. $\mathcal{K}_{DA,DD}(t)$ exhibits a larger amplitude, indicating that the bath induces the population σ_{DD} at an earlier time to influence the coherence dynamics of σ_{DA} at the current time, which is the feature of non-Markovian dynamics. $\mathcal{K}_{DA,DD}(t)$ initiates from 0, increases, and then decays over time due to the relaxation of the

bath. If at time τ_B $\mathcal{K}_{DA,DD}(t)$ decays to zero, the bath loses its memory at the timescale of τ_B, signifying that σ_{DD} before $t - \tau_B$ no longer influences the dynamics of σ_{DA} at time t.

Script 4.6: Plot the memory kernel

```
#plot the kernel without the last two boundary points that have numerical errors
plt.figure(figsize=(6,2))
plt.plot(timeVec[:-2], kernel[:-2,1,0].real,'b-', label='Re $\mathcal{K}_{DA,DD}$')
plt.plot(timeVec[:-2], kernel[:-2,0,0].real,'k-', label='Re $\mathcal{K}_{DD,DD}$')
plt.xlabel('$\Gamma t$',fontsize=15)
plt.ylabel('$\mathcal{K}$(t)',fontsize=15)
plt.legend(loc = 'upper right')
```

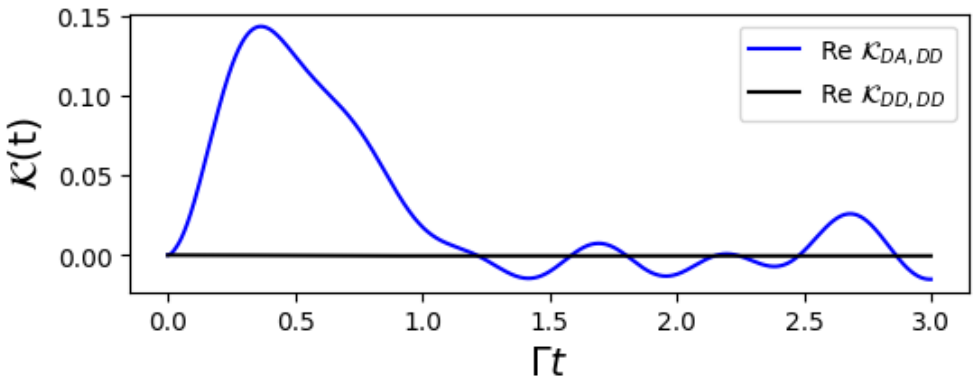

Figure 3: Memory kernel for the Spin-Boson model, here only $\mathcal{K}_{DD,DD}(t)$ and $\mathcal{K}_{DA,DD}(t)$ elements are shown.

5 Solution of the GQME

Given $\langle \mathcal{L} \rangle_n^0$ and $\mathcal{K}(\tau)$, as outlined in the preceding subsections, the GQME, Eq. (7) is solved using the 4th-order Runge-Kutta (RK4) method. More specifically, given the initial value problem

$$\frac{dy}{dt} = f(t, y) \quad \text{with an initial value} \quad y(t_0) = y_0 \ , \tag{25}$$

and substituting $\hat{\sigma}(t)$ for y, the RK4 method propagates y from time t_n to time t_{n+1} ($n = 0, 1, 2, ...$) as follows:

$$y_{n+1} = y_n + \frac{h}{6}\big(k_1 + 2k_2 + 2k_3 + k_4\big), \tag{26}$$

where $y_n = y(t_n)$, $y_{n+1} = y(t_{n+1})$, h is the time step, and

$$\begin{aligned} k_1 &= f\big(t_n, y_n\big) \\ k_2 &= f\left(t_n + \frac{h}{2}, y_n + \frac{h}{2}k_1\right) \\ k_3 &= f\left(t_n + \frac{h}{2}, y_n + \frac{h}{2}k_2\right) \\ k_4 &= f\big(t_n + h, y_n + hk_3\big). \end{aligned}$$

Script 5.1: GQME - Propagation via RK4 Method

```
def PropagateRK4(currentTime, memTime, kernel,
             sigma_hold, sigma):

    f_0 = Calculatef(currentTime, memTime,
                 kernel, sigma, sigma_hold)

    k_1 = sigma_hold + DT * f_0 / 2.
    f_1 = Calculatef(currentTime + DT / 2., memTime,
                 kernel, sigma, k_1)

    k_2 = sigma_hold + DT * f_1 /2.
    f_2 = Calculatef(currentTime + DT / 2., memTime,
                 kernel, sigma, k_2)

    k_3 = sigma_hold + DT * f_2
    f_3 = Calculatef(currentTime + DT, memTime,
                 kernel, sigma, k_3)

    sigma_hold += DT / 6. * (f_0 + 2. * f_1 + 2. * f_2 + f_3)

    return sigma_hold
```

Compare to GQME in Eq. (7), the time-derivate function $f(t, y)$ in Eq. (25) is

$$f\left(t, \hat{\sigma}\right) = -\frac{i}{\hbar} \sum_{lm} \langle \mathcal{L}_{jk,lm} \rangle_n^0 \hat{\sigma}_{lm}(t) - \sum_{lm} \int_0^t d\tau \mathcal{K}_{jk,lm}(\tau) \hat{\sigma}_{lm}(t-\tau) \quad , \tag{27}$$

which is calculated using the extended trapezoidal rule using the function *Calculatef*

Script 5.2: Calculating the function f

```
def Calculatef(currentTime, memTime, kernel, sigma_array, kVec):
    global LN0, HBAR

    memTimeSteps = int(memTime / DT)
    currentTimeStep = int(currentTime / DT)

    f_t = np.zeros(kVec.shape, dtype=np.complex_)

    f_t -= 1.j / HBAR * LN0 @ kVec

    limit = memTimeSteps
    if currentTimeStep < (memTimeSteps - 1):
        limit = currentTimeStep
    for l in range(limit):
        f_t -= DT * kernel[l,:,:] @ sigma_array[currentTimeStep - l]

    return f_t
```

With the functions *PropagateRK4* and *Calculatef* defined, the GQME is solved to obtain $\hat{\sigma}(t)$.

Script 5.3: GQME - Propagation of the Density Matrix

```python
#read the memory kernel
timeVec,kernel = wr.read_superoper_array("K_Output/K_")

# array for reduced density matrix elements
sigma = np.zeros((TIME_STEPS, DOF_E_SQ), dtype=np.complex_)
# array to hold copy of sigma
sigma_hold = np.zeros(DOF_E_SQ, dtype = np.complex_)

# sets the initial state at Donor State
sigma[0,0] = 1.
sigma_hold[0] = 1.

# loop to propagate sigma
print(">>> Starting GQME propagation, memory time =", MEM_TIME)
for l in range(TIME_STEPS - 1): # it propagates to the final time step
    if l%100==0: print(l)
    currentTime = l * DT

    sigma_hold = PropagateRK4(currentTime, MEM_TIME, kernel, sigma_hold, sigma)

    sigma[l + 1] = sigma_hold.copy()

# prints sigma to files
wr.output_operator_array(timeVec, sigma, "GQME_Output/Sigma")

# Read the reference data and plot
timeVec, sigma_tt_tfd = wr.read_operator_array("TTTFD_Output/TFDSigma_")
timeVec, sigma = wr.read_operator_array("GQME_Output/Sigma_")
plt.figure(figsize=(6,2))
plt.plot(timeVec, sigma[:,0],'b-', label='GQME')
plt.plot(timeVec, sigma_tt_tfd[:,0] ,'ko', markersize=4, markevery=60,
  ↪  label='benchmark_TT-TFD')
plt.xlabel('$\Gamma t$',fontsize=15)
plt.ylabel('$\sigma_{DD}$(t)',fontsize=15)
plt.legend()
```

The result shows that the dynamics calculated from GQME are the same as the Exact TT-TFD result. Which demonstrates the correctness of our memory kernel and GQME approach.

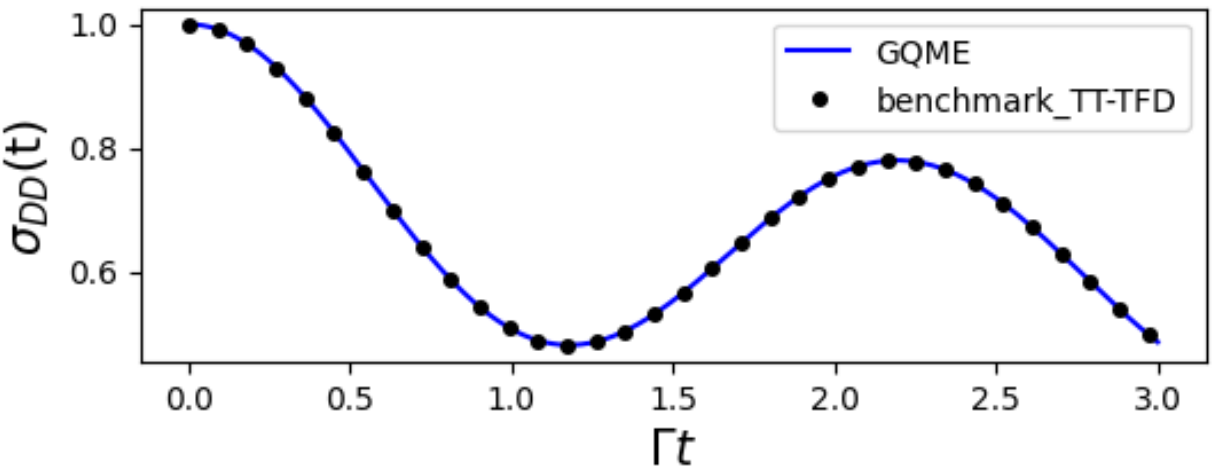

Figure 4: GQME result of the population on the donor state $|D\rangle$ for the Spin-Boson model, compared with numerically exact TT-TFD result.

6 Quantum Algorithms of GQME based on Dilation

6.1 Solving the GQME to get the propagator

In this section, we introduce the quantum simulation of the GQME. To start with, the open quantum system's *non-unitary* time evolution propagator $\mathcal{G}(t)$ is given by:

$$\hat{\sigma}(t) = \mathcal{G}(t)\hat{\sigma}(0) \quad . \tag{28}$$

This non-unitarity arises due to the interaction of the system with its surrounding environment, leading to irreversible processes and the often intricate behavior of the system over time. Substituting Eq. (28) into Eq. (7) and noting that the GQME should be satisfied for an arbitrary choice of $\hat{\sigma}(0)$, it is straightforward to show that $\mathcal{G}(t)$ satisfies the same GQME as $\hat{\sigma}(t)$:

$$\frac{d}{dt}\mathcal{G}(t) = -\frac{i}{\hbar}\langle\mathcal{L}\rangle_n^0\mathcal{G}(t) - \int_0^t d\tau\ \mathcal{K}(\tau)\mathcal{G}(t-\tau) \quad . \tag{29}$$

Therefore, start from identity superoperator $\mathcal{G}(0) = I$, we can calculate $\mathcal{G}(t)$ by solving the GQME with the same $\langle \mathcal{L} \rangle_n^0$ and $\mathcal{K}(\tau)$ given in previous sections.

Script 6.1: Calculating $\mathcal{G}(t)$ by solving the GQME

```
#read the memory kernel
timeVec,kernel = wr.read_superoper_array("K_Output/K_")

# array for Propagator superoperator elements
G_prop = np.zeros((TIME_STEPS, DOF_E_SQ, DOF_E_SQ), dtype=np.complex_)

#time 0 propagator: identity superoperator
G_prop[0] = np.eye(DOF_E_SQ)
#array to hold copy of G propagator
G_prop_hold = np.eye((DOF_E_SQ), dtype=np.complex_)

# loop to propagate G_prop using GQME
print(">>> Starting GQME propagation, memory time =", MEM_TIME)
for l in range(TIME_STEPS - 1): # it propagates to the final time step
    if l%100==0: print(l)
    currentTime = l * DT

    G_prop_hold = PropagateRK4(currentTime, MEM_TIME, kernel, G_prop_hold, G_prop)

    G_prop[l + 1] = G_prop_hold.copy()
```

6.2 Dilation of the non-unitary propagator

Next, we delve into the crucial step in our workflow, wherein we harness the power of the Sz.-Nagy unitary dilation procedure[30–33] to perform simulations on a quantum computer. This technique enables us to transform the non-unitary propagator $\mathcal{G}(t)$, into a unitary propagator that inhabits an extended Hilbert space.[3]

We initiate the process by computing the operator norm of $\mathcal{G}(t)$ to assess whether it qualifies as a "contraction". For $\mathcal{G}(t)$ to meet the criteria for being a contraction, its operator norm must satisfy the condition $||\mathcal{G}(t)||_O = \sup \frac{||\mathcal{G}(t)\boldsymbol{v}||}{||\boldsymbol{v}||} \leq 1$. In scenarios where the original $\mathcal{G}(t)$ does not satisfy the contraction requirement, we introduce a normalization factor denoted as n_c, which can be chosen as a number greater than $||\mathcal{G}(t)||_O$. This factor is utilized to redefine $\mathcal{G}(t)$ into a contraction form, specifically $\mathcal{G}'(t) = \mathcal{G}(t)/n_c$.

With $\mathcal{G}'(t)$ as a "contraction", the unitary operator denoted as $\mathcal{U}_{\mathcal{G}'}(t)$ is defined as:

$$\mathcal{U}_{\mathcal{G}'}(t) = \begin{pmatrix} \mathcal{G}'(t) & \mathcal{D}_{\mathcal{G}'^{\dagger}}(t) \\ \mathcal{D}_{\mathcal{G}'}(t) & -\mathcal{G}'^{\dagger}(t) \end{pmatrix} \quad , \tag{30}$$

Here, $\mathcal{D}_{\mathcal{G}'}(t) = \sqrt{I - \mathcal{G}'^{\dagger}(t)\mathcal{G}'(t)}$ and $\mathcal{D}_{\mathcal{G}'^{\dagger}} = \sqrt{I - \mathcal{G}'(t)\mathcal{G}'^{\dagger}(t)}$, with $\mathcal{D}_{\mathcal{G}'}(t)$ representing the so-called defect superoperator of $\mathcal{G}'(t)$. The resulting $\mathcal{U}_{\mathcal{G}'}(t)$ is a unitary superoperator, and resides in an extended Hilbert space that have the double size of the original system's Hilbert space. Importantly, $\mathcal{U}_{\mathcal{G}'}(t)$ replicates the effect of $\mathcal{G}'(t)$ in the original Hilbert space.

$$\mathcal{G}'(t)\hat{\sigma}(0) \xrightarrow{\text{unitary dilation}} \mathcal{U}_{\mathcal{G}'}(t) \left(\hat{\sigma}(0)^T, 0, \cdots, 0\right)^T . \tag{31}$$

By zero-padding the input vector to match the dimensionality of the expanded Hilbert space, the result vector obtained from the action of $\mathcal{U}_{\mathcal{G}'}(t)$ on the extended input vector, when projecting onto the original Hilbert space, is equivalent to the vector of $\mathcal{G}'(t)$ acting on the original input vector.

The following function *dilate* defined the dilation procedure, it gives $\mathcal{U}_{\mathcal{G}'}(t)$ and normalization factor n_c with $\mathcal{G}(t)$ as input.

Script 6.2: Dilation of the non-unitary propagator

```python
from numpy import linalg as la
import scipy.linalg as sp

def dilate(array):

 # Normalization factor of 1.5 to ensure contraction
 norm = la.norm(array,2)*1.5
 array_new = array/norm

 ident = np.eye(array.shape[0])

 # Calculate the conjugate transpose of the G propagator
 fcon = (array_new.conjugate()).T

 # Calculate the defect matrix for dilation
 fdef = sp.sqrtm(ident - np.dot(fcon, array_new))

 # Calculate the defect matrix for the conjugate of the G propagator
 fcondef = sp.sqrtm(ident - np.dot(array_new, fcon))

 # Dilate the G propagator to create a unitary operator
 array_dilated = np.block([[array_new, fcondef], [fdef, -fcon]])

 return array_dilated, norm
```

6.3 Quantum Simulation of GQME with QASM Simulator

In this section, we will delve into the simulation of GQME using Qiskit's QASM simulator, focusing on the spin-boson model.[3] The quantum algorithm starts from initializing the quantum circuit with the initial state $\left(\hat{\sigma}(0)^T, 0, \cdots, 0\right)^T$. For the spin-boson model, this requires 3 qubits with 2 from the four components of $\hat{\sigma}(0)$ as in Eq. (15) and 1 from the dilation procedure that doubles the space. After initialization, the dilated propagator $\mathcal{U}_{G'}(t)$ is converted into a quantum gate and applied to the quantum circuit. Then, measuring two qubits at 00 or 11 [the first or fourth component in Eq. (15)] with the dilated qubit at 0, the electronic populations can be retrieved by taking the square root of the measuring probability and multiplying by the normalization factor n_c is the dilation process: $\hat{\sigma}_{DD}(t) = n_c * \sqrt{P_{000}}$ and $\hat{\sigma}_{AA}(t) = n_c * \sqrt{P_{011}}$. The quantum circuit is shown in figure 5.

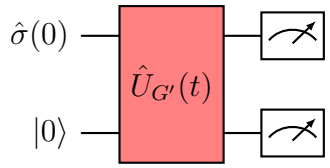

Figure 5: Circuit for implementing the GQME with a one-qubit dilation.

For each specific time t, we generate the quantum circuit and perform the simulations. We implement the quantum circuit using Qiskit's QASM simulator, which is shown below.

Script 6.3: Installing and importing Qiskit dependencies

```
!pip install qiskit==0.45
!pip install qiskit-aer

from qiskit import QuantumRegister, ClassicalRegister, QuantumCircuit, execute, Aer
from qiskit.visualization import plot_histogram
from qiskit.quantum_info import Operator
from qiskit.compiler import transpile
```

Script 6.4: QASM simulation for GQME

```python
# Create a dictionary to store the measurement results
result = {'000': 0, '001': 0, '010': 0, '011': 0, '100': 0, '101': 0, '110': 0, '111': 0}

# Create lists to store the population for the acceptor and donor states
pop_accept = []
pop_donor = []

# initial state in the dilated space
rho0_dilated = np.concatenate((np.array([1 + 0j, 0, 0, 0]),np.zeros(DOF_E_SQ)))

for i in range(TIME_STEPS):

    qr = QuantumRegister(3)  # Create a quantum register with 3 qubits
    cr = ClassicalRegister(3)  # Create a classical register to store measurement results
    qc = QuantumCircuit(qr, cr)  # Combine the quantum and classical registers to create the quantum circuit

    # Initialize the quantum circuit with the initial state
    qc.initialize(rho0_dilated, qr)

    # Dilated propagator
    U_G, norm = dilate(G_prop[i])

    # Create a custom unitary operator with the dilated propagator
    U_G_op = Operator(U_G)

    # Apply the unitary operator to the quantum circuit's qubits
    qc.unitary(U_G_op, qr)
    # Measure the qubits and store the results in the classical register
    qc.measure(qr, cr)

    #Run the Simulation and Plot the Results
    shots = 2000  # Number of shots
    counts = execute(qc, Aer.get_backend('qasm_simulator'), shots=shots).result().get_counts()

    # Update the result dictionary
    for x in counts:
       result[x] = counts[x]

    # Calculate the populations of donor and acceptor states from measurement probabilities
    pd = np.sqrt(result['000'] / 2000) * norm  # Multiply by the normalization factor
    pa = np.sqrt(result['011'] / 2000) * norm  # Multiply by the normalization factor

    pop_donor.append(pd)  # Stacking the population for the donor state
    pop_accept.append(pa)  # Stacking the population for the acceptor state
```

With the QAMS simulations complete, we can compare the resulting electronic state population dynamics to the Exact TT-TFD result.

Script 6.5: Visualizing the Results

```
# Read the exact TT-TFD results
timeVec, sigma_tt_tfd = wr.read_operator_array("TTTFD_Output/TFDSigma_")
# Plot the population of the donor and acceptor states
plt.figure(figsize=(6,2))
plt.plot(timeVec, pop_donor, 'r-', label="quantum simulation")
plt.plot(timeVec, sigma_tt_tfd[:,0].real ,'ko', markersize=4, markevery=60,
    label='benchmark_TT-TFD')
plt.xlabel('$\Gamma t$',fontsize=15)
plt.ylabel('$\sigma_{DD}$(t)',fontsize=15)
plt.legend(loc = 'upper right')
```

This is shown in figure 6.

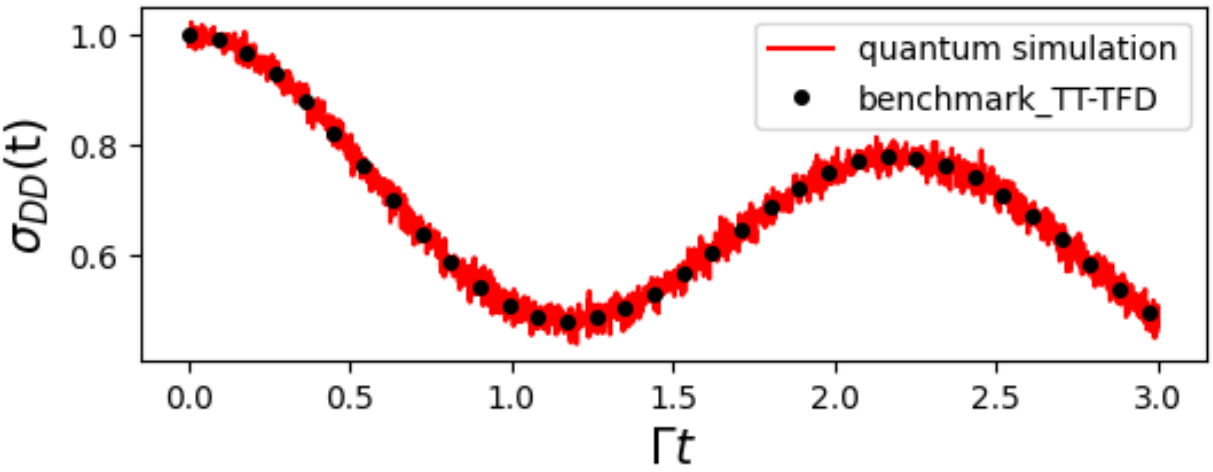

Figure 6: Electronic donor state population dynamics of the spin-boson model, simulated by the GQME-based quantum algorithm as implemented on the IBM QASM quantum simulator. The result is compared to the numerically exact TT-TFD result.

7 Conclusion

In part III of the whole series, we have covered the basics of generalized quantum master equations, illustrating their simulation on quantum computers through the spin-boson model

example. We first introduce the basis of GQME and provide the classical simulation with Python. Then we introduce the quantum algorithm based on Sz.-Nagi dilation that can simulate GQME on a quantum computer. We hope this part serves as a starting point for the simulations of exact non-Markovian open quantum systems dynamics with quantum computers.

8 Acknowledgement

We acknowledge the financial support of the National Science Foundation under award number 2124511, CCI Phase I: NSF Center for Quantum Dynamics on Modular Quantum Devices (CQD-MQD).

References

(1) Nitzan, A. *Chemical Dynamics in Condensed Phases*; Oxford University Press: New York, 2006.

(2) Breuer, H.-P.; Petruccione, F. *The Theory of Open Quantum Systems*; Oxford University Press, 2007.

(3) Wang, Y.; Mulvihill, E.; Hu, Z.; Lyu, N.; Shivpuje, S.; Liu, Y.; Soley, M. B.; Geva, E.; Batista, V. S.; Kais, S. Simulating Open Quantum System Dynamics on NISQ Computers with Generalized Quantum Master Equations. *J. Chem. Theory Comput.* **2023**, *19*, 4851–4862.

(4) Montoya-Castillo, A.; Reichman, D. R. Approximate but accurate quantum dynamics from the Mori formalism: I. Nonequilibrium dynamics. *J. Chem. Phys.* **2016**, *144*, 184104.

(5) Montoya-Castillo, A.; Reichman, D. R. Approximate but accurate quantum dynamics

from the Mori formalism. II. Equilibrium time correlation functions. *J. Chem. Phys.* **2017**, *146*, 084110.

(6) Zwanzig, R. Memory Effects in Irreversible Thermodynamics. *Phys. Rev.* **1961**, *124*, 983–992.

(7) Pfalzgraff, W. C.; Montoya-Castillo, A.; Kelly, A.; Markland, T. E. Efficient construction of generalized master equation memory kernels for multi-state systems from nonadiabatic quantum-classical dynamics. *J. Chem. Phys.* **2019**, *150*, 244109.

(8) Mulvihill, E.; Schubert, A.; Sun, X.; Dunietz, B. D.; Geva, E. A modified approach for simulating electronically nonadiabatic dynamics via the generalized quantum master equation. *J. Chem. Phys.* **2019**, *150*, 034101.

(9) Mulvihill, E.; Gao, X.; Liu, Y.; Schubert, A.; Dunietz, B. D.; Geva, E. Combining the mapping Hamiltonian linearized semiclassical approach with the generalized quantum master equation to simulate electronically nonadiabatic molecular dynamics. *J. Chem. Phys.* **2019**, *151*, 074103.

(10) Mulvihill, E.; Lenn, K. M.; Gao, X.; Schubert, A.; Dunietz, B. D.; Geva, E. Simulating energy transfer dynamics in the Fenna–Matthews–Olson complex via the modified generalized quantum master equation. *J. Chem. Phys.* **2021**, *154*, 204109.

(11) Mulvihill, E.; Geva, E. Simulating the dynamics of electronic observables via reduced-dimensionality generalized quantum master equations. *J. Chem. Phys.* **2022**, *156*, 044119.

(12) Ng, N.; Limmer, D. T.; Rabani, E. Nonuniqueness of generalized quantum master equations for a single observable. *J. Chem. Phys.* **2021**, *155*, 156101.

(13) Sayer, T.; Montoya-Castillo, A. Efficient formulation of multitime generalized quantum

master equations: Taming the cost of simulating 2D spectra. *J. Chem. Phys.* **2024**, *160*, 044108.

(14) May, V.; Kühn, O. *Charge and Energy Transfer Dynamics in Molecular Systems*, 2nd ed.; Wiley-VCH: Weinheim, 2004.

(15) Lai, Y.; Geva, E. On simulating the dynamics of electronic populations and coherences via quantum master equations based on treating off-diagonal electronic coupling terms as a small perturbation. *J. Chem. Phys.* **2021**, *155*, 204101.

(16) Sun, X.; Geva, E. Exact vs. asymptotic spectral densities in the Garg-Onuchic-Ambegaokar charge transfer model and its effect on Fermi's golden rule rate constants. *J. Chem. Phys.* **2016**, *144*, 044106.

(17) Lyu, N.; Mulvihill, E.; Soley, M. B.; Geva, E.; Batista, V. S. Tensor-Train Thermo-Field Memory Kernels for Generalized Quantum Master Equations. *J. Chem. Theory Comput.* **2023**, *19*, 1111–1129.

(18) Makarov, D. E.; Makri, N. Path integrals for dissipative systems by tensor multiplication. Condensed phase quantum dynamics for arbitrarily long time. *Chem. Phys. Lett.* **1994**, *221*, 482.

(19) Wang, H.; Thoss, M. Multilayer Formulation of the Multiconfiguration Time-dependent Hartree Theory. *J. Chem. Phys.* **2003**, *119*, 1289–1299.

(20) Shi, Q.; Geva, E. A new approach to calculating the memory kernel of the generalized quantum master equation for an arbitrary system-bath coupling. *J. Chem. Phys.* **2003**, *119*, 12063.

(21) Shi, Q.; Chen, L.; Nan, G.; Xu, R.-X.; Yan, Y. Efficient hierarchical Liouville space propagator to quantum dissipative dynamics. *J. Chem. Phys.* **2009**, *130*, 084105.

(22) Meyer, H.-D., Gatti, F., Worth, G. A., Eds. *Multidimensional Quantum Dynamics: MCTDH Theory and Applications*; Wiley-VCH: Weinheim, 2009.

(23) Kelly, A.; Brackbill, N.; Markland, T. E. Accurate nonadiabatic quantum dynamics on the cheap: Making the most of mean field theory with master equations. *J. Chem. Phys.* **2015**, *142*, 094110.

(24) Shi, Q.; Xu, Y.; Yan, Y.; Xu, M. Efficient propagation of the hierarchical equations of motion using the matrix product state method. *J. Chem. Phys.* **2018**, *148*, 174102.

(25) Xu, M.; Yan, Y.; Liu, Y.; Shi, Q. Convergence of high order memory kernels in the Nakajima-Zwanzig generalized master equation and rate constants: Case study of the spin-boson model. *J. Chem. Phys.* **2018**, *148*, 164101.

(26) Shi, Q.; Geva, E. A Semiclassical Generalized Quantum Master Equation for an Arbitrary System-Bath Coupling. *J. Chem. Phys.* **2004**, *120*, 10647–10658.

(27) Zhang, M.-L.; Ka, B. J.; Geva, E. Nonequilibrium quantum dynamics in the condensed phase via the generalized quantum master equation. *J. Chem. Phys.* **2006**, *125*, 044106.

(28) Kelly, A.; Montoya-Castillo, A.; Wang, L.; Markland, T. E. Generalized quantum master equations in and out of equilibrium: When can one win? *J. Chem. Phys.* **2016**, *144*, 184105.

(29) Dan, X.; Xu, M.; Yan, Y.; Shi, Q. Generalized master equation for charge transport in a molecular junction: Exact memory kernels and their high order expansion. *J. Chem. Phys.* **2022**, *156*, 134114.

(30) Levy, E.; Shalit, O. M. Dilation theory in finite dimensions: the possible, the impossible and the unknown. *Rocky Mt. J. Math.* **2014**, *44*, 203–221.

(31) Hu, Z.; Xia, R.; Kais, S. A quantum algorithm for evolving open quantum dynamics on quantum computing devices. *Sci. Rep.* **2020**, *10*, 1–9.

(32) Hu, Z.; Head-Marsden, K.; Mazziotti, D. A.; Narang, P.; Kais, S. A general quantum algorithm for open quantum dynamics demonstrated with the Fenna-Matthews-Olson complex. *Quantum* **2022**, *6*, 726.

(33) Zhang, Y.; Hu, Z.; Wang, Y.; Kais, S. Quantum Simulation of the Radical Pair Dynamics of the Avian Compass. *J. Phys. Chem. Lett.* **2022**, *14*, 832–837.