Multiple ancilla qubit simulation for time independent TFIM model

References:

- Zhiyan Ding, Chi-Fang Chen and Lin Lin: Single-ancilla ground state preparation via Lindbladians
- Zhiyan Ding and Xiantao Li and Lin Lin: Simulating Open Quantum Systems Using Hamiltonian Simulations

TFIM damping model, time independent Hamiltonian

```
import matplotlib.pyplot as plt
import qsimulations as qs
import numpy as np
from qutip import *

taylor_aprox_order = (
    8  # Taylor approximation used for simulating exp(-
i*sqrt*(dt)*H_tilde)
)

g = 1  # Couppling coefficient
gamma = 0.00  # Damping parameter

mu = 0.1

systemSize = 2  # System Hamiltonian
nrAncillas = 2  # Ancilla size
J = systemSize  # Nr of jump operators is equal to the number of
lattice elements
systemSize_dim = np.power(2, systemSize)  # Hamiltonian system size


T = 10  # Final time
dt = 0.01  # Time step
time_vec = np.arange(0, T, dt)  # Time vector to simulate on


def H_operator(t=0):
    H = np.zeros((systemSize_dim, systemSize_dim))
    if systemSize > 1:
        for i in np.arange(1, systemSize, 1):
            H = H - qs.Pauli_array(qs.Z, i, systemSize) @
qs.Pauli_array(
                qs.Z, i + 1, systemSize
            )
        H = H - qs.Pauli_array(qs.Z, systemSize, systemSize) @
qs.Pauli_array(
            qs.Z, 1, systemSize
        )
    for i in np.arange(1, systemSize + 1, 1):
        H = H - g * qs.Pauli_array(qs.X, i, systemSize)
    for i in np.arange(1, systemSize + 1, 1):
        H = H + mu * qs.Pauli_array(qs.Z, i, systemSize)
    return Qobj(H)


print(H_operator())


def V_damping(i, t=0):
    if i == 0:
        sum = 0
        for j in np.arange(1, J + 1, 1):
            sum = sum + V_damping(j).full().conj().T @
V_damping(j).full()
        return Qobj(-1j * H_operator().full() - 0.5 * sum)

    if i >= 1 and i <= systemSize_dim:
        return Qobj(
            0.5
            * np.sqrt(gamma)
            * (
                qs.Pauli_array(qs.X, i, systemSize)
                + 1j * qs.Pauli_array(qs.Y, i, systemSize)
            )
        )
    return 0


def H_operator_derivative(t):
    return Qobj(0)


def V_operator_derivative(i, t):
    return Qobj(0)


QSystem = qs.qsimulations(systemSize, systemSize, nrAncillas)
QSystem.H_op = H_operator
QSystem.H_op_derivative = H_operator_derivative
QSystem.V_op = V_damping
QSystem.V_op_derivative = V_operator_derivative
QSystem._update_module_varibles()
QSystem._prep_energy_states()
```

```
rho_ground = QSystem.rho_ground
rho_highest_en = QSystem.rho_highest_en

starting_state = rho_ground
mesurement_op = rho_ground
```

```
Quantum object: dims=[[4], [4]], shape=(4, 4), type='oper',
dtype=Dense, isherm=True
Qobj data =
[[-1.8 -1.  -1.   0. ]
 [-1.   2.   0.  -1. ]
 [-1.   0.   2.  -1. ]
 [ 0.  -1.  -1.  -2.2]]
[[ 0.65305897+0.j  0.06662845-0.j  0.26282052+0.j -0.70710678-0.j]]
```

Extra math

Simulating extra jump operator free TFIM time independent model and evaluating results both from Qutip and numerical Taylor approximation.

```
import scipy.linalg as la
import scipy, numpy, math

# All ground state energy vectors:
eigenValues, eigenVectors = la.eig(H_operator(0).full())
idx = eigenValues.argsort()
eigenValues = eigenValues[idx]
eigenVectors = eigenVectors[idx]

print(eigenValues)
print(eigenVectors)
print("Calculated: ", np.array(eigenVectors[0]).conj().T  @
np.array(eigenVectors[0]))


print(np.einsum('i,j->ij', eigenVectors[0], eigenVectors[0]))

print(starting_state)
# Time evolution of time independent hamiltonian, without damping
operators

def single_taylor(matrix, order, time_step):
    sum_tmp = np.zeros(np.power(2,systemSize))
    for i in np.arange(0, order, 1):
        # tmp = np.power(matrix, i)
        tmp = np.linalg.matrix_power(matrix, i)
        tmp = tmp / math.factorial(i)
        tmp = np.power(-1j*time_step, i)*tmp
        sum_tmp = sum_tmp + tmp
    return sum_tmp

results_Taylor = []
time_evolution = starting_state.full()

N = (int)(T/dt)
matrix = H_operator(0).full()
order = 5

for n in np.arange(0, N, 1):
    time_evolution = single_taylor(matrix, order, dt) @ time_evolution
@ np.conj(single_taylor(matrix, order, dt)).T
    results_Taylor.append(time_evolution)


result_taylor_overlap = []
for i in results_Taylor:
        result_taylor_overlap.append(np.trace(i @
mesurement_op.full()))


[-2.86781638+0.j -1.96182516+0.j  2.        +0.j  2.82964154+0.j]
[[ 6.53058975e-01+0.j  6.66284535e-02-0.j  2.62820518e-01+0.j
  -7.07106781e-01-0.j]
 [ 6.53058975e-01+0.j  6.66284535e-02-0.j  2.62820518e-01+0.j
   7.07106781e-01+0.j]
 [-2.59684102e-01+0.j -5.59491952e-01-0.j  7.87104137e-01+0.j
  -1.75220789e-16-0.j]
 [-2.82120751e-01+0.j  8.23462236e-01+0.j  4.92257888e-01+0.j
  -1.11859175e-16-0.j]]
Calculated:  (1+0j)
[[ 0.42648602+0.j  0.04351231+0.j  0.1716373 +0.j -0.46178243+0.j]
 [ 0.04351231+0.j  0.00443935+0.j  0.01751132+0.j -0.04711343+0.j]
 [ 0.1716373 +0.j  0.01751132+0.j  0.06907462+0.j -0.18584217+0.j]
 [-0.46178243+0.j -0.04711343+0.j -0.18584217+0.j  0.5       +0.j]]
Quantum object: dims=[[4], [4]], shape=(4, 4), type='oper',
dtype=Dense, isherm=True
Qobj data =
[[ 0.42648602  0.04351231  0.1716373  -0.46178243]
 [ 0.04351231  0.00443935  0.01751132 -0.04711343]
 [ 0.1716373   0.01751132  0.06907462 -0.18584217]
 [-0.46178243 -0.04711343 -0.18584217  0.5       ]]
```

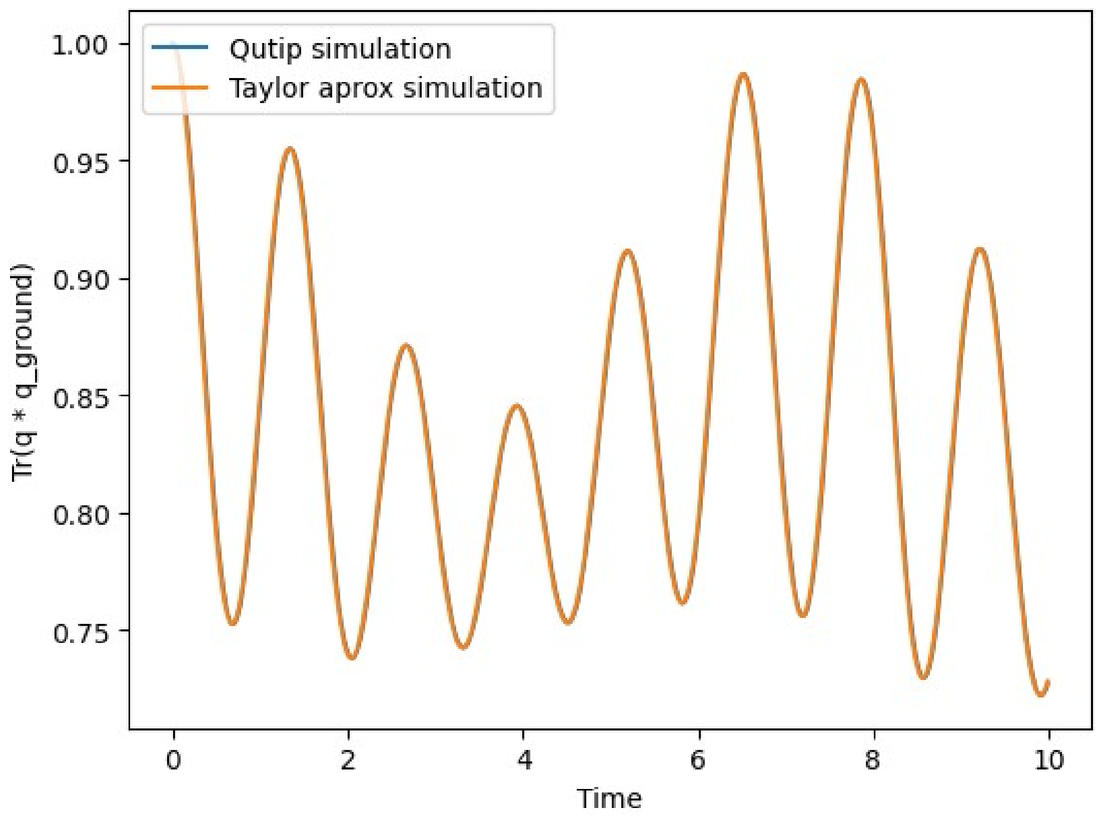
Exact simulation

```
import matplotlib.pyplot as plt
from qutip import mesolve
import numpy as np

rho = starting_state
```

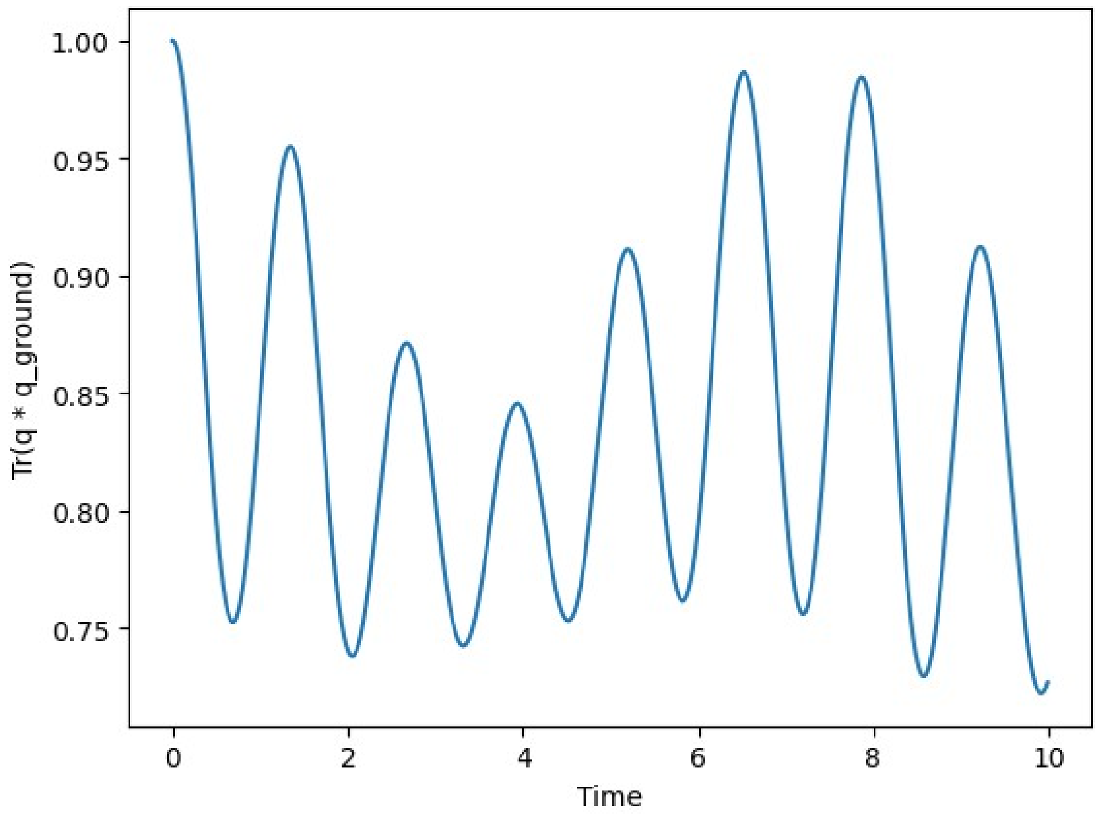
```
exact_trace = []
for t in time_vec:
    sum = Qobj(np.zeros((QSystem._systemSizeDim,
QSystem._systemSizeDim)))
    for j in np.arange(1, J + 1, 1):
        sum = (
            sum
            + QSystem.V_op(j) @ rho @ QSystem.V_op(j).conj().trans()
            - 0.5
            * (
                QSystem.V_op(j).conj().trans() @ QSystem.V_op(j) @ rho
                + rho @ QSystem.V_op(j).conj().trans() @
QSystem.V_op(j)
            )
        )
    delta_rho = -1j * (QSystem.H_op() @ rho - rho @ QSystem.H_op()) +
sum
    rho = rho + dt * delta_rho
    exact_trace.append((rho @ mesurement_op).tr())

V1 = V_damping(1)
V2 = V_damping(2)
# results2 = mesolve(
#     QSystem.H_op(), Qobj(starting_state), time_vec, [V1, V2],
[mesurement_op]
# )

results2 = mesolve(
    QSystem.H_op(), Qobj(starting_state), time_vec, c_ops=None,
e_ops=[mesurement_op]
)


# Add Taylor simulation for only the Hamiltonian system

plt.figure()
plt.xlabel("Time")
plt.ylabel("Tr(q * q_ground)")
# plt.plot(time_vec, exact_trace, label="Exact simulation")
plt.plot(time_vec, results2.expect[0], label="Qutip simulation")
plt.plot(time_vec, result_taylor_overlap, label="Taylor aprox
simulation")
plt.legend(loc="upper left")
```

```
/home/robi/.local/lib/python3.9/site-packages/matplotlib/cbook/
__init__.py:1335: ComplexWarning: Casting complex values to real
discards the imaginary part
  return np.asarray(x, float)
```

```
<matplotlib.legend.Legend at 0x761ef6aa5d60>
```



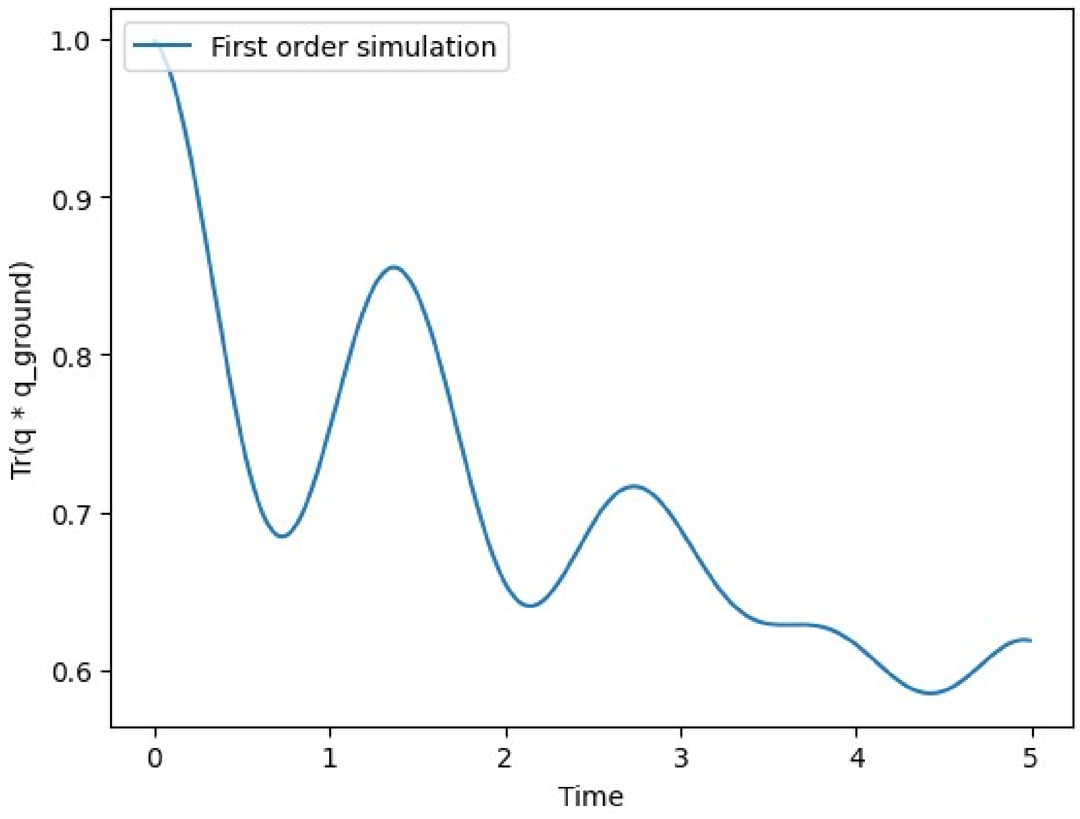
```
plt.figure()
plt.xlabel("Time")
plt.ylabel("Tr(q * q_ground)")
# plt.plot(time_vec, exact_trace, label="Exact simulation")
plt.plot(time_vec, results2.expect[0], label="Qutip simulation")
plt.plot(time_vec, results_Taylor, label="Taylor aprox simulation")
plt.legend(loc="upper left")

----------------------------------------------------------------------
-----
ValueError                                Traceback (most recent call
last)
Cell In[6], line 6
      4 # plt.plot(time_vec, exact_trace, label="Exact simulation")
      5 plt.plot(time_vec, results2.expect[0], label="Qutip
simulation")
----> 6 plt.plot(time_vec, results_Taylor, label="Taylor aprox
simulation")
      7 plt.legend(loc="upper left")

File ~/.local/lib/python3.9/site-packages/matplotlib/pyplot.py:2812,
in plot(scalex, scaley, data, *args, **kwargs)
   2810 @_copy_docstring_and_deprecators(Axes.plot)
   2811 def plot(*args, scalex=True, scaley=True, data=None,
**kwargs):
-> 2812     return gca().plot(
   2813         *args, scalex=scalex, scaley=scaley,
   2814         **({"data": data} if data is not None else {}),
**kwargs)

File
~/.local/lib/python3.9/site-packages/matplotlib/axes/_axes.py:1688, in
Axes.plot(self, scalex, scaley, data, *args, **kwargs)
   1445 """
   1446 Plot y versus x as lines and/or markers.
   1447
   (...)
   1685 (``'green'``) or hex strings (``'#008000'``).
   1686 """
   1687 kwargs = cbook.normalize_kwargs(kwargs, mlines.Line2D)
-> 1688 lines = [*self._get_lines(*args, data=data, **kwargs)]
   1689 for line in lines:
   1690     self.add_line(line)

File
~/.local/lib/python3.9/site-packages/matplotlib/axes/_base.py:311, in
_process_plot_var_args.__call__(self, data, *args, **kwargs)
    309     this += args[0],
    310     args = args[1:]
--> 311 yield from self._plot_args(
    312     this, kwargs, ambiguous_fmt_datakey=ambiguous_fmt_datakey)

File
~/.local/lib/python3.9/site-packages/matplotlib/axes/_base.py:507, in
_process_plot_var_args._plot_args(self, tup, kwargs, return_kwargs,
ambiguous_fmt_datakey)
    504     raise ValueError(f"x and y must have same first dimension,
but "
    505                      f"have shapes {x.shape} and {y.shape}")
    506 if x.ndim > 2 or y.ndim > 2:
--> 507     raise ValueError(f"x and y can be no greater than 2D, but
have "
    508                      f"shapes {x.shape} and {y.shape}")
    509 if x.ndim == 1:
    510     x = x[:, np.newaxis]

ValueError: x and y can be no greater than 2D, but have shapes (1000,)
and (1000, 4, 4)
```



First order approximation

```
import math
import numpy as np

ancilla = 2  # Ancillary system size
QSystem.set_nr_of_ancillas(ancilla)
ancilla_dim = np.power(2, ancilla)
total_systemSize = systemSize + ancilla  # Total system size
total_systemSize_dim = np.power(2, total_systemSize)
taylor_aproximation_order = 10

# First order scheme
psi_ancilla = 1
for i in range(ancilla):
    psi_ancilla = np.kron(psi_ancilla, qs.ket_0)
rho_ancilla = Qobj(psi_ancilla.conj().T @ psi_ancilla)

rho = starting_state
first_order_trace = []
for t in time_vec:
    # Extended system, zero initialized ancilla + hamiltonian system
    system = Qobj(
        tensor(rho_ancilla, rho),
        dims=[[ancilla_dim, systemSize_dim], [ancilla_dim,
systemSize_dim]],
    )
    # First element of taylor approximation, I
    approximation = Qobj(
        qeye(ancilla_dim * systemSize_dim),
        dims=[[ancilla_dim, systemSize_dim], [ancilla_dim, 
systemSize_dim]],
    )

    approximation = qs.Taylor_approximtion(
        QSystem.H_tilde_first_order(dt),
        taylor_aproximation_order,
        np.sqrt(dt),
        approximation,
    )
    evolved_system = approximation @ system @ 
approximation.conj().trans()

    rho = evolved_system.ptrace(1)
    first_order_trace.append((rho @ mesurement_op).tr())

plt.figure()
plt.xlabel("Time")
plt.ylabel("Tr(q * q_ground)")
plt.plot(time_vec, first_order_trace, label="First order simulation")
plt.legend(loc="upper left")
```

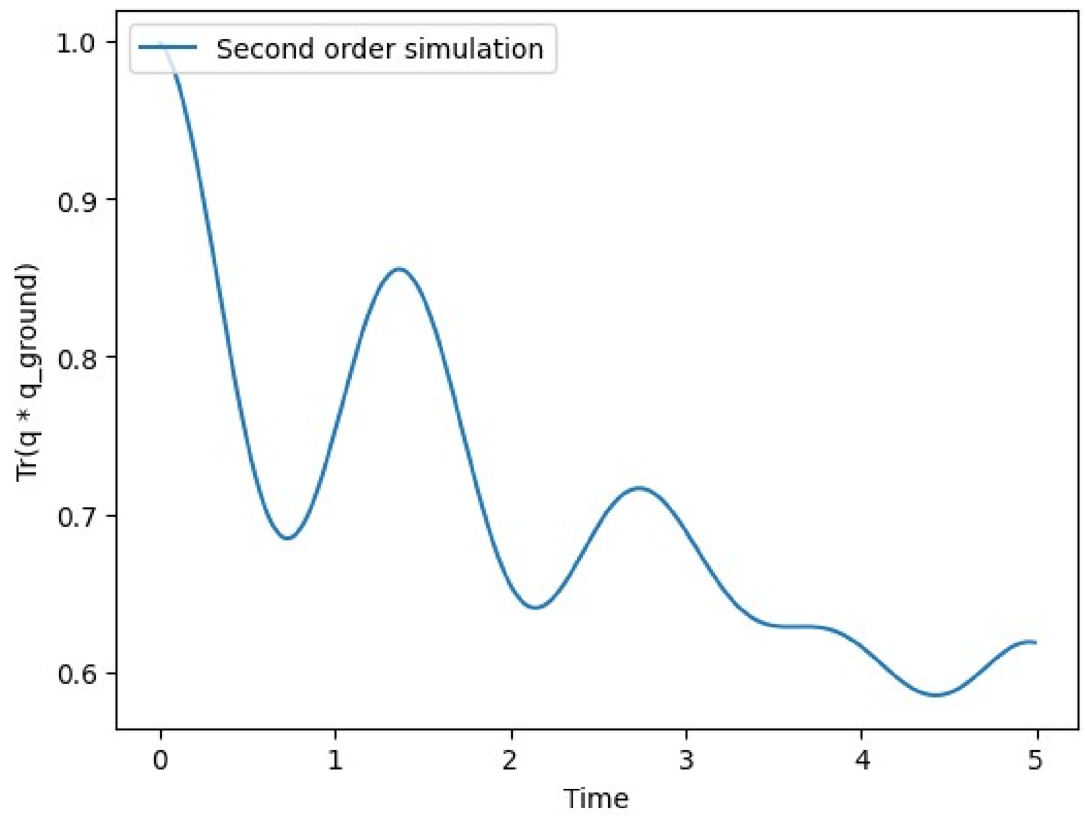
```
<matplotlib.legend.Legend at 0x7a760c38b6d0>
```



Second order approximation

```
import math
import numpy as np

ancilla = 5  # Ancillary system size
QSystem.set_nr_of_ancillas(ancilla)
ancilla_dim = np.power(2, ancilla)
total_systemSize = systemSize + ancilla  # Total system size
total_systemSize_dim = np.power(2, total_systemSize)
taylor_aproximation_order = 10

# First order scheme
psi_ancilla = 1
for i in range(ancilla):
    psi_ancilla = np.kron(psi_ancilla, qs.ket_0)
rho_ancilla = Qobj(psi_ancilla.conj().T @ psi_ancilla)

rho = starting_state
second_order_trace = []
for t in time_vec:
    # print(t)
    # Extended system, zero initialized ancilla + hamiltonian system
    system = Qobj(
        tensor(rho_ancilla, rho),
        dims=[[ancilla_dim, systemSize_dim], [ancilla_dim,
systemSize_dim]],
    )
    # First element of taylor approximation, I
    approximation = Qobj(
        qeye(ancilla_dim * systemSize_dim),
        dims=[[ancilla_dim, systemSize_dim], [ancilla_dim,
systemSize_dim]],
    )

    approximation = qs.Taylor_approximtion(
        QSystem.H_tilde_second_order(dt),
        taylor_aproximation_order,
        np.sqrt(dt),
        approximation,
    )
    evolved_system = approximation @ system @
approximation.conj().trans()

    rho = evolved_system.ptrace(1)
    second_order_trace.append((rho @ mesurement_op).tr())

plt.figure()
plt.xlabel("Time")
plt.ylabel("Tr(q * q_ground)")
plt.plot(time_vec, second_order_trace, label="Second order
simulation")
plt.legend(loc="upper left")
```

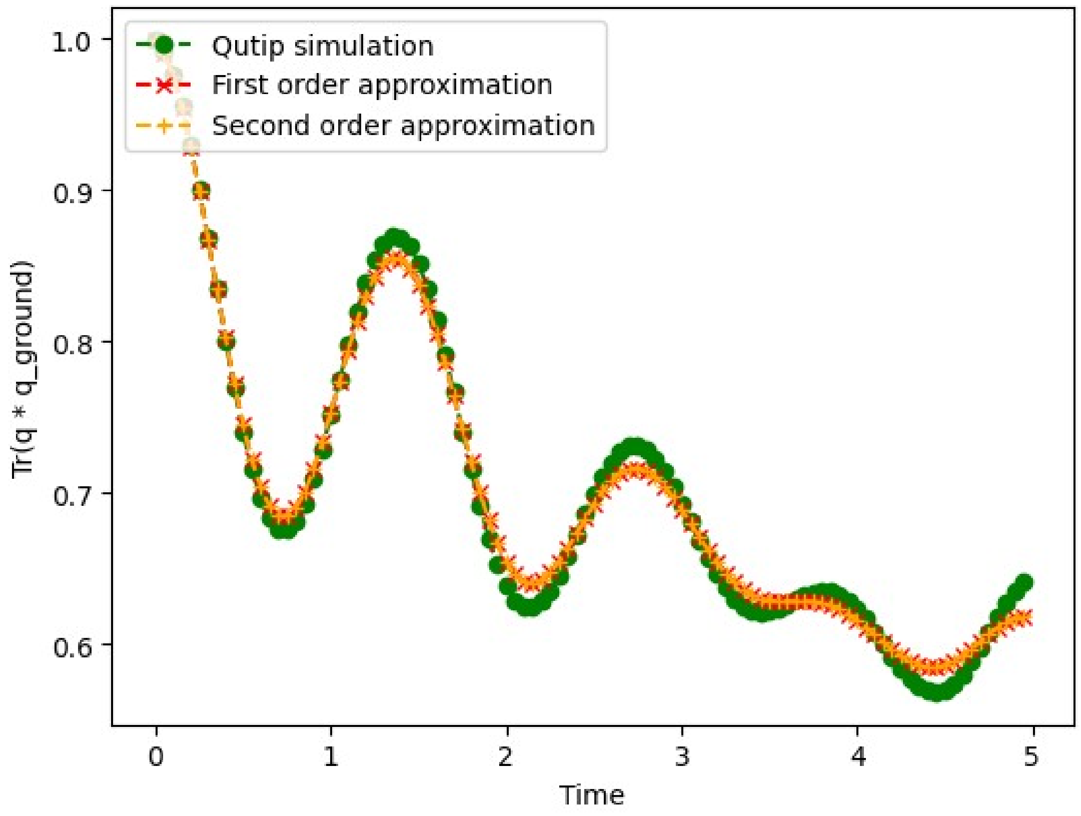
```
<matplotlib.legend.Legend at 0x7a760c2e9160>
```



```python
import matplotlib.pyplot as plt

total_nr_of_points = 100
plot_density = (int)(np.size(time_vec) / total_nr_of_points)
print(plot_density)

plt.figure()
plt.xlabel("Time")
plt.ylabel("Tr(q * q_ground)")
# plt.plot(time_vec, exact_trace, label="Exact simulation")
plt.plot(
    time_vec,
    exact_trace,
    label="Qutip simulation",
    color="green",
    marker="o",
    linestyle="dashed",
    markevery=plot_density,
)
plt.plot(
    time_vec,
    first_order_trace,
    label="First order approximation",
    color="red",
    marker="x",
    linestyle="dashed",
    markevery=plot_density,
)
plt.plot(
    time_vec,
    second_order_trace,
    label="Second order approximation",
    color="orange",
    marker="+",
    linestyle="dashed",
    markevery=plot_density,
)
plt.legend(loc="upper left")
```

```
5
```

```
<matplotlib.legend.Legend at 0x7a760c2da070>
```


Qutip simulation
First order approximation
Second order approximation
Tr(q * q_ground)
Time