Truncated Equations of Motion solver using Mathematica-Python: Documentation

Saeed A. Khan
Department of Electrical Engineering, Princeton University
(Dated: Thursday 1st February, 2024)

Documentation for Mathematica-Python code for calculation of truncated equations of motion (TEOMs) for both unconditional and conditional master equation evolution under continuous measurement.

I. MATHEMATICA CODE FOR EQUATION OF MOTION OF ARBITRARY OPERATOR EXPECTATION VALUES

As an example, we consider the master equation for the two-mode system:

$$d\hat{\rho} = -i[\hat{\mathcal{H}}, \hat{\rho}]\, dt + \kappa\mathcal{D}[\hat{a}, \hat{a}^\dagger]\hat{\rho}\, dt + \gamma\mathcal{D}[\hat{b}, \hat{b}^\dagger]\hat{\rho}\, dt + \mathcal{S}_{\rm meas}(dW)\hat{\rho} \tag{1}$$

where the system Hamiltonian is given by:

$$\hat{\mathcal{H}} = -\Delta_{da}\hat{a}^\dagger\hat{a} - \Delta_{db}\hat{b}^\dagger\hat{b} - \frac{\Lambda}{2}\hat{b}^\dagger\hat{b}^\dagger\hat{b}\hat{b} + g\left(\hat{a}^\dagger\hat{b} + \hat{a}\hat{b}^\dagger\right) + \eta\left(e^{-i\phi_\eta}\hat{a} + e^{i\phi_\eta}\hat{a}^\dagger\right) \tag{2}$$

and we have introduced a generalized form of the Lindblad dissipative superoperator: $\mathcal{D}[\hat{d}_1, \hat{d}_2]\hat{\rho} = \hat{d}_1\hat{\rho}\hat{d}_2 - \frac{1}{2}\hat{d}_2\hat{d}_1\hat{\rho} - \frac{1}{2}\hat{\rho}\hat{d}_2\hat{d}_1$. For $\hat{d}_2 = \hat{d}_1^\dagger$, the superoperator returns to the usual form of a diagonal dissipative superoperator describing linear loss. Finally, the stochastic measurement superoperator takes the form (for heterodyne measurement of canonically conjugate quadratures):

$$\mathcal{S}_{\rm meas}(dW)\hat{\rho} = \sqrt{\frac{\gamma}{2}}\left[\hat{b}\hat{\rho} + \hat{\rho}\hat{b}^\dagger - \langle\hat{b} + \hat{b}^\dagger\rangle\right]\, dW^X(t) - i\sqrt{\frac{\gamma}{2}}\left[\hat{b}\hat{\rho} - \hat{\rho}\hat{b}^\dagger - \langle\hat{b} - \hat{b}^\dagger\rangle\right]\, dW^P(t) \tag{3}$$

The equation of motion for the expectation value of an arbitrary operator $\hat{o}$ is given by:

$$d\langle\hat{o}\rangle = -i\,{\rm tr}\{[\hat{\mathcal{H}}, \hat{\rho}]\hat{o}\}\, dt + \kappa\,{\rm tr}\{\mathcal{D}[\hat{a}, \hat{a}^\dagger]\hat{\rho}\hat{o}\}\, dt + \gamma\,{\rm tr}\{\mathcal{D}[\hat{b}, \hat{b}^\dagger]\hat{\rho}\hat{o}\}\, dt + {\rm tr}\{\mathcal{S}_{\rm meas}(dW)\hat{\rho}\hat{o}\} \tag{4}$$

The Mathematica code in `TEOMSCalculator.nb` provides code to calculate individual terms on the right-hand side of this general equation, and describes three functions, `hTerm`, `dissTerm`, and `measTerm` to calculate the contributions due to Hamiltonian evolution, dissipative evolution, and stochastic measurement-induced evolution respectively. To see how these functions are defined, we expand the right-hand side above and rewrite it as:

$$\begin{aligned}
d\langle\hat{o}\rangle = {} & i\Delta_{da}\underbrace{[\langle\hat{o}\hat{a}^\dagger\hat{a}\rangle - \langle\hat{a}^\dagger\hat{a}\hat{o}\rangle]}_{\equiv\,\texttt{hTerm}[\hat{o},\hat{a}^\dagger\hat{a}]}dt + i\Delta_{db}\underbrace{[\langle\hat{o}\hat{b}^\dagger\hat{b}\rangle - \langle\hat{b}^\dagger\hat{b}\hat{o}\rangle]}_{\equiv\,\texttt{hTerm}[\hat{o},\hat{b}^\dagger\hat{b}]}dt + i\frac{\Lambda}{2}\underbrace{[\langle\hat{o}\hat{b}^\dagger\hat{b}^\dagger\hat{b}\hat{b}\rangle - \langle\hat{b}^\dagger\hat{b}^\dagger\hat{b}\hat{b}\hat{o}\rangle]}_{\equiv\,\texttt{hTerm}[\hat{o},\hat{b}^\dagger\hat{b}^\dagger\hat{b}\hat{b}]}dt \\
& - ig\underbrace{[\langle\hat{o}\hat{a}^\dagger\hat{b}\rangle - \langle\hat{a}^\dagger\hat{b}\hat{o}\rangle]}_{\equiv\,\texttt{hTerm}[\hat{o},\hat{a}^\dagger\hat{b}]}dt - ig\underbrace{[\langle\hat{o}\hat{a}\hat{b}^\dagger\rangle - \langle\hat{a}\hat{b}^\dagger\hat{o}\rangle]}_{\equiv\,\texttt{hTerm}[\hat{o},\hat{a}\hat{b}^\dagger]}dt - i\eta e^{-i\phi_\eta}\underbrace{[\langle\hat{o}\hat{a}\rangle - \langle\hat{a}\hat{o}\rangle]}_{\equiv\,\texttt{hTerm}[\hat{o},\hat{a}]}dt - i\eta e^{i\phi_\eta}\underbrace{[\langle\hat{o}\hat{a}^\dagger\rangle - \langle\hat{a}^\dagger\hat{o}\rangle]}_{\equiv\,\texttt{hTerm}[\hat{o},\hat{a}^\dagger]}dt \\
& + \kappa\underbrace{\left[\langle\hat{a}^\dagger\hat{o}\hat{a}\rangle - \frac{1}{2}\langle\hat{o}\hat{a}^\dagger\hat{a}\rangle - \frac{1}{2}\langle\hat{a}^\dagger\hat{a}\hat{o}\rangle\right]}_{\equiv\,\texttt{dissTerm}[\hat{o},\hat{a},\hat{a}^\dagger]}dt + \gamma\underbrace{\left[\langle\hat{b}^\dagger\hat{o}\hat{b}\rangle - \frac{1}{2}\langle\hat{o}\hat{b}^\dagger\hat{b}\rangle - \frac{1}{2}\langle\hat{b}^\dagger\hat{b}\hat{o}\rangle\right]}_{\equiv\,\texttt{dissTerm}[\hat{o},\hat{b},\hat{b}^\dagger]}dt \\
& + \sqrt{\frac{\gamma}{2}}\underbrace{\left[\langle\hat{b}\hat{o}\rangle + \langle\hat{b}^\dagger\hat{o}\rangle - \langle\hat{b}\rangle\langle\hat{o}\rangle - \langle\hat{b}^\dagger\rangle\langle\hat{o}\rangle\right]}_{\equiv\,\texttt{measTerm}[\hat{o},\hat{b},\hat{b}^\dagger]}dW^X(t) + i\sqrt{\frac{\gamma}{2}}\underbrace{\left[-\langle\hat{b}\hat{o}\rangle + \langle\hat{b}^\dagger\hat{o}\rangle + \langle\hat{b}\rangle\langle\hat{o}\rangle - \langle\hat{b}^\dagger\rangle\langle\hat{o}\rangle\right]}_{\equiv\,\texttt{measTerm}[\hat{o},-\hat{b},\hat{b}^\dagger]}dW^P(t)
\end{aligned} \tag{5}$$

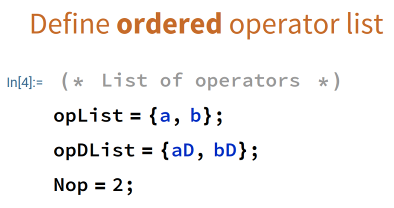
The Hamiltonian evolution function `hTerm[]` takes two arguments: the first is the operator whose expectation value is being evaluated, while the second is the specific Hamiltonian term. Currently, the function does not takes sums of operators, so these sums should be expanded before passing to `hTerm`, as shown above for terms arising due to the coupling g and the drive η.

The dissipative evolution function `dissTerm` takes three arguments: the first is the operator whose expectation value is being evaluated, while the second and third define the dissipative evolution, as shown above. The stochastic measurement evolution function `measTerm` also takes three arguments.

II. RUNNING THE MATHEMATICA CODE TO OBTAIN ARBITRARY EQUATIONS OF MOTION

Running the Mathematica code involves the following steps:

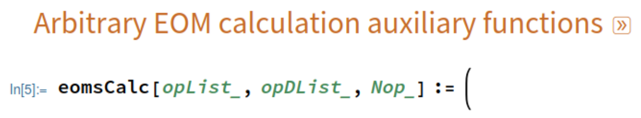
1. **Initialize number of operators defining the system modes:** We define a set of annihilation modes in `opList`, and the corresponding creation operators in `opDList`. The total number of modes is given by `Nop`. For Eq. (1), the system is described by two modes, and the initialization is shown below:



Define **ordered** operator list

```
In[4]:= (* List of operators *)
        opList = {a, b};
        opDList = {aD, bD};
        Nop = 2;
```

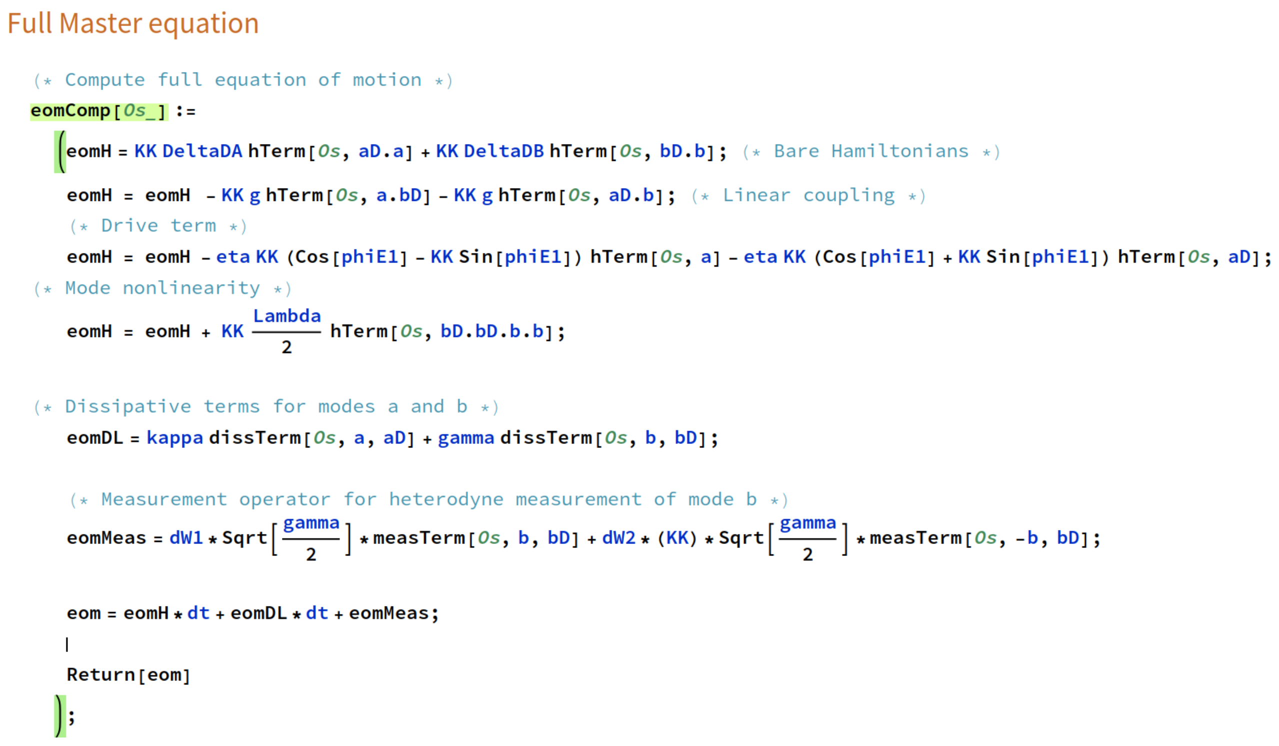
2. **Initialize auxiliary functions for arbitrary EOM calculation:** This cell defines the function `eomsCalc` that takes as argument `opList`, `opDList`, and `Nop` and returns no values. It defines other functions that account for commutation relations between operators, normal-ordering, and the specific evolution functions `hTerm`, `dissTerm`, and `measTerm`.



Arbitrary EOM calculation auxiliary functions »

```
In[5]:= eomsCalc[opList_, opDList_, Nop_] := (
```

3. **Define equation of motion for arbitrary operator:** The function `eomComp` defines the equation of motion for an arbitrary operator as defined by Eq. (5). It takes as argument the arbitrary operator $\hat{o}$ and returns the right-hand side of Eq. (5). For the example system defined by Eq. (1), the function definition takes the form:



Full Master equation

```
 (* Compute full equation of motion *)
 eomComp[Os_] :=
  (eomH = KK DeltaDA hTerm[Os, aD.a] + KK DeltaDB hTerm[Os, bD.b]; (* Bare Hamiltonians *)
   eomH = eomH - KK g hTerm[Os, a.bD] - KK g hTerm[Os, aD.b]; (* Linear coupling *)
   (* Drive term *)
   eomH = eomH - eta KK (Cos[phiE1] - KK Sin[phiE1]) hTerm[Os, a] - eta KK (Cos[phiE1] + KK Sin[phiE1]) hTerm[Os, aD];
 (* Mode nonlinearity *)
   eomH = eomH + KK Lambda/2 hTerm[Os, bD.bD.b.b];

 (* Dissipative terms for modes a and b *)
   eomDL = kappa dissTerm[Os, a, aD] + gamma dissTerm[Os, b, bD];

   (* Measurement operator for heterodyne measurement of mode b *)
   eomMeas = dW1 * Sqrt[gamma/2] * measTerm[Os, b, bD] + dW2 * (KK) * Sqrt[gamma/2] * measTerm[Os, -b, bD];

   eom = eomH * dt + eomDL * dt + eomMeas;
   |
   Return[eom]
  );
```

Each of the above terms can be directly compared to Eq. (5); note that `KK` is used as a placeholder for the complex unit i, for reasons that will become clear later. The term `eomH` describes the contribution from Hamiltonian terms, `eomDL` includes the dissipative contributions, and `eomMeas` describes the measurement contributions.

Having gone through steps 1-3, we can now *calculate equations of motion for an arbitrary operator*. An example for the system defined by Eq. (1) for the equation of motion of the expectation value $\langle\hat{a}^{\dagger}\hat{b}\rangle$ is given below:

```
In[114]:= eomsCalc[opList, opDList, Nop];
          Simplify[eomComp[aD.b]]
```

```
Out[115]= -1/2 dt (gamma + kappa) aD.b + (dW2 √gamma KK ((b - bD) aD.b - aD.b.b + aD.bD.b))/√2 + (dW1 √gamma (-((b + bD) aD.b) + aD.b.b + aD.bD.b))/√2 -
           dt KK (-b eta Cos[phiE1] + g aD.a + (DeltaDA - DeltaDB) aD.b - g bD.b - Lambda aD.bD.b.b + b eta KK Sin[phiE1])
```

The above can be verified as an exercise. It is important to note that in Mathematica, we make use of the tensor product defined by the dot ('.') to retain the order of operators and prevent re-ordering of non-commuting operators. The appropriate re-ordering rules (defined by commutation relations) are defined in the function `eomsCalc`.

III. OBTAINING EQUATIONS OF MOTION FOR TRUNCATED CUMULANTS

Section I describes the part of the Mathematica code that enables calculation of equations of motion for arbitrary operator expectation values. However, such equations of motion exhibit an unending hierarchy, thus not forming a closed set. The Truncated Equations of Motion (TEOMs) arise from an approach to obtain a closed set of equations for system variables. This approach is based on the following simple observation: for specific nonlinearities and excitation models, quantum dynamics may be more efficiently described in alternative bases. One such basis is the use of cumulants, which provide a very efficient description of Gaussian states. Such a description is broadly valid for **weakly-nonlinear bosonic systems under coherent excitation**. The dynamics of such systems are generally well-described by cumulants of at most second-order; all higher-order cumulants vanish, even though higher-order moments for such systems will generally be nonzero.

The general approach to TEOMs involves first choosing an order N_c of cumulants to retain. For Gaussian states, we retain only cumulants up to second-order, as mentioned above, so $N_c = 2$. Then, we must be able to obtain equations of motion for moments up to N_c. This requirement is met by the code described in Section I, so we follow steps 1-3 first. Lastly, we must rewrite the obtained equations in terms of cumulants of up to order N_c, setting all higher-order cumulants to zero.

Here we mention important restrictions on the truncation code. First, the current version of the code currently enables truncation only to $N_c = 2$ and not higher. Another important variable is the order of nonlinearity N_{NL} in the system model, Eq. (1). This determines the order of moments that appear in dynamical equations for cumulants of up to order N_c; for example the Kerr-nonlinearity of Eq. (1) will generally lead to second-order moments coupling to fourth-order moments, as shown above in the dynamical equation for $\langle\hat{a}^{\dagger}\hat{b}\rangle$. However a higher-order nonlinearity than the Kerr will lead to second-order moments coupling to moments higher than fourth-order, which must be rewritten in terms of lower-order cumulants and truncated appropriately. The current implementation of the code is for $N_c = 2$ and $N_{\mathrm{NL}} \leq 4$ (Four and three wave mixing, as well as all quadratic Hamiltonians). The Mathematica code carrying out the truncation steps is described below.

4. **Initialize auxiliary functions for cumulant EOMs and truncation:** We define the function `trunc2`, which takes as argument `opList`, `opDList`, and `Nop`, returns no values, and should be run as-is. This function defines other functions that express moments in terms of cumulants, and define a replacement list to set cumulants of order higher than $N_c = 2$ to zero.

Cumulant expression and truncation auxiliary functions »

```
In[∘]:= trunc2[opList_, opDList_, Nop_] := (
```

5. **Compute TEOMs:** Finally, the function `teomsCalc` makes use of `eomComp` and `trunc2` to calculate equations of motion for cumulants up to second-order, and truncating to the same order to obtain TEOMs describing the system of interest. `teomsCalc` takes as argument `opList`, `opDList`, `Nop`, and `Nw`. The final argument is the number of independent stochastic increments dW included in the stochastic measurement superoperator, and must be chosen consistently with the definition of the `eomComp` function. For unconditional evolution, `Nw` = 0. `teomsCalc` returns 4 arguments: (i) List of unique variables (first and second-order cumulants) whose TEOMs are being calculated, (ii) List of unique variable *names as strings* for later use, (iii) TEOMs for first-order cumulants, and (iv) TEOMs for second-order cumulants.

Define function to calculate EOMs and truncate to second-order cumulants »

```
teomsCalc[opList_, opDList_, Nop_, Nw_] := (
    (* --------------------------------------------------------- *)
    (* Function to calculate EOMs, truncating to second-order cumulants. Returns
       array of variables, variable names, and EOMs for first and second-order cumulants *)
```

Use of the `teomsCalc` function is straightforward, and is shown below:

Function to calculate TEOMs in terms of cumulants

```
In[106]:= (* Number of independent stochastic terms *)
          Nw = 2;
          returnA = teomsCalc[opList, opDList, Nop, Nw];

In[108]:= (* First element of returned array is list of variable names *)
          varNames = returnA[[1]]

Out[108]= {a, b, aD, bD, Caa, Cab, Cbb, CaDa, CaDb, CbDa, CbDb, CaDaD, CaDbD, CbDbD}

In[109]:= (* Second element of returned array is list of variable names as strings *)
          varNamesString = returnA[[2]]

Out[109]= {a, b, aD, bD, Caa, Cab, Cbb, CaDa, CaDb, CbDa, CbDb, CaDaD, CaDbD, CbDbD}

In[110]:= (* Third element of returned array is list of EOMs for first-order cumulants *)
          foL = returnA[[3]]
```

Out[110]=

$$\Big\{\frac{(\mathrm{Cab}+\mathrm{CbDa})\ \mathrm{dW1}\sqrt{\mathrm{gamma}}}{\sqrt{2}}-\frac{\mathrm{a\ dt\ kappa}}{2}-\frac{(\mathrm{Cab}-\mathrm{CbDa})\ \mathrm{dW2}\sqrt{\mathrm{gamma}}\ \mathrm{KK}}{\sqrt{2}}-\mathrm{dt\ KK}\,(-\mathrm{a\ DeltaDA}+\mathrm{b\ g}+\mathrm{eta\ Cos[phiE1]}+\mathrm{eta\ KK\ Sin[phiE1]}),$$
$$-\frac{1}{2}\mathrm{b\ dt\ gamma}+\mathrm{b\ DeltaDB\ dt\ KK}-\mathrm{a\ dt\ g\ KK}+\mathrm{b}^2\,\mathrm{bD\ dt\ KK\ Lambda}+\mathrm{CbDb}\left(\frac{\mathrm{dW1}\sqrt{\mathrm{gamma}}}{\sqrt{2}}+\frac{\mathrm{dW2}\sqrt{\mathrm{gamma}}\ \mathrm{KK}}{\sqrt{2}}+2\,\mathrm{b\ dt\ KK\ Lambda}\right)+\mathrm{Cbb}\left(\frac{\mathrm{dW1}\sqrt{\mathrm{gamma}}}{\sqrt{2}}-\frac{\mathrm{dW2}\sqrt{\mathrm{gamma}}\ \mathrm{KK}}{\sqrt{2}}+\mathrm{bD\ dt\ KK\ Lambda}\right),$$
$$\frac{(\mathrm{CaDb}+\mathrm{CaDbD})\ \mathrm{dW1}\sqrt{\mathrm{gamma}}}{\sqrt{2}}-\frac{\mathrm{aD\ dt\ kappa}}{2}-\frac{(\mathrm{CaDb}-\mathrm{CaDbD})\ \mathrm{dW2}\sqrt{\mathrm{gamma}}\ \mathrm{KK}}{\sqrt{2}}+\mathrm{dt\ KK}\,(-\mathrm{aD\ DeltaDA}+\mathrm{bD\ g}+\mathrm{eta\ Cos[phiE1]}-\mathrm{eta\ KK\ Sin[phiE1]}),$$
$$\frac{1}{2}\left(\sqrt{2}\ \mathrm{CbDb\ dW1}\sqrt{\mathrm{gamma}}-\mathrm{CbDb\ KK}\left(\sqrt{2}\ \mathrm{dW2}\sqrt{\mathrm{gamma}}+4\,\mathrm{bD\ dt\ Lambda}\right)-\right.$$
$$\left.\mathrm{dt}\left(-2\,\mathrm{aD\ g\ KK}+\mathrm{bD}\,(\mathrm{gamma}+2\,\mathrm{DeltaDB\ KK})+2\,\mathrm{b\ bD}^2\,\mathrm{KK\ Lambda}\right)+\mathrm{CbDbD}\left(\sqrt{2}\ \mathrm{dW1}\sqrt{\mathrm{gamma}}+\sqrt{2}\ \mathrm{dW2}\sqrt{\mathrm{gamma}}\ \mathrm{KK}-2\,\mathrm{b\ dt\ KK\ Lambda}\right)\right)\Big\}$$

```
In[111]:= (* Fourth element of returned array is list of EOMs for second-order cumulants *)
          soL = returnA[[4]]
```

The first two elements of the returned array `returnA` are seen to be the first and second-order cumulants describing the two-mode system; there are 14 distinct such cumulants, as expected. Note that `varNameString` appears the same as `varNames`, but is actually an array of strings; the distinction will become clear later. The third element of `returnA` shows the four TEOMs for the first-order cumulants. The fourth array returned is not shown for brevity, but is simply the right-hand side for TEOMs calculated for the ten second-order cumulants.

newpage

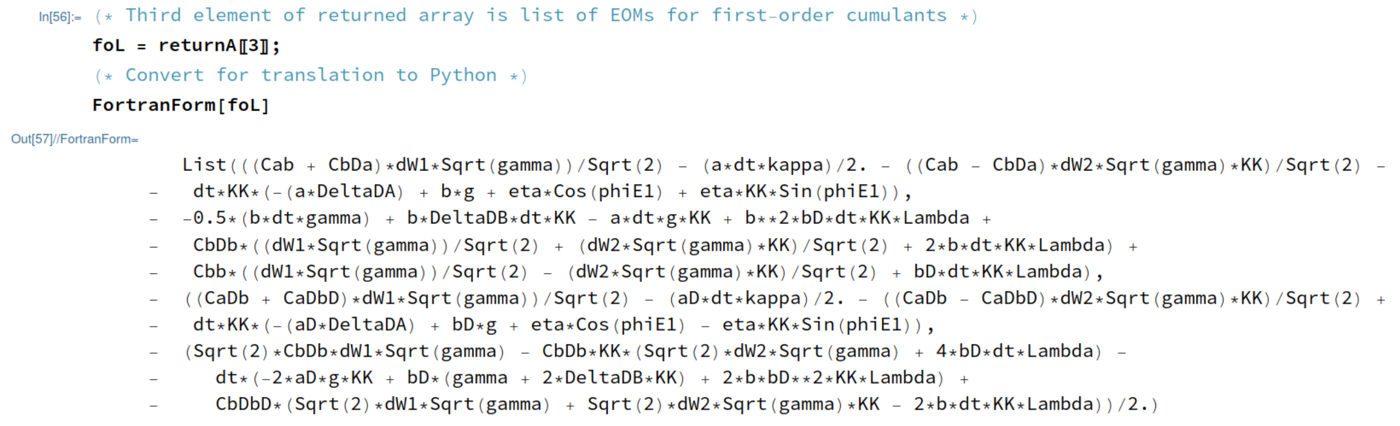
IV. TRANSLATION TO PYTHON CODE

Solving the TEOMs is computationally expensive, and while Mathematica is well-suited to obtaining the TEOMs, we implement the actual simulation of TEOMs in Python. This requires translating the obtained TEOMs to Python code. While Mathematica does not have a simple '`ToPython`' function, it provides a function labelled `FortranForm` that facilitates part of this translation. However, certain additional replacements must still be performed manually. These are indicated in Table I.

FortranForm of Mathematica code	Python
List()	[]
Cos	np.cos
Sin	np.sin
Sqrt	np.sqrt
KK	1j
-(space)(space)	(space)

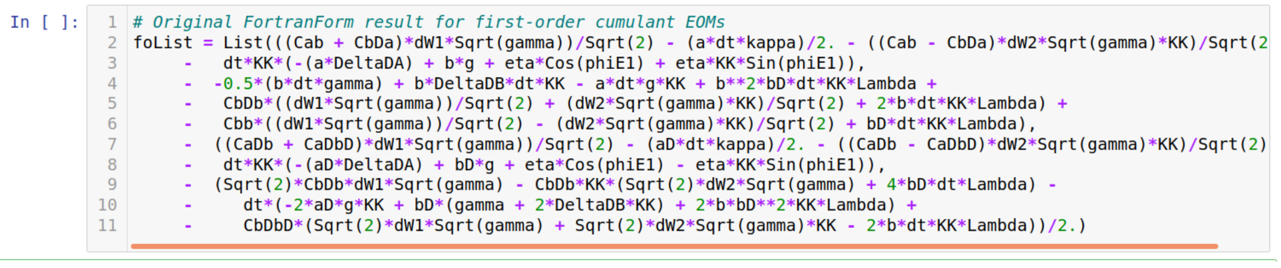
TABLE I. Replacement rules for conversion from Mathematica FortranForm[] to Python.

To see how these replacements are performed in practice, we start by converting TEOMs for first-order cumulants as shown below using `FortranForm`:



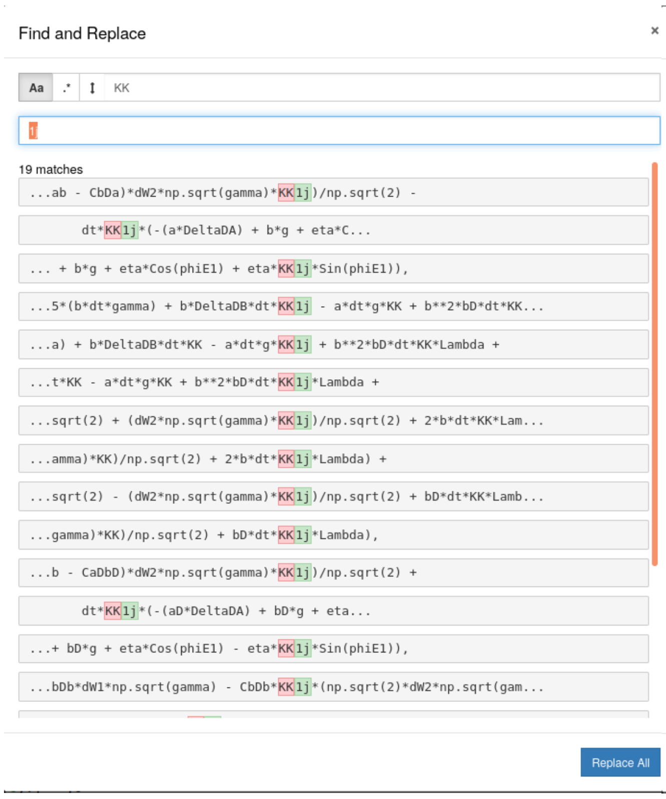
```
In[56]:= (* Third element of returned array is list of EOMs for first-order cumulants *)
         foL = returnA[[3]];
         (* Convert for translation to Python *)
         FortranForm[foL]
Out[57]//FortranForm=
          List(((Cab + CbDa)*dW1*Sqrt(gamma))/Sqrt(2) - (a*dt*kappa)/2. - ((Cab - CbDa)*dW2*Sqrt(gamma)*KK)/Sqrt(2) -
     -     dt*KK*(-(a*DeltaDA) + b*g + eta*Cos(phiE1) + eta*KK*Sin(phiE1)),
     -    -0.5*(b*dt*gamma) + b*DeltaDB*dt*KK - a*dt*g*KK + b**2*bD*dt*KK*Lambda +
     -     CbDb*((dW1*Sqrt(gamma))/Sqrt(2) + (dW2*Sqrt(gamma)*KK)/Sqrt(2) + 2*b*dt*KK*Lambda) +
     -     Cbb*((dW1*Sqrt(gamma))/Sqrt(2) - (dW2*Sqrt(gamma)*KK)/Sqrt(2) + bD*dt*KK*Lambda),
     -    ((CaDb + CaDbD)*dW1*Sqrt(gamma))/Sqrt(2) - (aD*dt*kappa)/2. - ((CaDb - CaDbD)*dW2*Sqrt(gamma)*KK)/Sqrt(2) +
     -     dt*KK*(-(aD*DeltaDA) + bD*g + eta*Cos(phiE1) - eta*KK*Sin(phiE1)),
     -    (Sqrt(2)*CbDb*dW1*Sqrt(gamma) - CbDb*KK*(Sqrt(2)*dW2*Sqrt(gamma) + 4*bD*dt*Lambda) -
     -       dt*(-2*aD*g*KK + bD*(gamma + 2*DeltaDB*KK) + 2*b*bD**2*KK*Lambda) +
     -       CbDbD*(Sqrt(2)*dW1*Sqrt(gamma) + Sqrt(2)*dW2*Sqrt(gamma)*KK - 2*b*dt*KK*Lambda))/2.)
```

Similar conversions are performed for all the returned arrays in `returnA` and are not shown here for brevity (they are included in the accompanying example Mathematica notebook, `TEOMSCalculator.nb`). The above can be copied *as plain text* and transplanted to a Python script or Jupyter notebook. This is shown below for the first-order cumulants TEOMs:



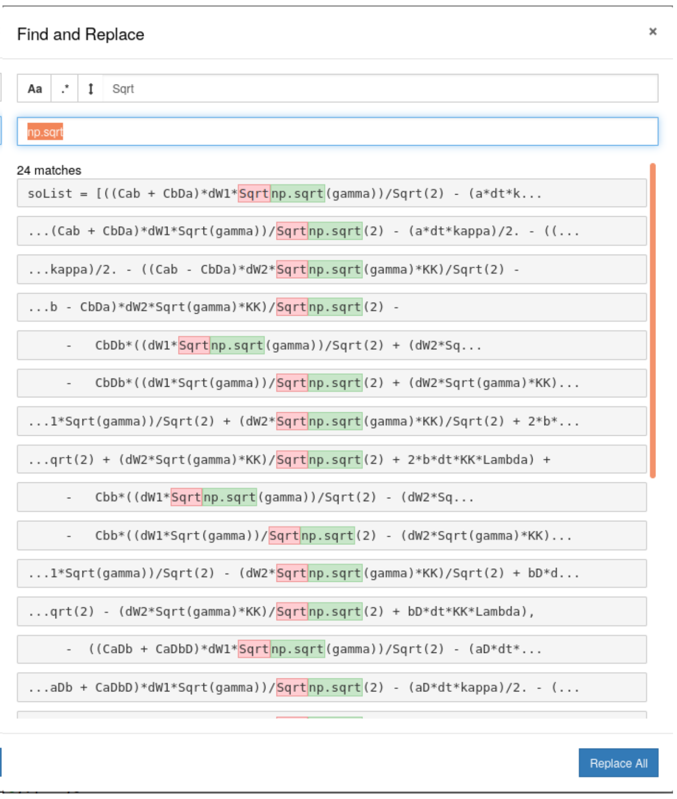
```
In [ ]: # Original FortranForm result for first-order cumulant EOMs
        foList = List(((Cab + CbDa)*dW1*Sqrt(gamma))/Sqrt(2) - (a*dt*kappa)/2. - ((Cab - CbDa)*dW2*Sqrt(gamma)*KK)/Sqrt(2
             -   dt*KK*(-(a*DeltaDA) + b*g + eta*Cos(phiE1) + eta*KK*Sin(phiE1)),
             -  -0.5*(b*dt*gamma) + b*DeltaDB*dt*KK - a*dt*g*KK + b**2*bD*dt*KK*Lambda +
             -   CbDb*((dW1*Sqrt(gamma))/Sqrt(2) + (dW2*Sqrt(gamma)*KK)/Sqrt(2) + 2*b*dt*KK*Lambda) +
             -   Cbb*((dW1*Sqrt(gamma))/Sqrt(2) - (dW2*Sqrt(gamma)*KK)/Sqrt(2) + bD*dt*KK*Lambda),
             -  ((CaDb + CaDbD)*dW1*Sqrt(gamma))/Sqrt(2) - (aD*dt*kappa)/2. - ((CaDb - CaDbD)*dW2*Sqrt(gamma)*KK)/Sqrt(2)
             -   dt*KK*(-(aD*DeltaDA) + bD*g + eta*Cos(phiE1) - eta*KK*Sin(phiE1)),
             -  (Sqrt(2)*CbDb*dW1*Sqrt(gamma) - CbDb*KK*(Sqrt(2)*dW2*Sqrt(gamma) + 4*bD*dt*Lambda) -
             -     dt*(-2*aD*g*KK + bD*(gamma + 2*DeltaDB*KK) + 2*b*bD**2*KK*Lambda) +
             -     CbDbD*(Sqrt(2)*dW1*Sqrt(gamma) + Sqrt(2)*dW2*Sqrt(gamma)*KK - 2*b*dt*KK*Lambda))/2.)
```

Pasted code for the other elements is shown in the accompanying Jupyter notebook, `TEOMSExample.ipynb`. The above code will throw errors if executed in Python; we must make the replacements in Table I. Examples of these replacements are shown below. The left panel shows the replacement of `KK` by the complex unit in Python, `1j`. The right panel shows the replacement of `Sqrt` by `np.sqrt`; similar replacements work for other functions like `Cos` etc., as indicated in Table I.



19 matches

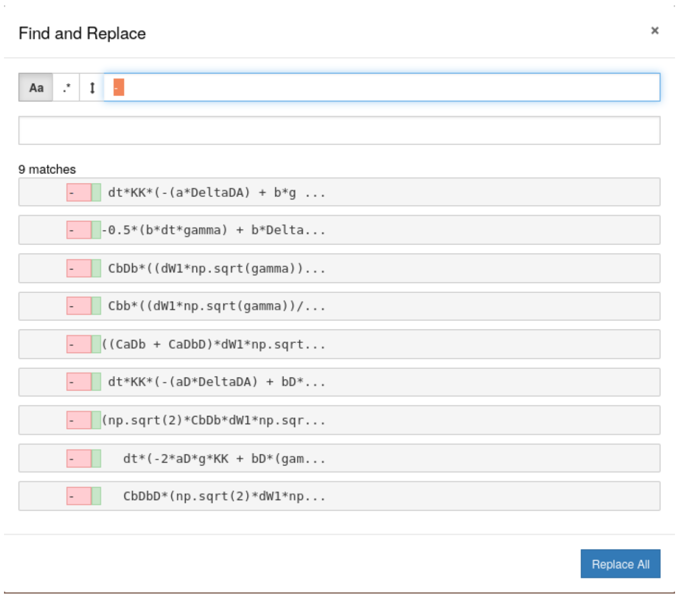
```
...ab - CbDa)*dW2*np.sqrt(gamma)*KK1j)/np.sqrt(2) -
        dt*KK1j*(-(a*DeltaDA) + b*g + eta*C...
... + b*g + eta*Cos(phiE1) + eta*KK1j*Sin(phiE1)),
...5*(b*dt*gamma) + b*DeltaDB*dt*KK1j - a*dt*g*KK + b**2*bD*dt*KK...
...a) + b*DeltaDB*dt*KK - a*dt*g*KK1j + b**2*bD*dt*KK*Lambda +
...t*KK - a*dt*g*KK + b**2*bD*dt*KK1j*Lambda +
...sqrt(2) + (dW2*np.sqrt(gamma)*KK1j)/np.sqrt(2) + 2*b*dt*KK*Lam...
...amma)*KK)/np.sqrt(2) + 2*b*dt*KK1j*Lambda) +
...sqrt(2) - (dW2*np.sqrt(gamma)*KK1j)/np.sqrt(2) + bD*dt*KK*Lamb...
...gamma)*KK)/np.sqrt(2) + bD*dt*KK1j*Lambda),
...b - CaDbD)*dW2*np.sqrt(gamma)*KK1j)/np.sqrt(2) +
        dt*KK1j*(-(aD*DeltaDA) + bD*g + eta...
...+ bD*g + eta*Cos(phiE1) - eta*KK1j*Sin(phiE1)),
...bDb*dW1*np.sqrt(gamma) - CbDb*KK1j*(np.sqrt(2)*dW2*np.sqrt(gam...
```



24 matches

```
soList = [((Cab + CbDa)*dW1*Sqrtnp.sqrt(gamma))/Sqrt(2) - (a*dt*k...
...(Cab + CbDa)*dW1*Sqrt(gamma))/Sqrtnp.sqrt(2) - (a*dt*kappa)/2. - ((...
...kappa)/2. - ((Cab - CbDa)*dW2*Sqrtnp.sqrt(gamma)*KK)/Sqrt(2) -
...b - CbDa)*dW2*Sqrt(gamma)*KK)/Sqrtnp.sqrt(2) -
   -   CbDb*((dW1*Sqrtnp.sqrt(gamma))/Sqrt(2) + (dW2*Sq...
   -   CbDb*((dW1*Sqrt(gamma))/Sqrtnp.sqrt(2) + (dW2*Sqrt(gamma)*KK)...
...1*Sqrt(gamma))/Sqrt(2) + (dW2*Sqrtnp.sqrt(gamma)*KK)/Sqrt(2) + 2*b*...
...qrt(2) + (dW2*Sqrt(gamma)*KK)/Sqrtnp.sqrt(2) + 2*b*dt*KK*Lambda) +
   -   Cbb*((dW1*Sqrtnp.sqrt(gamma))/Sqrt(2) - (dW2*Sq...
   -   Cbb*((dW1*Sqrt(gamma))/Sqrtnp.sqrt(2) - (dW2*Sqrt(gamma)*KK)...
...1*Sqrt(gamma))/Sqrt(2) - (dW2*Sqrtnp.sqrt(gamma)*KK)/Sqrt(2) + bD*d...
...qrt(2) - (dW2*Sqrt(gamma)*KK)/Sqrtnp.sqrt(2) + bD*dt*KK*Lambda),
   -  ((CaDb + CaDbD)*dW1*Sqrtnp.sqrt(gamma))/Sqrt(2) - (aD*dt*...
...aDb + CaDbD)*dW1*Sqrt(gamma))/Sqrtnp.sqrt(2) - (aD*dt*kappa)/2. - (...
```

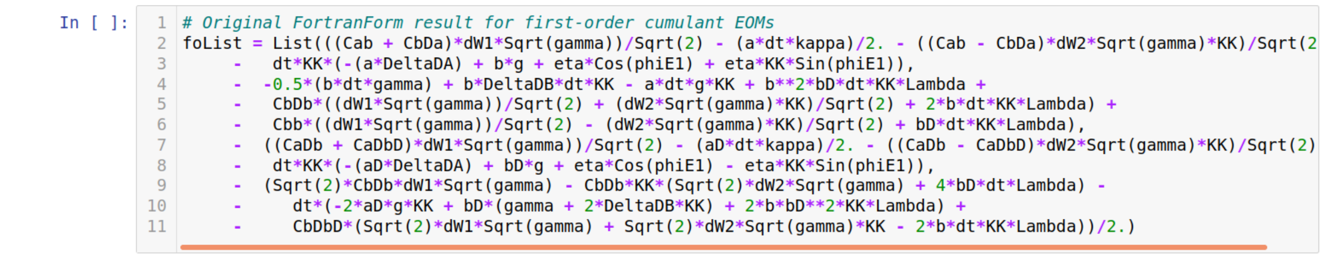
The next panel shows the replacement of -(space)(space) by (space), which is very important, since Python interprets the '-' as a minus sign in the TEOMs, while in Fortran this is simply an indicator of 'next line'.



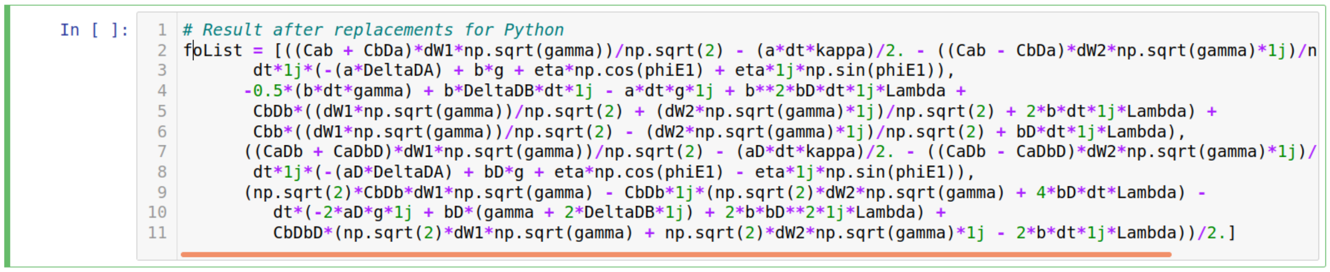
9 matches

```
      -  dt*KK*(-(a*DeltaDA) + b*g ...
      - -0.5*(b*dt*gamma) + b*Delta...
      -  CbDb*((dW1*np.sqrt(gamma))...
      -  Cbb*((dW1*np.sqrt(gamma))/...
      - ((CaDb + CaDbD)*dW1*np.sqrt...
      -  dt*KK*(-(aD*DeltaDA) + bD*...
      - (np.sqrt(2)*CbDb*dW1*np.sqr...
      -    dt*(-2*aD*g*KK + bD*(gam...
      -    CbDbD*(np.sqrt(2)*dW1*np...
```

Implementing all the rules of Table I, we obtain the transformed version of TEOMs for first-order cumulants; the original and transformed lists are shown below:

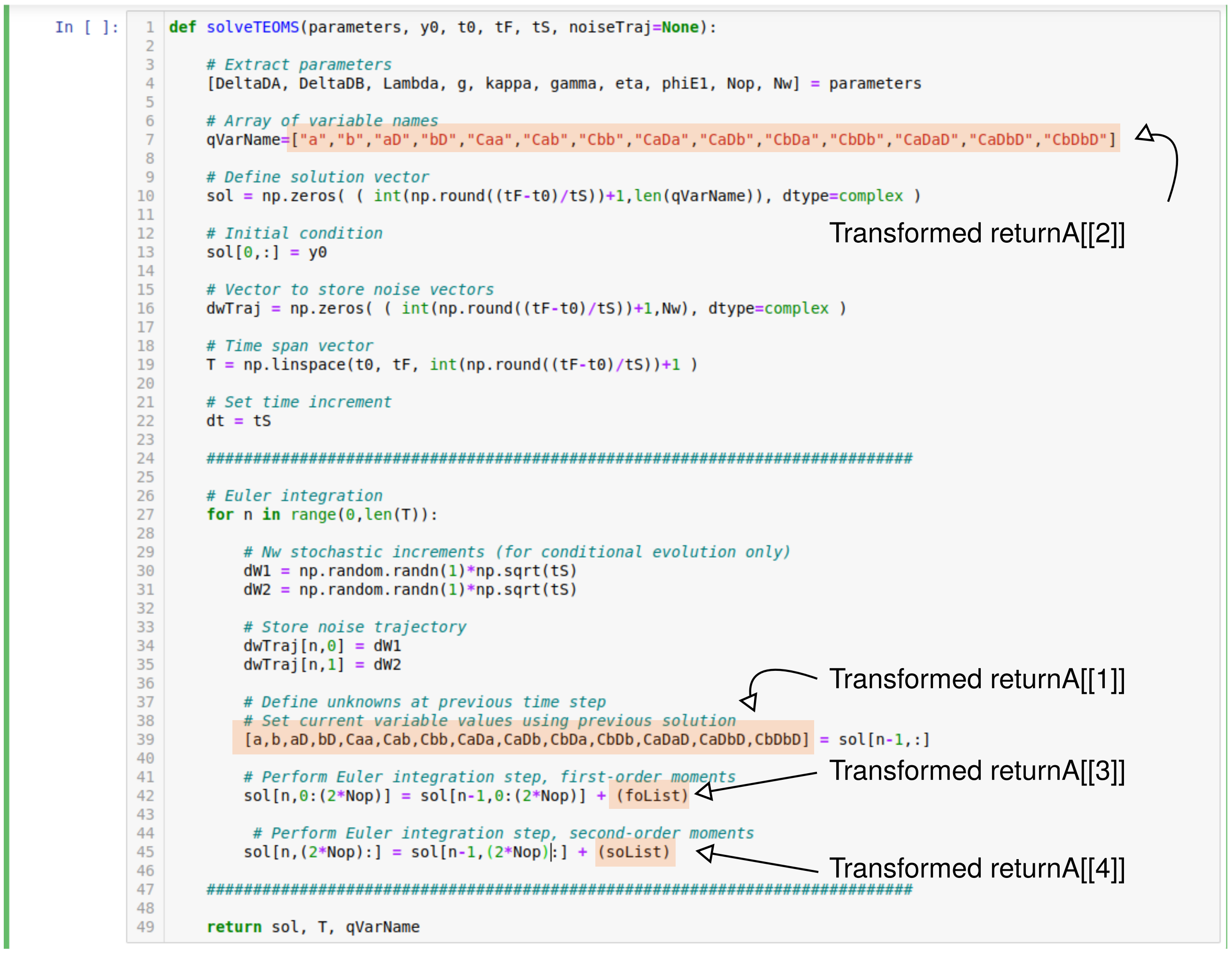


```
In [ ]:
# Original FortranForm result for first-order cumulant EOMs
foList = List(((Cab + CbDa)*dW1*Sqrt(gamma))/Sqrt(2) - (a*dt*kappa)/2. - ((Cab - CbDa)*dW2*Sqrt(gamma)*KK)/Sqrt(2
     -   dt*KK*(-(a*DeltaDA) + b*g + eta*Cos(phiE1) + eta*KK*Sin(phiE1)),
     -  -0.5*(b*dt*gamma) + b*DeltaDB*dt*KK - a*dt*g*KK + b**2*bD*dt*KK*Lambda +
     -   CbDb*((dW1*Sqrt(gamma))/Sqrt(2) + (dW2*Sqrt(gamma)*KK)/Sqrt(2) + 2*b*dt*KK*Lambda) +
     -   Cbb*((dW1*Sqrt(gamma))/Sqrt(2) - (dW2*Sqrt(gamma)*KK)/Sqrt(2) + bD*dt*KK*Lambda),
     -  ((CaDb + CaDbD)*dW1*Sqrt(gamma))/Sqrt(2) - (aD*dt*kappa)/2. - ((CaDb - CaDbD)*dW2*Sqrt(gamma)*KK)/Sqrt(2)
     -   dt*KK*(-(aD*DeltaDA) + bD*g + eta*Cos(phiE1) - eta*KK*Sin(phiE1)),
     -  (Sqrt(2)*CbDb*dW1*Sqrt(gamma) - CbDb*KK*(Sqrt(2)*dW2*Sqrt(gamma) + 4*bD*dt*Lambda) -
     -     dt*(-2*aD*g*KK + bD*(gamma + 2*DeltaDB*KK) + 2*b*bD**2*KK*Lambda) +
     -     CbDbD*(Sqrt(2)*dW1*Sqrt(gamma) + Sqrt(2)*dW2*Sqrt(gamma)*KK - 2*b*dt*KK*Lambda))/2.)
```



```
In [ ]:
# Result after replacements for Python
foList = [((Cab + CbDa)*dW1*np.sqrt(gamma))/np.sqrt(2) - (a*dt*kappa)/2. - ((Cab - CbDa)*dW2*np.sqrt(gamma)*1j)/n
       dt*1j*(-(a*DeltaDA) + b*g + eta*np.cos(phiE1) + eta*1j*np.sin(phiE1)),
      -0.5*(b*dt*gamma) + b*DeltaDB*dt*1j - a*dt*g*1j + b**2*bD*dt*1j*Lambda +
       CbDb*((dW1*np.sqrt(gamma))/np.sqrt(2) + (dW2*np.sqrt(gamma)*1j)/np.sqrt(2) + 2*b*dt*1j*Lambda) +
       Cbb*((dW1*np.sqrt(gamma))/np.sqrt(2) - (dW2*np.sqrt(gamma)*1j)/np.sqrt(2) + bD*dt*1j*Lambda),
      ((CaDb + CaDbD)*dW1*np.sqrt(gamma))/np.sqrt(2) - (aD*dt*kappa)/2. - ((CaDb - CaDbD)*dW2*np.sqrt(gamma)*1j)/
       dt*1j*(-(aD*DeltaDA) + bD*g + eta*np.cos(phiE1) - eta*1j*np.sin(phiE1)),
      (np.sqrt(2)*CbDb*dW1*np.sqrt(gamma) - CbDb*1j*(np.sqrt(2)*dW2*np.sqrt(gamma) + 4*bD*dt*Lambda) -
         dt*(-2*aD*g*1j + bD*(gamma + 2*DeltaDB*1j) + 2*b*bD**2*1j*Lambda) +
         CbDbD*(np.sqrt(2)*dW1*np.sqrt(gamma) + np.sqrt(2)*dW2*np.sqrt(gamma)*1j - 2*b*dt*1j*Lambda))/2.]
```

We can now transplant the TEOMs into a suitable Python integrator. We have calculated the right-hand side of Eq. (5) in a form that is suitable for lowest-order Euler integration. We provide below a simple form of the TEOM evolution function:



```
In [ ]:
def solveTEOMS(parameters, y0, t0, tF, tS, noiseTraj=None):

    # Extract parameters
    [DeltaDA, DeltaDB, Lambda, g, kappa, gamma, eta, phiE1, Nop, Nw] = parameters

    # Array of variable names
    qVarName=["a","b","aD","bD","Caa","Cab","Cbb","CaDa","CaDb","CbDa","CbDb","CaDaD","CaDbD","CbDbD"]

    # Define solution vector
    sol = np.zeros( ( int(np.round((tF-t0)/tS))+1,len(qVarName)), dtype=complex )

    # Initial condition
    sol[0,:] = y0

    # Vector to store noise vectors
    dwTraj = np.zeros( ( int(np.round((tF-t0)/tS))+1,Nw), dtype=complex )

    # Time span vector
    T = np.linspace(t0, tF, int(np.round((tF-t0)/tS))+1 )

    # Set time increment
    dt = tS

    ############################################################################

    # Euler integration
    for n in range(0,len(T)):

        # Nw stochastic increments (for conditional evolution only)
        dW1 = np.random.randn(1)*np.sqrt(tS)
        dW2 = np.random.randn(1)*np.sqrt(tS)

        # Store noise trajectory
        dwTraj[n,0] = dW1
        dwTraj[n,1] = dW2

        # Define unknowns at previous time step
        # Set current variable values using previous solution
        [a,b,aD,bD,Caa,Cab,Cbb,CaDa,CaDb,CbDa,CbDb,CaDaD,CaDbD,CbDbD] = sol[n-1,:]

        # Perform Euler integration step, first-order moments
        sol[n,0:(2*Nop)] = sol[n-1,0:(2*Nop)] + (foList)

         # Perform Euler integration step, second-order moments
        sol[n,(2*Nop):] = sol[n-1,(2*Nop):] + (soList)
    ############################################################################

    return sol, T, qVarName
```

The above function integrates the TEOMs (both conditional and unconditional) using a lowest-order Euler method.