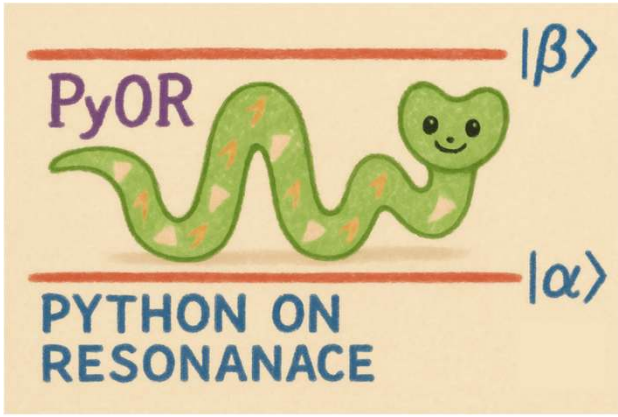
PyOR: A Versatile Magnetic Resonance Simulator for Learning and Teaching

Global NMR Discussion Meeting
September 9th, 2025

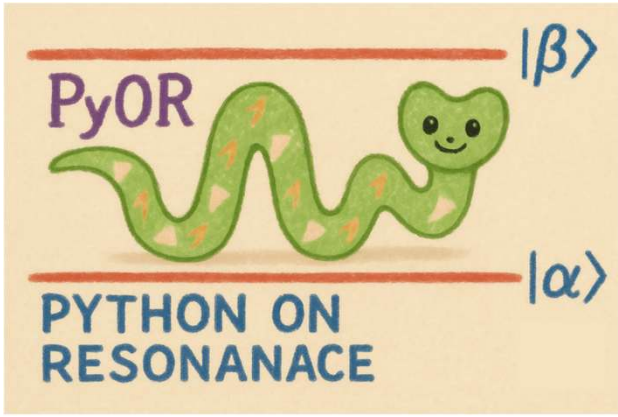
Vineeth Francis THALAKOTTOOR JOSE CHACKO
École Normale Supérieure Paris

PyOR



- **PY**thon **O**n **R**esonance
- A versatile magnetic resonance simulator: Liquid and Solid*, based on Python
- Genesis: "*Let's do it (multi-mode maser) semi-quantum mechanically...*", said Daniel.
- A hobby since 2024: A 'baby' compared to Gamma, SpinEvolution, Spinach, SIMPSON,…
  - Beta version: 24.08.2024 (https://github.com/VThalakottoor/PyOR-Jeener-Beta)
  - First Version: 18.04.2025

PyOR



- **PY**thon **O**n **R**esonance
- A versatile magnetic resonance simulator: Liquid and Solid*, based on Python
- Genesis: "*Let's do it (multi-mode maser) semi-quantum mechanically...*", said Daniel.
- A ho
  Spin
  - B
    J
  - F

PHYSICAL REVIEW LETTERS **133,** 158001 (2024)

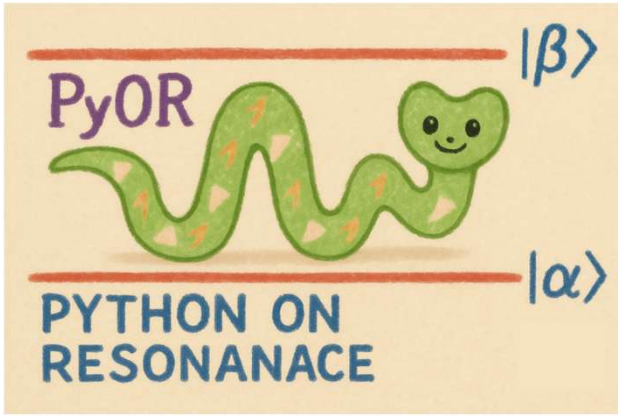
OR-

Multimode Masers of Thermally Polarized Nuclear Spins in Solution NMR

Vineeth Francis Thalakottoor Jose Chacko,[1] Alain Louis-Joseph,[2] and Daniel Abergel[1,*]
[1]*Laboratoire des Biomolécules, LBM, Département de Chimie, Ecole Normale Supérieure, PSL University, Sorbonne Université, CNRS, 75005 Paris, France*
[2]*Laboratoire de Physique de la Matière Condensée, UMR 7643, CNRS, École Polytechnique, IPP 91120 Palaiseau, France*

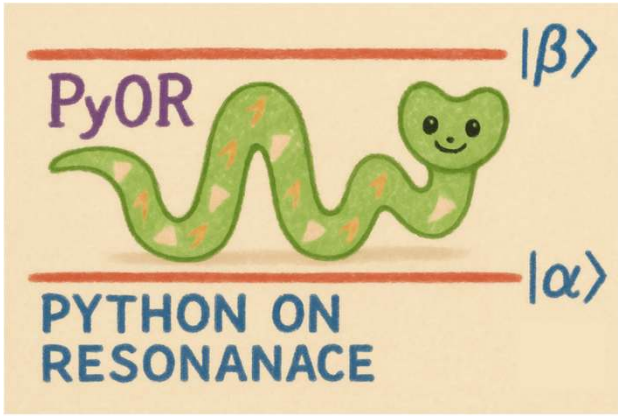
(Received 16 April 2024; revised 5 August 2024; accepted 5 September 2024; published 10 October 2024)

PyOR



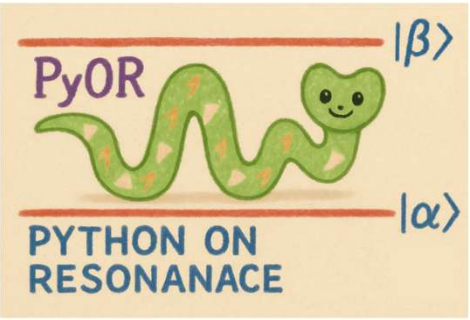
- **PY**thon **O**n **R**esonance
- A versatile magnetic resonance simulator: Liquid and Solid*, based on Python
- Genesis: "*Let's do it (multi-mode maser) semi-quantum mechanically...*", said Daniel.
- A hobby since 2024: A 'baby' compared to Gamma, SpinEvolution, Spinach, SIMPSON, SpinDynamica, …
  - Beta version: 24.08.2024 (https://github.com/VThalakottoor/PyOR-Jeener-Beta)
  - First Version: 18.04.2025

PyOR



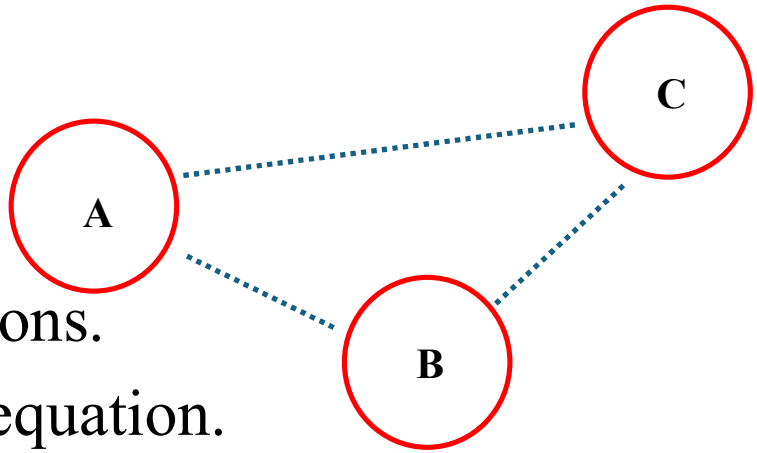
- Motto: "Everybody can simulate Magnetic Resonance"
- Motivation and Purpose:
  - Why this package?: Nonlinear NMR – Radiation damping and MASER
  - A package for beginners (with curiosity to know spin physics) from any background to learn magnetic resonance.
  - A package for teaching magnetic resonance in a classroom.
  - User can easily understand the spin physics from the source code.
  - User can modify the source code, if needed.
- Not a quantitative approach, but a qualitative one

Features in brief

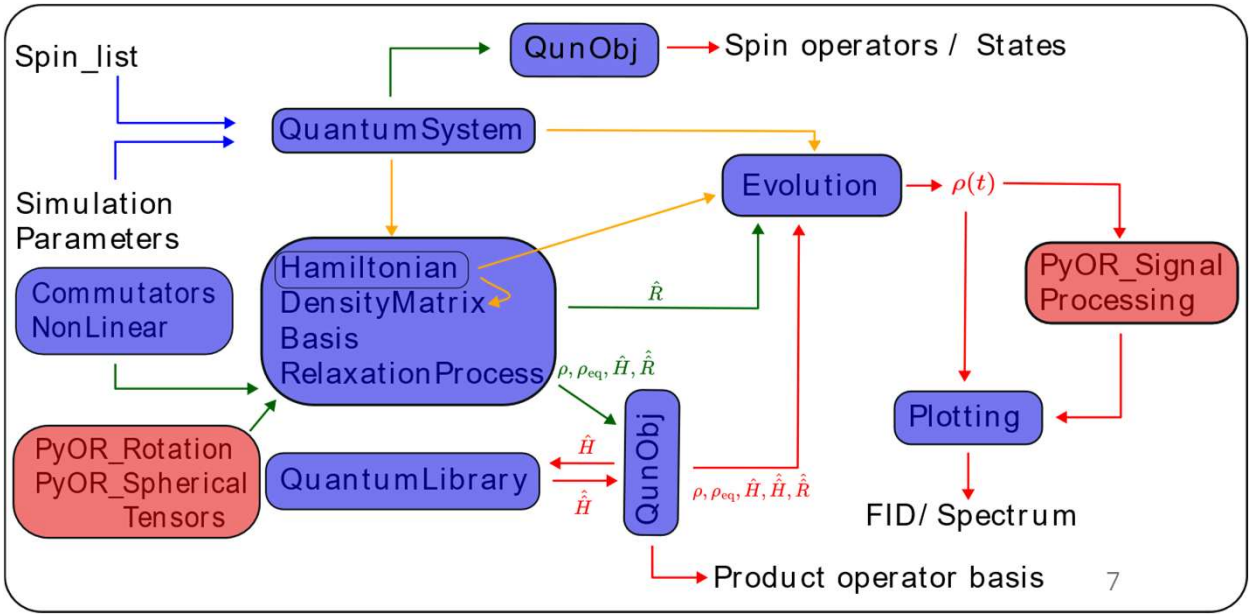


- Generate **spin operators** for a system with "any" number of spins (tested for up to 6) with any spin quantum number.
- Solve the **Liouville Equation** in **Hilbert Space or Liouville Space** with relaxation.
  - Unitary Propagation
  - Solve set of Ordinary Differential Equations (ODEs)
- Relaxation
  - **Redfield** Master Equation (Phenomenological, Dipolar relaxation, Random Field Fluctuation)
  - **Lindblad** Master Equation (Dipolar relaxation)
- Pre-written Hamiltonians or write your own.
- PyOR is developed mainly using **Numpy, Scipy** and **Matplotlib**.

Overview



- Define the spin system and its interactions.
- Choose propagation space and master equation.
- Define chemical shift and Coupling.
- Generate Hamiltonian.
- Initialize the density matrix.
- Evolve the density matrix.
- Compute expectation values.
- Plot FID and spectrum.

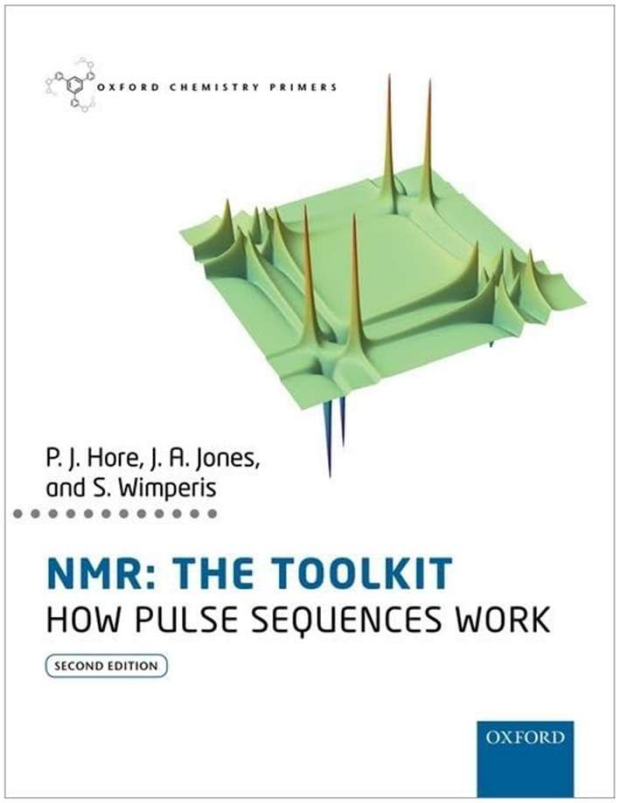


How do I get started with PyOR?

- From https://github.com/VThalakottoor/PyOR Download:
  - **Source_Doc: modules, Python files**
  - **Examples: Jupyter notebook files**
- Install Anaconda – Python Distribution
  - https://www.anaconda.com/download
- Install Visual Studio Code (VS Code)
  - https://code.visualstudio.com/
- Install Ipympl - Enables the use of the interactive features of matplotlib
  - pip install ipympl

Choose the Problem to Simulate

- Everybody has a problem to solve in NMR.
- For **beginners** looking for problems:
  - NMR: THE TOOLKIT: How Pulse Sequences Work (Oxford Chemistry Primers)



Example: Pulse - Acquisition

Define path to "Source_Doc"

There is no, pip install … (I expect user to modify the source code according to their problem. PyOR is not a black box.)

```
# Define the source path
SourcePath = 'path to/Source_Doc'

# Add source path
import sys
sys.path.append(SourcePath)
```

Import PyOR Modules

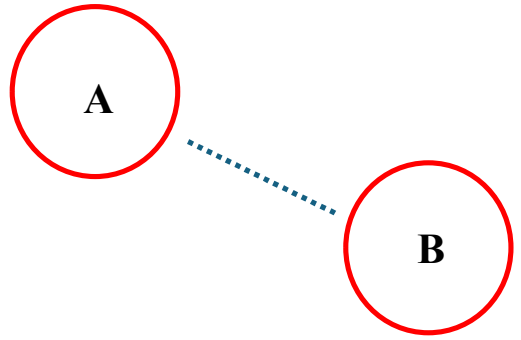
- **Module to generate spin operators**
  - `from PyOR_QuantumSystem (module .py) import QuantumSystem (class) as QunS`
- **Module to create quantum objects (states and operators)**
  - `from PyOR_QuantumObject import QunObj`
- **Module with quantum Library**
  - `from PyOR_QuantumLibrary import QuantumLibrary`
- **Module to generate Product Operator basis**
  - `from PyOR_Basis import Basis`
- **Module to generate Hamiltonians**
  - `from PyOR_Hamiltonian import Hamiltonian`

Import PyOR Modules

- **Module to generate Equilibrium Density Matrix**
  - `from PyOR_DensityMatrix import DensityMatrix`
- **Module to make Hard Pulse**
  - `from PyOR_HardPulse import HardPulse`
- **Module to make time Evolution**
  - `from PyOR_Evolution import Evolutions`
- **Module for plotting**
  - `from PyOR_Plotting import Plotting`
- **Module for Signal Processing**
  - `import PyOR_SignalProcessing as Spro`

Define your Spin System (Two spin system with J coupling and Phenomenological Relaxation)

```
# Define the spin system
Spin_list = {"A" : "H1", "B" : "H1"}
# Spin_list = {"A" : "H1"}
# Spin_list = {"A" : "H1", "B" : "H2"}

QS = QunS(Spin_list)

# Initialize the system
QS.Initialize()
```



Generate the Angular Momentum Spin Operators ("Quantum Objects")

X, Y and Z components of angular momentum operator

 # "X" spin operator, Particle A and B

Whole system: **QS.Ax** and **QS.Bx**

Sub system: **QS.Ax_sub** and **QS.Bx_sub**

 # "Y" spin operator, Particle A and B

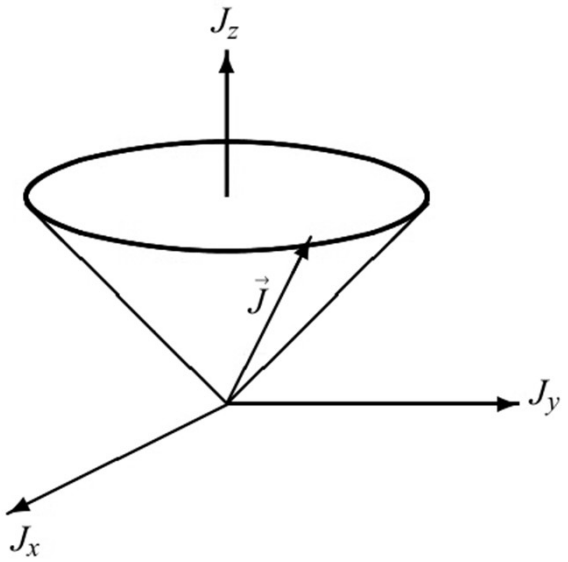
Whole system: **QS.Ay** and **QS.By**

Sub system: **QS.Ay_sub** and **QS.By_sub**

 # "Z" spin operator, Particle A and B

Whole system: **QS.Az** and **QS.Bz**

Sub system: **QS.Az_sub** and **QS.Bz_sub**



Generate the Angular Momentum Spin Operators ("Quantum" Objects)

Rasing and lowering angular momentum operators

 # "+" spin operator, Particle A and B

Whole system: **QS.Ap** and **QS.Bp**

Sub system: **QS.Ap_sub** and **QS.Bp_sub**

$$\hat{J}_{\pm} = \hat{J}_x \pm i\hat{J}_y$$

 # "-" spin operator, Particle A and B

Whole system: **QS.Am** and **QS.Bm**

Sub system: **QS.Am_sub** and **QS.Bm_sub**

Quantum Objects

- **Create your own states and operators.**
- **PyOR treats states and operators as an object with attributes and methods.**

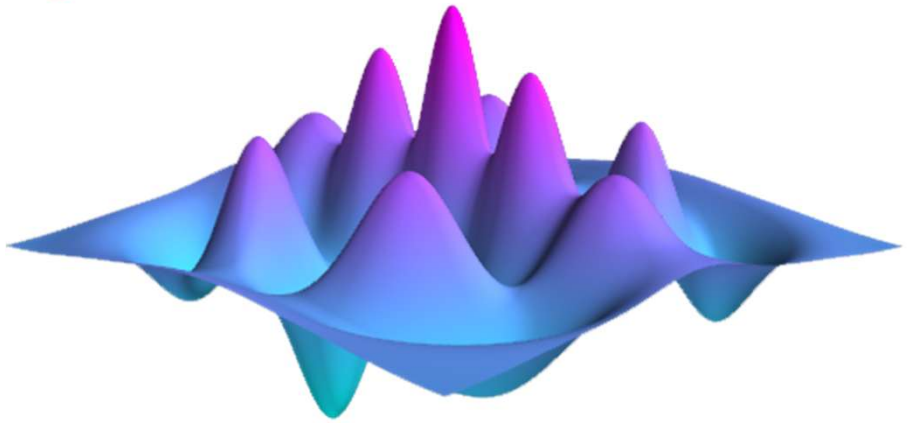
```
# Vector (ket) (from PyOR_QuantumObject import QunObj)
ket1 = QunObj([[1], [0]])
ket2 = QunObj([[0], [1]])
# Matrix (Spin operator)
Sx = QunObj([[0.0, 0.5],[0.5,0.0]])
Sy = QunObj([[0.0, -0.5j],[0.5j,  0.0]])
Sz = QunObj([[0.5, 0.0],[0.0, -0.5]])
# Identity Operator
Id = QunObj([[1., 0.0],[0.0, 1]])
```

Attributes of Quantum Objects

- `ket1.matrix` $\begin{bmatrix} 1.0 \\ 0 \end{bmatrix}$
- `Sx.matrix` $\begin{bmatrix} 0 & 0.5 \\ 0.5 & 0 \end{bmatrix}$
- `Ket1.type` `'ket'`
- `Sx.type` `'operator'`

- `Sx.datatype` `dtype('complex128')`
- `Sx.shape` `(2,2)`
- `Sx.data`

```
array([[0. +0.j, 0.5+0.j],
       [0.5+0.j, 0. +0.j]])
```



Inspiration from QuTiP
(Quantum Toolbox in Python)

Methods of Quantum Objects

- **Rotate a state:** `ket3 = ket1.Rotate(180,Sx)`
- **Multiply two objects:** `ket5 = Sx * ket1`
- **Add two objects:** `ket4 = 2 * ket1 + 5 *  ket2`
- **Create the Hamiltonian:**
  - `H =  10 * Sx * Sx + 10 * Sy * Sy + 10 * Sz * Sz`
- **Adjoint:** `bra1 = ket1.Adjoint()`
  - `Bra1.type    'bra'`
- **Conjugate:** `ket3.Conjugate()`
- **Transpose:** `ket3.Tranpose().matrix` $[0 \quad -1.0i]$

Methods of Quantum Objects

- **Trace:** `Sx.Trace()` Oj
- **Frobenius Norm:** `Sx.Norm()` 0.7071067
- **Exponential of an operator:** `Sx.Expm()`
- **Check Hermitian:** `Sx.Hermitian()` True
- **Check Commutation:** `Sx.Commute(Sy)` False
- **Tensor Product:** `Sx.TensorProduct(Id)`

$$\begin{bmatrix} 0 & 0 & 0.5 & 0 \\ 0 & 0 & 0 & 0.5 \\ 0.5 & 0 & 0 & 0 \\ 0 & 0.5 & 0 & 0 \end{bmatrix}$$

- **Outer Product:** `ket1.OuterProduct(ket1)`

$$\begin{bmatrix} 1.0 & 0 \\ 0 & 0 \end{bmatrix}$$

- **Inner Product:**
  - `Sx.InnerProduct(Sx)`
  - `ket1.InnerProduct(ket1)`
- **Normalize:** `Sx.Normalize()`

Back to the Two Spin System

 # "X" spin operator (whole system), Particle A

QS.Ax.matrix $\begin{bmatrix} 0 & 0 & 0.5 & 0 \\ 0 & 0 & 0 & 0.5 \\ 0.5 & 0 & 0 & 0 \\ 0 & 0.5 & 0 & 0 \end{bmatrix}$

 # "X" spin operator (whole system), Particle B

QS.Bx.matrix $\begin{bmatrix} 0 & 0.5 & 0 & 0 \\ 0.5 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.5 \\ 0 & 0 & 0.5 & 0 \end{bmatrix}$

 # "X" spin operator (sub system), Particle A

QS.Ax_sub.matrix $\begin{bmatrix} 0 & 0.5 \\ 0.5 & 0 \end{bmatrix}$

Set Parameters

$$\frac{d}{dt}\rho = \frac{-i}{\hbar}[H_0, \rho]$$

```
# Master Equation
QS.PropagationSpace = "Hilbert"
    "Hilbert" or "Liouville" or "Schrodinger"
QS.MasterEquation = "Redfield"
    "Redfield" or "Lindblad"

# Operator Basis
QS.Basis_SpinOperators_Hilbert = "Zeeman"
    "Zeeman" or "Singlet Triplet"
```

The user can change basis to the Hamiltonian eigenbasis using

```
Basis.BasisChange_HamiltonianEigenStates(H)
```

Set Parameters

```
# B0 Field in Tesla, Static Magnetic field (B0)
along Z
QS.B0 = 9.4

# Offset Frequency in rotating frame (Hz)
QS.OFFSET["A"] = 10.0
QS.OFFSET["B"] = 50.0

# Define J coupling between Spins
QS.JcoupleValue("A","B",5.0)
```

Set Parameters

```
# Define initial and final Spin Temperature
QS.I_spintemp["A"] = 300.0
QS.I_spintemp["B"] = 300.0
QS.F_spintemp["A"] = 300.0
QS.F_spintemp["B"] = 300.0

# Relaxation Process
QS.Rprocess = "Phenomenological" # "Auto-correlated Random Field Fluctuation", "Auto-correlated Dipolar Heteronuclear Ernst", "Auto-correlated Dipolar Homonuclear Ernst", … Look module: PyOR_Relaxation
QS.R1 = 1
QS.R2 = 2

QS.Update()
```

Generate Hamiltonian

```
# generate Larmor Frequencies
Ham = Hamiltonian(QS)
# Zeeman Hamiltonian (Rotating frame)
Hz = Ham.Zeeman_RotFrame()
Hz.Inverse2PI().Round(2).matrix
# J coupling Hamiltonian
Hj = Ham.Jcoupling()
Hj.Inverse2PI().Round(2).matrix
```

$$\begin{bmatrix} -30.0 & 0 & 0 & 0 \\ 0 & 20.0 & 0 & 0 \\ 0 & 0 & -20.0 & 0 \\ 0 & 0 & 0 & 30.0 \end{bmatrix}$$

$$\begin{bmatrix} 1.25 & 0 & 0 & 0 \\ 0 & -1.25 & 2.5 & 0 \\ 0 & 2.5 & -1.25 & 0 \\ 0 & 0 & 0 & 1.25 \end{bmatrix}$$

Equilibrium Density Matrix

```
DM = DensityMatrix(QS,Ham)

Thermal_DensMatrix = False

if Thermal_DensMatrix:
    # High Temperature
    HT_approx = False
    # Initial Density Matrix
    rho_in = DM.EqulibriumDensityMatrix(QS.Ispintemp,HT_approx)
    # Equlibrium Density Matrix
    rhoeq = DM.EqulibriumDensityMatrix(QS.Fspintemp,HT_approx)
else:
    rho_in = QS.Az + QS.Bz
    rhoeq = QS.Az + QS.Bz
```

rho_in.matrix

$$\begin{bmatrix} 1.0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1.0 \end{bmatrix}$$

Decomposition of the Density Matrix

```
# Product Operator Basis (PMZ / Shift Z basis)
  (Inspired from SpinDynamica)
BS = Basis(QS)

sort = 'negative to positive'
Index = False
Normal = True
Basis_PMZ, coh_PMZ, dic_PMZ =
BS.ProductOperators_SpinHalf_PMZ(sort,Index,Normal)
```

Decomposition of the Density Matrix

```
# Coherence order
coh_PMZ
[-2, -1, -1, -1, -1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 2]
# Dictionary
print(dic_PMZ)
['Im1 Im2 ', 'Im1 Iz2 ', 'Im1 Id2 ', 'Iz1 Im2 ', 'Id1 Im2 ', 'Im1
Ip2 ', 'Iz1 Iz2 ', 'Iz1 Id2 ', 'Id1 Iz2 ', 'Id1 Id2 ', 'Ip1 Im2 ',
'Iz1 Ip2 ', 'Id1 Ip2 ', 'Ip1 Iz2 ', 'Ip1 Id2 ', 'Ip1 Ip2 ']
# String index
Basis_PMZ_String = BS.String_to_Matrix(dic_PMZ, Basis_PMZ)
Basis_PMZ_String['Im1Im2'].matrix
```

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1.0 & 0 & 0 & 0 \end{bmatrix}$$

Decomposition of Density Matrix

```
# Decomposition of density matrix in PMZ basis
DM.DensityMatrix_Components(rho_in,Basis_PMZ,dic_PMZ)

Density Matrix = 1.0 Iz1 Id2  + 1.0 Id1 Iz2
```

Other basis:

1. **Spherical -** ProductOperators_SphericalTensor()
   or ProductOperators_SpinHalf_SphericalTensor
2. **Cartesian -** ProductOperators_SpinHalf_Cartesian()
3. **Zeeman -** ProductOperators_Zeeman()

Hard Pulse

```
HardP = HardPulse(QS)

flip_angle1 = 90.0    # Flip angle Spin 1
flip_angle2 = 90.0    # Flip angle Spin 2

rho = HardP.Rotate_Pulse(rho_in,flip_angle1,QS.Ay)
rho = HardP.Rotate_Pulse(rho,flip_angle2,QS.By)
rho.Tolarence(1.0e-5).matrix
```

$$\begin{bmatrix} 0 & 0.5 & 0.5 & 0 \\ 0.5 & 0 & 0 & 0.5 \\ 0.5 & 0 & 0 & 0.5 \\ 0 & 0.5 & 0.5 & 0 \end{bmatrix}$$

```
DM.DensityMatrix_Components(rho,Basis_PMZ,dic_PMZ)
Density Matrix = 0.70711 Im1 Id2  + 0.70711 Id1 Im2  + -0.70711 Id1
Ip2  + -0.70711 Ip1 Id2
```

Evolution

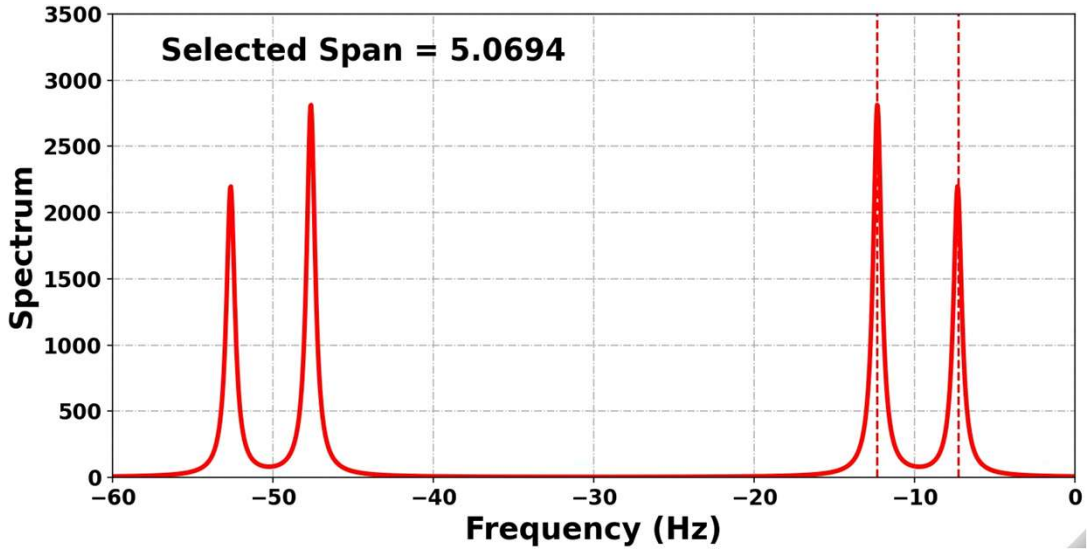
```
QS.AcqDT = 0.0001; QS.AcqAQ = 5.0; QS.OdeMethod = 'DOP853'
QS.PropagationMethod = "ODE Solver"
# "Unitary Propagator", "Unitary Propagator Time Dependent", "ODE Solver Lindblad", "ODE Solver ShapedPulse", "Relaxation", "Relaxation Lindblad", … Look module PyOR_Evolution
EVol = Evolutions(QS,Ham)

import time
start_time = time.time()
t, rho_t = EVol.Evolution(rho,rhoeq,Hz+Hj)
end_time = time.time()
timetaken = end_time - start_time
print("Total time = %s seconds " % (timetaken))
```

Signal and Spectrum

```
# Expectation Value
det_Mt = QS.Ap + QS.Bp
det_Z = QS.Az + QS.Bz

t, Mt = EVol.Expectation(rho_t,det_Mt)
t, Mz = EVol.Expectation(rho_t,det_Z)

# Fourier Transform
freq, spectrum =
Spro.FourierTransform(Mt,QS.AcqFS,5)
```

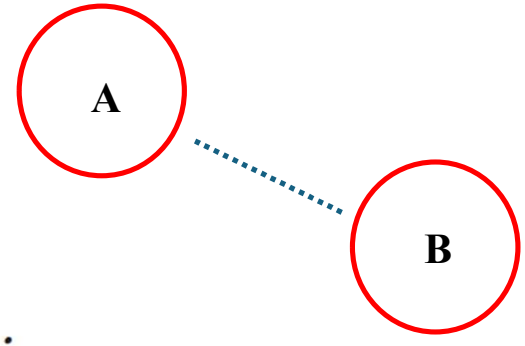
Plotting

```
%matplotlib ipympl
plot = Plotting(QS)

plot.PlotFigureSize = (10,5)
plot.PlotFontSize = 20
plot.PlotXlimt= (-60,0)
plot.PlotYlimt= (0,3500)

plot.Plotting_SpanSelector(freq,spectrum,"Frequency
(Hz)","Spectrum","red")
```



Example: NOE (Two spin system)

Set Parameters




```
# Master Equation
QS.PropagationSpace = "Liouville"
QS.MasterEquation = "Redfield"
```

$$\frac{d}{dt}\tilde{\rho} = \frac{-i}{\hbar}(\hat{H}_0 + i\hat{R})\tilde{\rho}$$

```
# Define paris of spins coupled by dipolar interaction
QS.Dipole_Pairs = [("A","B")]

# Relaxation Process
QS.Rprocess = "Auto-correlated Dipolar Homonuclear"
QS.RelaxParDipole_tau = 10.0e-12 # s
QS.RelaxParDipole_bIS = [30.0e3] # Hz
```

Commutation Superoperator

```
# generate Larmor Frequencies

Ham = Hamiltonian(QS)
Hz = Ham.Zeeman_RotFrame()

# J coupling Hamiltonian
Hj = Ham.Jcoupling()
```

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	−298.45	−15.71	0	0	0	0	0	0	0	0	0	0	0	0	0
0	−15.71	−47.12	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	−376.99	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	298.45	0	0	0	15.71	0	0	0	0	0	0	0
0	0	0	0	0	0	−15.71	0	0	15.71	0	0	0	0	0	0
0	0	0	0	0	−15.71	251.33	0	0	0	15.71	0	0	0	0	0
0	0	0	0	0	0	0	−78.54	0	0	0	15.71	0	0	0	0
0	0	0	0	15.71	0	0	0	47.12	0	0	0	0	0	0	0
0	0	0	0	0	15.71	0	0	0	−251.33	−15.71	0	0	0	0	0
0	0	0	0	0	0	15.71	0	0	−15.71	0	0	0	0	0	0
0	0	0	0	0	0	0	15.71	0	0	0	−329.87	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	376.99	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	78.54	−15.71	0
0	0	0	0	0	0	0	0	0	0	0	0	0	−15.71	329.87	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

```
# Generating the commutation superoperator
QS.RowColOrder = 'C'
QLib = QuantumLibrary(QS)

Hz_L = QLib.CommutationSuperoperator(Hz+Hj)
```

```
Hz_L.Round(2).matrix
```

$$\hat{H}_0 = H_0 \otimes \mathbb{1} - \mathbb{1} \otimes H_0^T$$

Quantum Library

```
# Create states
ket1 = QLib.Basis_Ket(2,0, PrintDefault=True)
ket2 = QLib.Basis_Ket(2,1, PrintDefault=True)
psi1 = 1 * ket1 + 2 * ket2
# Outer Product
rho1 = QLib.OuterProduct(psi1,psi1)

# Vectorization
vec1 = QLib.DMToVec(rho1)

# Back to vector
DM1 = QLib.VecToDM(vec1,shape=(2,2))
```

Ket1.matrix

$$\begin{bmatrix} 1.0 \\ 0 \end{bmatrix}$$

ket1.matrix

$$\begin{bmatrix} 0 \\ 1.0 \end{bmatrix}$$

$$\begin{bmatrix} 1.0 & 2.0 \\ 2.0 & 4.0 \end{bmatrix}$$ rho1.matrix

$$\begin{bmatrix} 1.0 \\ 2.0 \\ 2.0 \\ 4.0 \end{bmatrix}$$ vec1.matrix

$$\begin{bmatrix} 1.0 & 2.0 \\ 2.0 & 4.0 \end{bmatrix}$$ DM1.matrix

And many other interesting methods available.

Back to NOE

```
# Density Matrix
rho_in_L = DM.EquilibriumDensityMatrix(QS.Ispintemp,HT_approx)
rhoeq = DM.EquilibriumDensityMatrix(QS.Fspintemp,HT_approx)
# Pulse
flip_angle1 = 0.0   # Flip angle Spin 1
flip_angle2 = 180.0 # Flip angle Spin 2
rho = HardP.Rotate_Pulse(rho_in_L,flip_angle1,QS.Ay)
rho = HardP.Rotate_Pulse(rho,flip_angle2,QS.By)
# Relaxation Superoperator  (from PyOR_Relaxation import RelaxationProcess)
RPro = RelaxationProcess(QS)
R_L = RPro.Relaxation()
```

rho_in_L.matrix

$$\begin{bmatrix} 0.250016003847122 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0.25 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0.25 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0.249983996152878 \end{bmatrix}$$

Back to NOE

```
# Density Matrix
rho_in_L = DM.EquilibriumDensityMatrix(QS.Ispintemp,HT_approx
rhoeq = DM.EquilibriumDensityMatrix(QS.Fspintemp,HT_approx)
# Pulse
flip_angle1 = 0.0
flip_angle2 = 180.
rho = HardP.Rotate
rho = HardP.Rotate
# Relaxation Super
RPro = RelaxationF
R_L = RPro.Relaxat
```

rho_in_L.matrix

$$\begin{bmatrix} 0.250016003847122 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0.25 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0.25 \\ 0 \end{bmatrix}$$

R_L.Round(2).matrix

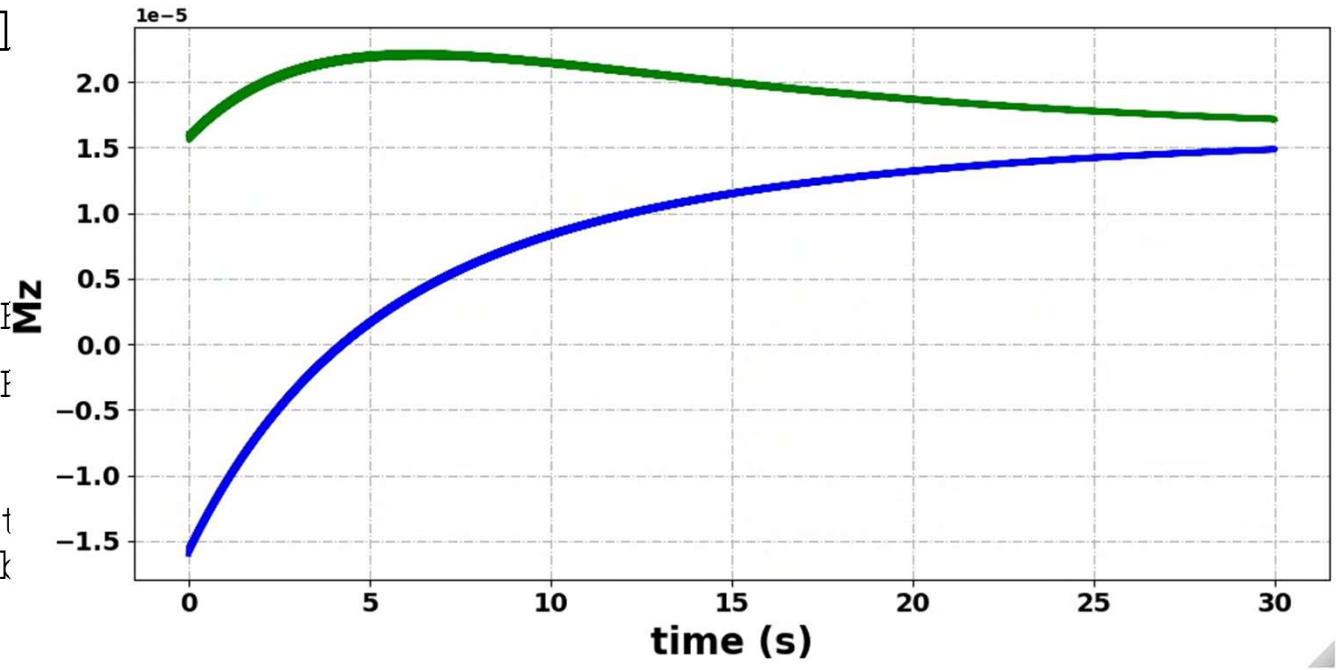
$$\begin{bmatrix}
0.16 & 0 & 0 & 0 & 0 & -0.03 & -0.03 & 0 & 0 & -0.03 & -0.03 & 0 & 0 & 0 & 0 & -0.11 \\
0 & 0.15 & 0.06 & 0 & 0 & 0 & 0 & 0.03 & 0 & 0 & 0 & 0.03 & 0 & 0 & 0 & 0 \\
0 & 0.06 & 0.15 & 0 & 0 & 0 & 0 & 0.03 & 0 & 0 & 0 & 0.03 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0.16 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0.15 & 0 & 0 & 0 & 0.06 & 0 & 0 & 0 & 0 & 0.03 & 0.03 & 0 \\
-0.03 & 0 & 0 & 0 & 0 & 0.07 & 0.03 & 0 & 0 & 0.03 & -0.02 & 0 & 0 & 0 & 0 & -0.03 \\
-0.03 & 0 & 0 & 0 & 0 & 0.03 & 0.07 & 0 & 0 & -0.02 & 0.03 & 0 & 0 & 0 & 0 & -0.03 \\
0 & 0.03 & 0.03 & 0 & 0 & 0 & 0 & 0.15 & 0 & 0 & 0 & 0.06 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0.06 & 0 & 0 & 0 & 0.15 & 0 & 0 & 0 & 0 & 0.03 & 0.03 & 0 \\
-0.03 & 0 & 0 & 0 & 0 & 0.03 & -0.02 & 0 & 0 & 0.07 & 0.03 & 0 & 0 & 0 & 0 & -0.03 \\
-0.03 & 0 & 0 & 0 & 0 & -0.02 & 0.03 & 0 & 0 & 0.03 & 0.07 & 0 & 0 & 0 & 0 & -0.03 \\
0 & 0.03 & 0.03 & 0 & 0 & 0 & 0 & 0.06 & 0 & 0 & 0 & 0.15 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.16 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0.03 & 0 & 0 & 0 & 0.03 & 0 & 0 & 0 & 0 & 0.15 & 0.06 & 0 \\
0 & 0 & 0 & 0 & 0.03 & 0 & 0 & 0 & 0.03 & 0 & 0 & 0 & 0 & 0.06 & 0.15 & 0 \\
-0.11 & 0 & 0 & 0 & 0 & -0.03 & -0.03 & 0 & 0 & -0.03 & -0.03 & 0 & 0 & 0 & 0 & 0.16
\end{bmatrix}$$

Nuclear Overhauser Effect

```
# Evolution
QS.PropagationMethod = "Relaxation"
t, rho_t = EVol.Evolution(rho,rhoeq,Hz_L,R_L)
# Expextation
det_Z1 = QS.Az
det_Z2 = QS.Bz
t, signal_Z1 = EVol.Expectation(rho_t,det_Z1)
t, signal_Z2 = EVol.Expectation(rho_t,det_Z2)
# Plotting
plot.PlottingMulti([t,t],[signal_Z1,signal_Z2],"time
(s)","Mz",["green","blue"])
```

Nuclear Overhauser Effect

```
# Evolution
QS.PropagationMethod = "Relaxation"
t, rho_t = EVol
# Expextation
det_Z1 = QS.Az
det_Z2 = QS.Bz
t, signal_Z1 = EVol.
t, signal_Z2 = EVol.
# Plotting
plot.PlottingMulti([t
(s)","Mz",["green","
```

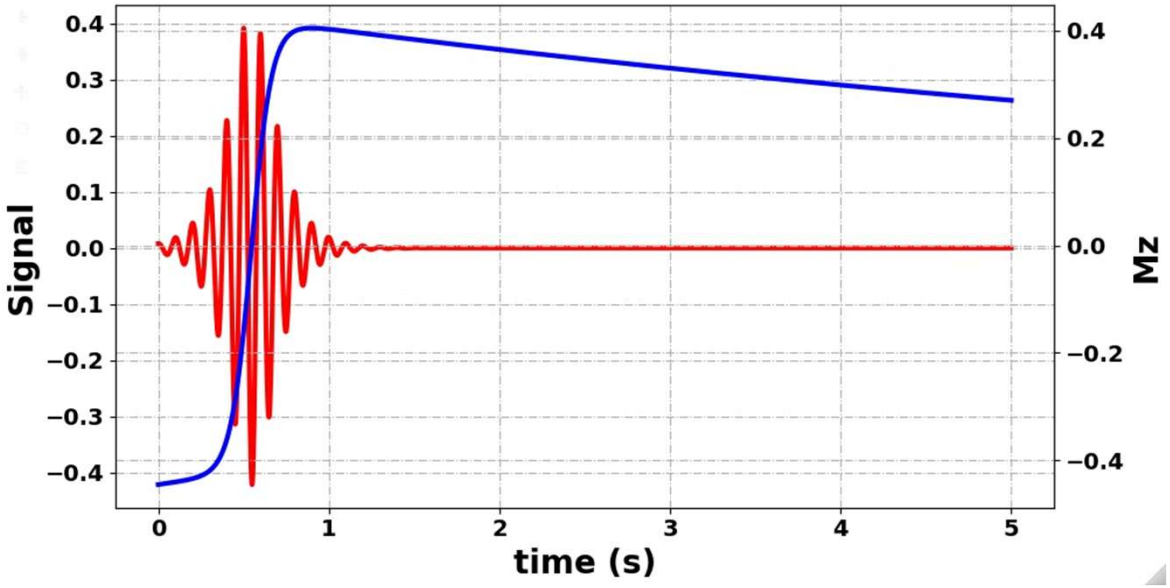


Features: Radiation Damping (Semi-Quantum)

```
# Spin list
Spin_list = {"A" : "H1"}

# Parameter
## Radaition Damping
QS.Rdamping = True
QS.RD_xi["A"] = 20
QS.RD_phase["A"] = 0

# Flip angle
flip_angle1 = 179.0

# Plotting
Plot.PlottingTwin_SpanSelector(t,signal,Mz,"time (s)", "Signal", "Mz",
"red", "blue", saveplt=True)
```

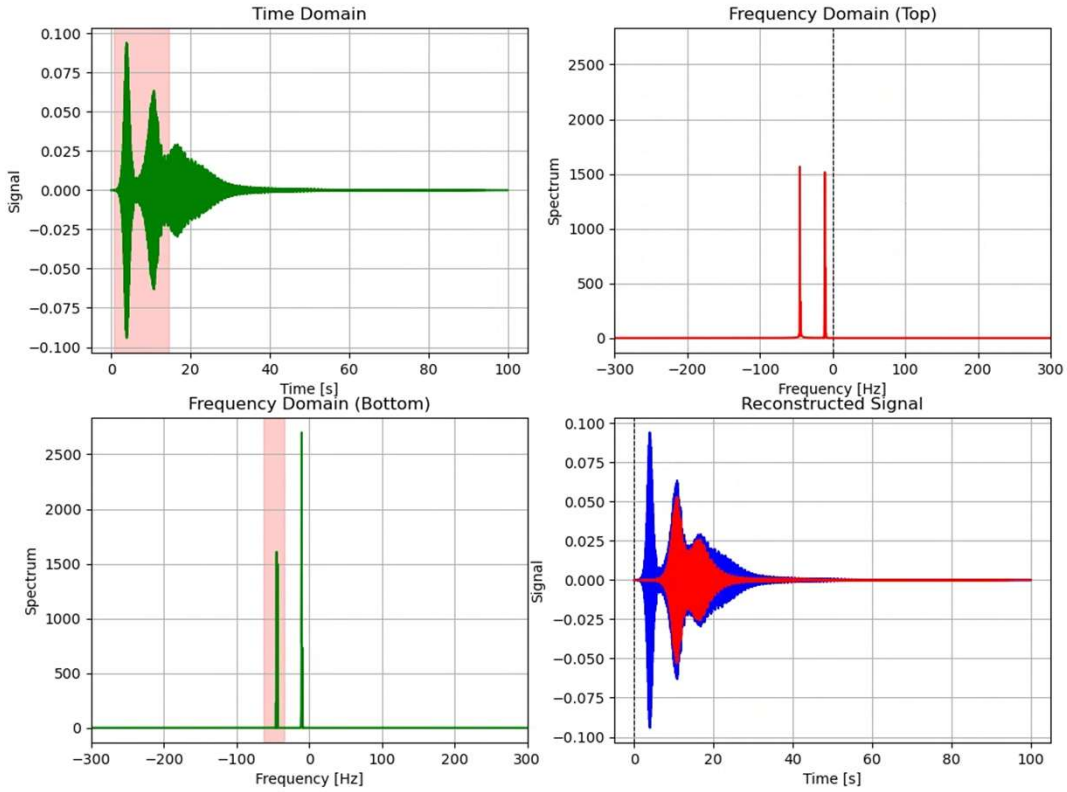
$$\mathbf{B}_{\mathrm{FB}} = Ge^{-i\psi}\sum_k m_k(t)$$

Thalakottoor, et.al., Phys. Rev. Lett. **133**, 158001



Features: Maser Data Analyzer

```
from PyOR_MaserDataAnalyzer import
MaserDataAnalyzer

# Simulation
Data1 = MaserDataAnalyzer("path to/
signal.npy", 0.0001,
simulation = True)

Data1.Plot()

# Bruker Data
Data1 = MaserDataAnalyzer("path to/
fid1.csv", 20.e-6)
```



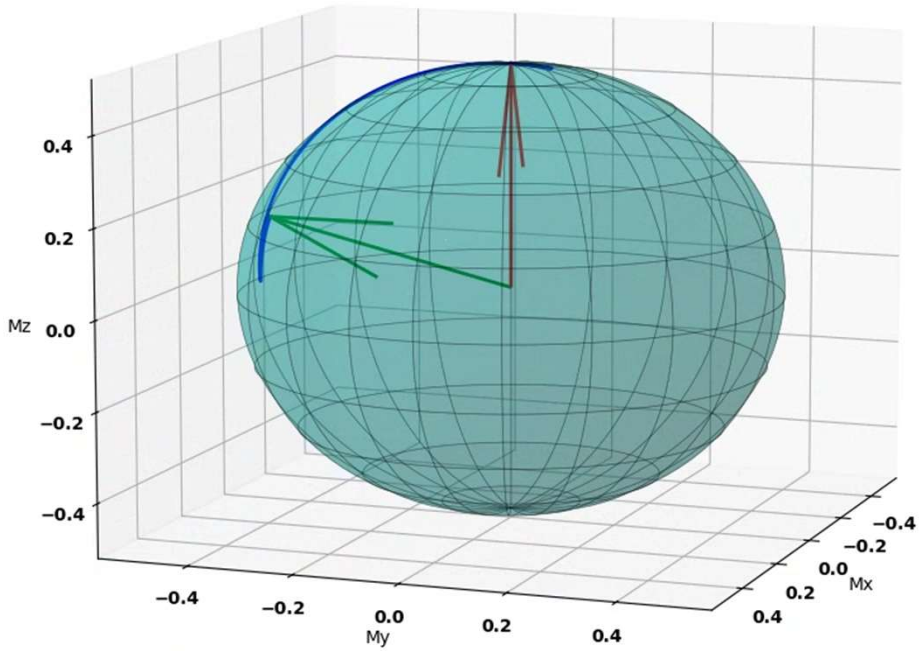
Features: Shaped Pulse (With Radiation Damping)

```
# Shape file

pulseFile = '/opt/topspin4.1.4/exp/stan/nmr/lists/wave/Rsnob.1000'
# Rsnob.1000 or square.1000 or Gaus1.1000

pulseLength = 1000.0e-6

RotatioAngle = 90.0

t, amp, phase =
Ham.ShapedPulse_Bruker(pulseFile,pulseLength,RotatioAngle)


# Interpolartion

Kind = "previous"

Iamp, Iphase = Ham.ShapedPulse_Interpolate(t,amp,phase,Kind)
```

Features: Shaped Pulse

```
# Shape Pulse Parameters

EVol = Evolutions(QS,Ham)

EVol.ShapeFunc = "Bruker"
EVol.ShapeParOmega = Iamp
EVol.ShapeParPhase = Iphase
EVol.ShapeParFreq = 0.0

# Acquisition parameters
EVol.AcqAQ = pulseLength
Npoints = 1000
EVol.AcqDT = EVol.AcqAQ/Npoints

EVol.PropagationMethod = "ODE Solver ShapedPulse"
```

```
# Plotting
plot.PlottingSphere(Mx1,My1,Mz1,rhoeq,plot
_vector,scale_datapoints)
```



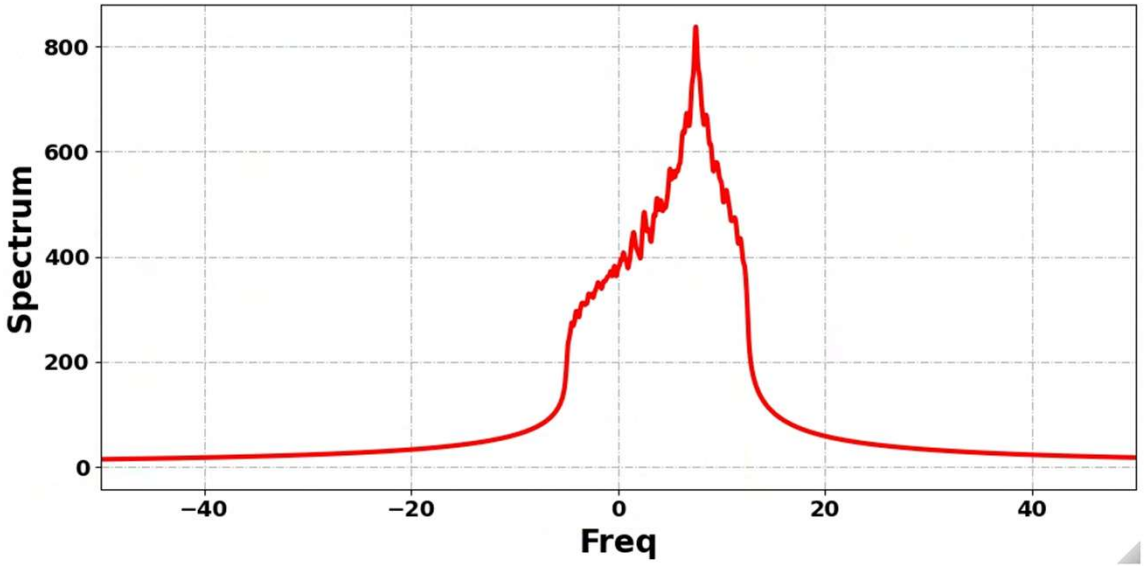
Features: Powder Average (CSA)

```
# Principle axis frame Tensor
delta_iso = 5.0 # Hz
delta_aniso = -10.0 # Hz

IT_PAF =
Ham.InteractionTensor_PAF_CSA(Iso=delta_iso,Aniso=delta_
aniso,Asymmetry=0.5)

# Crystal Orientation
import PyOR_CrystalOrientation as CO
alpha, beta, gamma, weight =
CO.Load_Crystallite_CSV("rep2000_cryst.csv")
```

$$\begin{bmatrix} 12.5 & 0 & 0 \\ 0 & 7.5 & 0 \\ 0 & 0 & -5.0 \end{bmatrix}$$

Features: Powder Average (CSA)

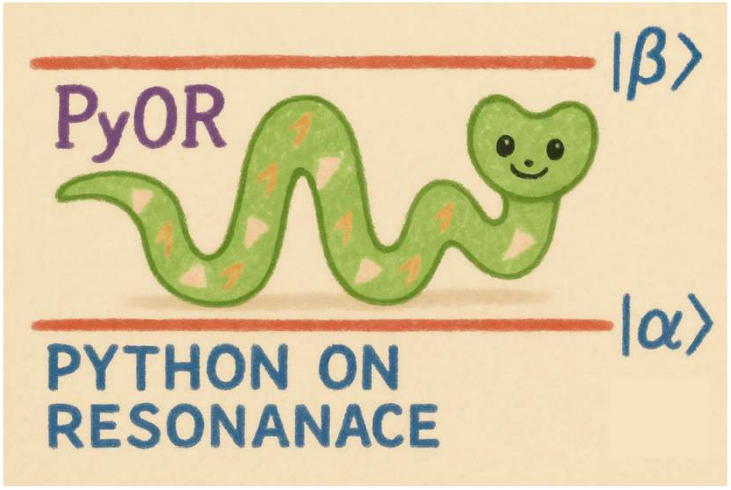
```
# Evolution
QS.AcqDT = 0.0001
QS.AcqAQ = 5.0
QS.Update()

QS.PropagationMethod =
 "Unitary Propagator"

EVol = Evolutions(QS,Ham)
EVol.Update()

A = "A" # Spin
B = ""  # Field
freq, spectrum = Ham.PowderSpectrum(EVol,rhoI, rhoeq, A, IT_PAF, B, "spin-field",
"secular", gamma, beta, alpha, weighted=True, weight = weight)
```



Conclusion

- PyOR is still a developing package:
  - There will be bugs.
  - Maybe I overlooked somethings.
  - If you see any mistakes and have any suggestions, write to me.
- Future implementations:
  - Various Hyperpolarization techniques.
  - Atomic magnetometry, NV centers.
  - Optimization.
- This project welcomes contributors

Download, Cite and Tutorials



- Documentation:
  - https://vthalakottoor.github.io/PyOR/
- Download and Examples:
  - https://github.com/VThalakottoor/PyOR
- Cite:
  - https://doi.org/10.5281/zenodo.15241169
- Tutorials:
  - MARQUISE (**Ma**gnetic **R**esonance **Qu**antum **I**nformation **S**cience and **E**ducation)
  - https://www.linkedin.com/company/marquise-education/
  - https://quantum-resonance.org/category/simulation/
- Contact: vineethfrancis.physics@gmail.com

Acknowledgements:

- Daniel Abergel
- ANR for funding (project ANR-22-CE29-0006 DynNonlinPol)

Dedication:

- Jean Jeener – My hero in NMR.
- *Gauri – A whisper of a memory*. And the soul and catalyst of PyOR …

Thank You

"Lost in the solitude of his immense power, he began to lose direction. He felt scattered about, multiplied, more solitary than ever."
Gabriel Garcia Marquez,
One Hundred Years of Solitude