File - Constants.py

```python
import numpy
from scipy import constants

mu0: float = 1.2566e-6  # [kg m A^-2 s^-2]
c: float = 2.998e8  # [m/s]
alpha: float = 1/137.035999
hbar: float = 1.054571817e-34  # [j*s]
h: float = 6.602607015e-34  # [j*s]
Z_H: int = 1
q_e: float = 1.6021766208e19  # [As]
m_e: float = 9.109e-31  # [kg]
m_n: float = 1.6889e-27  # [kg]
m_p: float = 1.6725e-27  # [kg]
mu_B: float = (q_e * hbar)/(2 * m_e)
g_ell: float = 1.0
g_s: float = 2.0
s_H: numpy.ndarray = numpy.array([1/2])  # possible s for Hydrogen
k_B: float = constants.value("Boltzmann constant")  # Boltzmann constant [J K^-1]
a0: float = constants.value("Bohr radius")  # 0.529 177 210 67 e-10 [m]


def mu_mag(L, S) -> float:
    """
    return the magnetic field with quantum number L and S
    :param L: quantume number L (float)
    :param S: quantum number S (S)
    :return: (float)
    """
    return -(mu_B/hbar)*(g_ell*L + g_s*S)


def mu_mass(m_1, m_2) -> float:
    """
    return the reduced mass
    :param m_1: mass 1 (float)
    :param m_2: mass 2 (float)
    :return: reduced mass (float)
    """
    return (m_1*m_2)/(m_1+m_2)


mu_H = mu_mass(m_p, m_e)  # reduced mass of hydrogen


if __name__ == '__main__':
    print(f"alpha = {alpha}, \n"
          f"hbar = {hbar}, \n"
          f"c = {c}, \n"
          f"a0 = {a0}, \n"
          f"mu_H = {mu_H}")

```

File - geometric_calculus.py

```python
import numpy as np
import Constants as const


class Geometric_calculus:
    """
    This class wraps up methods that calculate multiple geometric
     information such as angles and distances
    """

    def __init__(self, w_c, angle, a, L, B, C) -> None:
        """Constructor

        Parameters
        ----------
        w_c : float
            Calibrating frequency in Hz
        angle : float
            Angle of the beam producer in rad
        a : float
            Lenght of a side of the prism in meters
        L : float
            Distance between the center of the prism and the screen in meter
        B : float
            First Cauchy law's constant
        C : float
            Second Cauchy law's constant in meters^2

        Attributes
        ----------
        self.w_c: float
            Calibrating frequency in Hz of the system
        self.angle: float
            Angle of the beam producer in rad
        self.phi : float
            Angle of the prism
        self.a: float
            Lenght of a side of the prism
        self.L: float
            Distance between the center of the prism and the screen in meter
        self.B : float
            First Cauchy law's constant
        self.C : float
            Second Cauchy law's constant in meters^2
        self._x_ref : float
            Position of the beam on the screen for the frequency w_c in meters
        """
        self._w_c = w_c
        self._angle = angle
        self._a = a
        self._L = L
        self._B = B
        self._C = C
        self._phi = self._get_phi(self._w_c)
        self._x_ref = self.get_x_for_frequency(self._w_c)

    def _get_phi(self, w_c) -> float:
        """This methods calculates the value of
        the angle of the rotation of the prism which
        ensures that the beam with a frequency of w_c
        is perpendicular to the screan

        Parameters
        ----------
        w_c : float
            Calibrating frequency in Hz
        angle : float
            Angle of the beam producer in rad
        B : float
            First Cauchy law's constant
        C : float
            Second Cauchy law's constant in  meters^2
        Returns
        ---------
        phi: float
            value of the angle of rotation of the prism in rad
        """
        n = self._get_refraction_value(w_c)
        sin_argument = np.pi / 3 - np.arcsin((np.sin(self._angle + np.pi / 6) / n))
        arcsin_argument = n * np.sin(sin_argument)
        phi = np.arcsin(arcsin_argument) - np.pi / 6
        return phi

    def _get_refraction_value(self, w) -> float:
        """This methods calculates the value of
           the angle of the rotation of the prism which
            ensures that the beam with a frequency of w_c
            is perpendicular to the screan

            Parameters
            ----------
            w : float
                 Calibrating frequency in frequency of the beam
            Returns
            ----------
            value: float
                The value of the refraction index for the given frequency
        """
```

File - geometric_calculus.py

```
 99         c = const.c
100         value = self._B + self._C / (((2 * np.pi * c) / w) ** 2)
101         return value
102 
103     def get_x_for_frequency(self, w) -> float:
104         """This methods calculates the value of
105             the position of the beam on the screen
106 
107             Parameters
108             ----------
109             w : float
110                 frequency of the beam in Hz
111 
112             Returns
113             ----------
114             value: float
115                 The value of the position of the beam on the screen in meters
116         """
117         x = self.get_h_of_first_refraction_of_beam(w) - self.get_differences_of_h(w)
118         return x
119 
120     def get_h_of_first_refraction_of_beam(self, w) -> float:
121         """This methods calculates the value of
122             the position of the beam on the screen if
123             it was refracted only once
124 
125             Parameters
126             ----------
127             w : float
128                 frequency of the beam in Hz
129 
130             Returns
131             ----------
132             value: float
133                 The value of the position of the beam on the screen in meters if the beam was refracted only once
134         """
135         n = self._get_refraction_value(w)
136         arcsin_argument = np.sin(np.pi / 6 + self._angle) / n
137         h_top = (self._L + (self._a / 3) * np.cos(self._phi)) * (
138             np.sin(np.arcsin(arcsin_argument) - np.pi / 6 + self._phi))
139         return h_top
140 
141     def get_differences_of_h(self, w) -> float:
142         """This methods calculates the value of
143             the difference of height between one and two diffraction
144 
145             Parameters
146             ----------
147             w : float
148                 frequency of the beam in Hz
149 
150             Returns
151             ----------
152             value: float
153                 the height differences in m
154         """
155         n = self._get_refraction_value(w)
156         H = (self._L + (self._a / 3) * np.cos(self._phi)) / np.cos(
157             np.arcsin((np.sin(np.pi / 6 + self._angle)) / n) - np.pi / 6 + self._phi)
158         d = (np.sqrt(3) * (self._a / 3)) / np.sin(np.arcsin((np.sin(np.pi / 6 + self._angle)) / n) + np.pi / 6)
159         den = np.sin(
160             np.pi / 2 + np.abs(np.arcsin(np.sin(np.pi / 6 + self._angle) / n) - np.pi / 6 + self._phi) - np.abs(
161                 np.arcsin(np.sin(np.pi / 6 + self._angle) / n) - np.arcsin(
162                     n * np.sin(np.pi / 3 - np.arcsin(np.sin(np.pi / 6 + self._angle) / n)))))
163         nim = np.sin(np.arcsin((np.sin(np.pi / 6 + self._angle)) / n) - np.arcsin(
164             n * np.sin(np.pi / 3 - np.arcsin((np.sin(np.pi / 6 + self._angle)) / n))))
165         diff_h = ((H - d) * nim) / den
166         return diff_h
167 
168     def get_delta_x(self, w) -> float:
169         """This methods calculates the value of
170             the distance between the beam and the beam of frequency of w_c
171             on the screen
172 
173             Parameters
174             ----------
175             w : float
176                 frequency of the beam in Hz
177 
178             Returns
179             ----------
180             value: float
181                 The value of the position of the beam on the screen in meters
182         """
183         delta_x = self.get_x_for_frequency(w) - self._x_ref
184         return delta_x
185 
186 
187 if __name__ == "__main__":
188     G = Geometric_calculus(3.5e15, np.pi / 4, 2e-2, 50e-2, 1.4580, 0.00354e-12)
189     print(G.get_delta_x(2.0e15))
190     print(G.get_delta_x(2.5e15))
191     print(G.get_delta_x(3e15))
192     print(G.get_delta_x(3.5e15))
193     print(G.get_delta_x(4e15))
194     print(G.get_delta_x(4.51e15))
195     print(G.get_delta_x(5e15))
196 
```

File - QuantumFactory.py

```python
import itertools

import mpmath
import numpy as np
import scipy as sc
import sympy as sp
from numba import prange
from scipy import integrate

import Constants as const


class QuantumFactory:
    """
    QuantumFactory is a module to combine a bunch of static method util in quantum mechanics
    """

    @staticmethod
    def zeta_n(n=sp.Symbol("n", real=True), r=sp.Symbol("r", real=True, positive=True),
               z=sp.Symbol("Z", real=True), mu=sp.Symbol('mu', real=True)):
        """
        return the zeta_n function
        :param n: orbital number n (int)
        :param r: rayon variable (sympy object)
        :param z: atomic number (int)
        :param mu: reduced mass
        :return: zeta_n function (sympy object)
        """
        return (2 * z * const.alpha * mu * const.c * r) / (n * const.hbar)

    @staticmethod
    def u_n_ell(n=sp.Symbol("n", real=True), ell=sp.Symbol("ell", real=True),
                z=sp.Symbol("Z", real=True), mu=sp.Symbol('mu', real=True)):
        """
        return the u_{n \ell} function
        :param n: orbital number n (int)
        :param ell: kinetic momentum  (int)
        :param z: atomic number (int)
        :param mu: reduced mass (float)
        :return: (sympy object)
        """
        u_coeff_norm = sp.sqrt(((2 * z * const.alpha * mu * const.c) / (n * const.hbar)) ** 3)
        u_coeff_fact = sp.sqrt(np.math.factorial(n - ell - 1) / (2 * n * np.math.factorial(n + ell)))
        u_coeff = u_coeff_norm * u_coeff_fact

        exp_term = sp.exp(-QuantumFactory.zeta_n(n=n, z=z, mu=mu) / 2)
        laguerre_term = sp.assoc_laguerre(n - ell - 1, 2 * ell + 1, QuantumFactory.zeta_n(n=n, z=z, mu=mu))

        return u_coeff * exp_term * (QuantumFactory.zeta_n(n=n, z=z, mu=mu) ** ell) * laguerre_term

    @staticmethod
    def Y_ell_m_ell(ell: int, m_ell: int):
        """
        Return the spherical harmonic function
        :param ell: kinetic momentum (int)
        :param m_ell: quantum number m_ell (int)
        :return: (sympy object)
        """
        from sympy.functions.special.spherical_harmonics import Ynm
        theta, phi = sp.Symbol("theta", real=True), sp.Symbol("phi", real=True)
        # return sp.FU['TR8'](Ynm(ell, m_ell, theta, phi).expand(func=True))
        # return Ynm(ell, m_ell, theta, phi)
        return Ynm(ell, m_ell, theta, phi).expand(func=True)

    @staticmethod
    def get_hydrogen_wave_function(n: int, ell: int, m_ell: int):
        """
        Give the hydrogen wave function of sympy
        :param n: quantum number n (int)
        :param ell: quantum number ell (int)
        :param m_ell: quantum number m_ell (int)
        :return: hydrogen wave function (sympy object)
        """
        from sympy.physics import hydrogen
        r, theta, phi = sp.Symbol("r", real=True, positive=True), \
                        sp.Symbol("theta", real=True), sp.Symbol("phi", real=True)
        return hydrogen.Psi_nlm(n, ell, m_ell, r, phi, theta, Z=1 / const.a0)

    @staticmethod
    def get_valid_ell_with_n(n: int) -> np.ndarray:
        """
        Gives the possible ell with a given n
        :param n: quantum number n (int)
        :return: array of possible quantum number ell (np.ndarray)
        """
        return np.array([i for i in np.arange(start=0, stop=n, step=1)])

    @staticmethod
    def get_valid_m_ell_with_ell(ell: int) -> np.ndarray:
        """
        Gives the possible m_ell with a given ell
        :param ell: quantum number ell (int)
        :return: array of possible m_ell (np.ndarray)
        """
        return np.array([i for i in np.arange(start=-ell, stop=ell + 1, step=1)])

    @staticmethod
    def get_valid_m_s_with_s(s: float) -> np.ndarray:
```

```
 99          """
100          Gives the possible m_s with a given s
101          :param s: quantum number s or spin (float)
102          :return: array of the possible m_s (numpy.ndarray)
103          """
104          return np.array([i for i in np.arange(start=-s, stop=s + 1, step=1)])
105
106      @staticmethod
107      def get_valid_transitions_n_to_n(n, n_prime) -> list:
108          """
109          Get a Transitions(list) container of all of the valid transition of n to n_prime
110          :param n: initial quantum number n (int)
111          :param n_prime: final quantum number n (int)
112          :return: Transitions(list) of Transition object of the initial state and final state (Transitions)
113          """
114          from Transition import Transition
115          from Transitions import Transitions
116          import warnings
117          warnings.warn("Warning! This method seems to be not valid", DeprecationWarning)
118          valid_transitions = Transitions()
119          for init_quantum_state in QuantumFactory.get_valid_quantum_state_for_n(n):
120              for end_quantum_state in init_quantum_state.get_valid_transitions_state_to_n(n_prime):
121                  valid_transitions.append(Transition(init_quantum_state, end_quantum_state))
122          return valid_transitions
123
124      @staticmethod
125      def get_valid_transitions_n_to_n_prime(n: int, n_prime: int, hydrogen: bool = False) -> list:
126          """
127          Get a Transitions(list) container of all of the valid transition of n to n_prime
128          :param n: initial quantum number n (int)
129          :param n_prime: final quantum number n (int)
130          :param hydrogen: if we want to cast quantum state in quantum hydrogen state (bool)
131          :return: Transitions(list) of Transition object of the initial state and final state (Transitions)
132          """
133          from Transition import Transition
134          from Transitions import Transitions
135          valid_transitions: Transitions = Transitions()
136          for init_quantum_state in QuantumFactory.get_valid_quantum_state_for_n(n, hydrogen=hydrogen):
137              for end_quantum_state in QuantumFactory.get_valid_quantum_state_for_n(n_prime, hydrogen=hydrogen):
138                  if Transition.possible(init_quantum_state, end_quantum_state):
139                      valid_transitions.append(Transition(init_quantum_state, end_quantum_state))
140          return valid_transitions
141
142      @staticmethod
143      def get_valid_quantum_state_for_n(n, s_array: np.ndarray = const.s_H, hydrogen: bool = False) -> np.ndarray:
144          """
145          Get all the valid quantum state for the orbital number n
146          :param n: orbital number n (int)
147          :param s_array: array of possible spin number s (numpy.ndarray)
148          :param hydrogen: if we want to cast quantum state in quantum hydrogen state (bool)
149          :return: array of valid quantum state (numpy.ndarray[QuantumState])
150          """
151          from QuantumState import QuantumState
152          valid_states: list = list()
153          for ell in QuantumFactory.get_valid_ell_with_n(n):
154              for m_ell in QuantumFactory.get_valid_m_ell_with_ell(ell):
155                  for s in s_array:
156                      for m_s in QuantumFactory.get_valid_m_s_with_s(s):
157                          valid_states.append(QuantumState(n, ell, m_ell, s, m_s, hydrogen))
158          return np.array(valid_states)
159
160      @staticmethod
161      def get_g_n(n: int) -> int:
162          """
163          Get the degeneration number of the level n
164          :param n: orbital level (int)
165          :return: degeneration number (int)
166          """
167          deg_dico = dict()
168          for quantum_state in QuantumFactory.get_valid_quantum_state_for_n(n):
169              energy = quantum_state.get_state_energy()
170              if energy in deg_dico.keys():
171                  deg_dico[energy] += 1
172              else:
173                  deg_dico[energy] = 1
174          return int(np.sum(list(deg_dico.values())))
175
176      @staticmethod
177      def get_g_unperturbed(n: int) -> int:
178          """
179          Get the degeneration number of the level n with unperturbed energy
180          :param n: quantum number n (int)
181          :return: degeneration number (int)
182          """
183          return int(2*(n**2))
184
185      @staticmethod
186      def get_state_energy_unperturbed(n: int, z=sp.Symbol("Z", real=True), mu=sp.Symbol('mu', real=True)) -> float:
187          """
188          Get the energy of the current quantum state
189          :param n: quantum number n (int)
190          :param z: atomic number (float)
191          :param mu: reduced mass (float)
192          :return: the energy of the current quantum state (float if z and mu are float else sympy object)
193          """
194          numerator = - (z ** 2) * (const.alpha ** 2) * mu * (const.c ** 2)
195          denumerator = 2 * (n ** 2)
196          return numerator / denumerator
```

File - QuantumFactory.py

```python

    @staticmethod
    def get_delta_energy_unperturbed(n: int, n_prime: int,
                                     z=sp.Symbol('Z', real=True), mu=sp.Symbol('mu', real=True)) -> float:
        """
        Getter of the transition energy without any perturbation
        :param n: initial orbital number n (int)
        :param n_prime: final orbital number n (int)
        :param z: atomic number (float)
        :param mu: reduced mass (float)
        :return: transition energy (float) or transition energy (sympy object)
        """
        e = QuantumFactory.get_state_energy_unperturbed(n, z, mu)
        e_prime = QuantumFactory.get_state_energy_unperturbed(n_prime, z, mu)
        return e - e_prime

    @staticmethod
    def get_transition_angular_frequency_unperturbed(n: int, n_prime: int,
                                                     z=sp.Symbol('Z', real=True), mu=sp.Symbol('mu', real=True)):
        """
        Getter of the transition angular frequency without any perturbation
        :param n: initial orbital number n (int)
        :param n_prime: final orbital number n (int)
        :param z: atomic number (float)
        :param mu: reduced mass (float)
        :return: angular frequency (float) or angular frequency (sympy object)
        """
        return QuantumFactory.get_delta_energy_unperturbed(n, n_prime, z, mu) / const.hbar

    @staticmethod
    def decay_number(n: int, T: float, z: int = const.Z_H, mu: float = const.mu_H):
        """
        Return the decay number of the level n for unperturbed energy.
        :param n: orbital number n (int)
        :param T: Current temperature (float)
        :param z: atomic number (int)
        :param mu: reduced mass (float)
        :return: a sympy expression of the decay number (sympy object)
        """
        g = QuantumFactory.get_g_unperturbed(n)
        return g * sp.exp(-QuantumFactory.get_state_energy_unperturbed(n, z, mu) / (const.k_B * T))

    @staticmethod
    def decay_number_ratio(n: int, n_prime: int, T: float, z: int = const.Z_H, mu: float = const.mu_H):
        """
        Return the ratio of decay number of the levels n to n_prime for unperturbed energy.
        :param n: initial orbital number n (int)
        :param n_prime: final orbital number n (int)
        :param T: Current temperature (float)
        :param z: atomic number (int)
        :param mu: reduced mass (float)
        :return: the ration of decay number (float)
        """
        N_i = QuantumFactory.decay_number(n, T, z, mu)
        N_j = QuantumFactory.decay_number(n_prime, T, z, mu)
        return (N_i / N_j).evalf()

    @staticmethod
    def intensity_of_the_beam(n: int, n_prime: int, T: float, z: int = const.Z_H, mu: float = const.mu_H):
        """
        Gives the intensity of the beam with a proportional function alpha
        :param n: initial orbital number n (int)
        :param n_prime: final orbital number n (int)
        :param T: Current temperature (float)
        :param z: atomic number (int)
        :param mu: reduced mass (float)
        :return: intensity of the beam (sympy object)
        """
        from Transitions import Transitions
        transitions = Transitions(n, n_prime)
        alpha = sp.Symbol('alpha')  # proportional function
        omega = QuantumFactory.get_transition_angular_frequency_unperturbed(n, n_prime, z, mu)
        N = QuantumFactory.decay_number(n, T, z, mu)
        Rs_mean = transitions.get_spontaneous_decay_mean(z, mu)
        return alpha * N * omega * Rs_mean

    @staticmethod
    def relative_intensity_of_the_beam(n: int, n_prime: int, i: int, j: int,
                                       T: float, z: int = const.Z_H, mu: float = const.mu_H):
        """
        Gives the intensity of the beam normalized with the intensity of another transition beam
        :param n: initial orbital number n (int)
        :param n_prime: final orbital number n (int)
        :param i: initial orbital number n of the base transition (int)
        :param j: final orbital number n of the base transition (int)
        :param T: Current temperature (float)
        :param z: atomic number (int)
        :param mu: reduced mass (float)
        :return: intensity of the beam normalized (float)
        """
        I1 = QuantumFactory.intensity_of_the_beam(n, n_prime, T, z, mu)
        I2 = QuantumFactory.intensity_of_the_beam(i, j, T, z, mu)
        return (I1 / I2).evalf()

    @staticmethod
    def bracket_product(wave_function, wave_function_prime, operator=None, algo="mcint"):
        """
        Call the algo function to make the scalar product <bra|operator|ket>
```

File - QuantumFactory.py

```
295         :param wave_function: bra
296         :param wave_function_prime: ket
297         :param operator: operator
298         :param algo: algo to use to compute the integral (str) element of
299                      {sympy, sympy_ns, scipy_nquad, scipy_tplquad, mcint}
300         :return: the result of algo(wave_function, wave_function_prime, operator)
301         """
302         algos = {"sympy": QuantumFactory.bracket_product_sympy,
303                  "sympy_ns": QuantumFactory.bracket_product_sympy_ns,
304                  "scipy_nquad": QuantumFactory.bracket_product_scipy_nquad,
305                  "scipy_tplquad": QuantumFactory.bracket_product_scipy_tplquad,
306                  "mcint": QuantumFactory.bracket_product_mcint}
307         assert algo in algos.keys()
308         return algos[algo](wave_function, wave_function_prime, operator)
309 
310     @staticmethod
311     def bracket_product_sympy(wave_function, wave_function_prime, operator=None):
312         """
313         Compute the scalar product <bra|operator|ket> with sympy
314         :param wave_function: bra
315         :param wave_function_prime: ket
316         :param operator: operator
317         :return: bracket product (float)
318         """
319         r, theta, phi = sp.Symbol("r", real=True, positive=True), sp.Symbol("theta", real=True), sp.Symbol("phi",
320                                                                                                             real=True)
321         jacobian = (r ** 2) * sp.sin(theta)
322         integral_core = sp.conjugate(wave_function) * (operator if operator is not None else 1) * wave_function_prime
323         integral_core_expension = sp.expand(sp.FU["TR8"](jacobian * integral_core), func=True).simplify()
324 
325         # creation of the Integral object and first try to resolve it
326         bracket_product = sp.Integral(integral_core_expension,
327                                       (phi, 0, 2 * mpmath.pi), (r, 0, mpmath.inf), (theta, 0, mpmath.pi),
328                                       risch=False).doit()
329 
330         # simplify the result of the first try and evaluation of the integral, last attempt
331         bracket_product = bracket_product.simplify().evalf(n=50, maxn=3_000, strict=True)
332         return bracket_product
333 
334     @staticmethod
335     def bracket_product_sympy_ns(wave_function, wave_function_prime, operator=None):
336         """
337         Compute the scalar product <bra|operator|ket> without simplification to use the symbolic integration of sympy
338         :param wave_function: bra
339         :param wave_function_prime: ket
340         :param operator: operator
341         :return: bracket product (float)
342         """
343         r, theta, phi = sp.Symbol("r", real=True, positive=True), \
344                         sp.Symbol("theta", real=True), sp.Symbol("phi", real=True)
345         jacobian = (r ** 2) * sp.sin(theta)
346         integral_core = wave_function * (operator if operator is not None else 1) * sp.conjugate(wave_function_prime)
347 
348         # creation of the Integral object and first try to resolve it
349         bracket_product = sp.integrate(jacobian * integral_core,
350                                        (phi, 0, 2 * np.pi), (r, 0, np.inf), (theta, 0, np.pi),
351                                        risch=False)
352 
353         # simplify the result of the first try and evaluation of the integral, last attempt
354         bracket_product = bracket_product.simplify().evalf(n=50, maxn=3_000, strict=True)
355         return bracket_product
356 
357     @staticmethod
358     def bracket_product_scipy_nquad(wave_function, wave_function_prime, operator=None):
359         """
360         Compute the scalar product <bra|operator|ket> with scipy and the integrate algorithm nquad
361         :param wave_function: bra
362         :param wave_function_prime: ket
363         :param operator: operator
364         :return: bracket product (float)
365         """
366         r, theta, phi = sp.Symbol("r", real=True), sp.Symbol("theta", real=True), sp.Symbol("phi", real=True)
367         jacobian = (r ** 2) * sp.sin(theta)
368         integral_core = sp.conjugate(wave_function) * (operator if operator is not None else 1) * wave_function_prime
369 
370         integral_core_expension = sp.expand(sp.FU["TR8"](jacobian * integral_core), func=True).simplify()
371 
372         def bound_r(_):
373             return [0, mpmath.inf]
374 
375         def bound_phi(_, __):
376             return [0, 2 * mpmath.pi]
377 
378         def bound_theta():
379             return [0, mpmath.pi]
380 
381         # Process the integral with scipy
382         integral_core_lambdify = sp.lambdify((theta, r, phi), integral_core_expension, modules="numpy")
383         bracket_product = QuantumFactory.complex_quadrature(sc.integrate.nquad, integral_core_lambdify,
384                                                             [bound_phi, bound_r, bound_theta])[0]
385         return bracket_product
386 
387     @staticmethod
388     def bracket_product_scipy_tplquad(wave_function, wave_function_prime, operator=None):
389         """
390         Compute the scalar product <bra|operator|ket> with scipy and the integrate algorithm tplquad
391         :param wave_function: bra
392         :param wave_function_prime: ket
```

```
393         :param operator: operator
394         :return: bracket product (float)
395         """
396         from sympy.utilities.lambdify import lambdastr
397         r, theta, phi = sp.Symbol("r", real=True), sp.Symbol("theta", real=True), sp.Symbol("phi", real=True)
398         jacobian = (r ** 2) * sp.sin(theta)
399         integral_core = sp.conjugate(wave_function) * (operator if operator is not None else 1) * wave_function_prime
400 
401         # integral_core_expension = sp.expand(sp.FU["TR8"](jacobian * integral_core), func=True).simplify()
402         integral_core_expension = sp.expand(jacobian * integral_core, func=True).simplify()
403 
404         # Process the integral with scipy
405         integral_core_lambdify = sp.lambdify((theta, r, phi), integral_core_expension, modules="numpy")
406         print(lambdastr((theta, r, phi), integral_core_expension))
407         bracket_product = QuantumFactory.complex_quadrature(sc.integrate.tplquad, integral_core_lambdify,
408                                                             0, 2 * np.pi, lambda b_phi: 0, lambda b_phi: np.inf,
409                                                             lambda b_theta, b_r: 0, lambda b_theta, b_r: np.pi)[0]
410         # bracket_product = sc.integrate.tplquad(integral_core_lambdify,
411         #                                        0, 2 * np.pi, lambda b_phi: 0, lambda b_phi: np.inf,
412         #                                        lambda b_theta, b_r: 0, lambda b_theta, b_r: np.pi)[0]
413         return bracket_product
414 
415     @staticmethod
416     def complex_quadrature(integrate_func, func, *args, **kwargs):
417         """
418         Gives the complex quadrature of a function
419         :param integrate_func: the integration function
420         :param func: the actual function to integrate
421         :param args: args to pass to integrate_func
422         :param kwargs: kwargs to pass to integrate_func
423         :return: (the complex quadrature, real error, imag error) (tuple)
424         """
425 
426         def real_func(x, y, z):
427             return sc.real(func(x, y, z))
428 
429         def imag_func(x, y, z):
430             return sc.imag(func(x, y, z))
431 
432         real_integral = integrate_func(real_func, *args, **kwargs)
433         imag_integral = integrate_func(imag_func, *args, **kwargs)
434         return real_integral[0] + 1j * imag_integral[0], real_integral[1:], imag_integral[1:]
435 
436     @staticmethod
437     def bracket_product_mcint(wave_function, wave_function_prime, operator=None):
438         """
439         Compute the scalar product <bra|operator|ket> with a monte carlo integration algorithm
440         :param wave_function: bra
441         :param wave_function_prime: ket
442         :param operator: operator
443         :return: bracket product (float)
444         """
445         r, theta, phi = sp.Symbol("r", real=True), sp.Symbol("theta", real=True), sp.Symbol("phi", real=True)
446         jacobian = (r ** 2) * sp.sin(theta)
447         integral_core = sp.conjugate(wave_function) * (operator if operator is not None else 1) * wave_function_prime
448 
449         # integral_core_expension = sp.expand(sp.FU["TR8"](jacobian * integral_core), func=True).simplify()
450         integral_core_expension = sp.expand(jacobian * integral_core, func=True).simplify()
451 
452         integral_core_lambdify = sp.lambdify((theta, r, phi), integral_core_expension, modules="numpy")
453 
454         def integrand(x):
455             theta, r, phi = x[0], x[1], x[2]
456             return integral_core_lambdify(theta, r, phi)
457 
458         domainsize = (0, 1)
459         result = QuantumFactory.monte_carlo_integration_sph(integrand, [[0, 1.15], [0, np.pi], [0, 2 * np.pi]],
460                                                             domainsize, 1_000_000)
461         return result
462 
463     @staticmethod
464     def monte_carlo_integration(integrand, bornes: list, domainsize=(0.0, 1.0), n_sample: int = int(1e6)):
465         """
466         Compute the integral with a monte carlo integration algorithm
467         :param integrand: the function to integrate (lambda or func)
468         :param bornes:
469         :param domainsize: (tuple of len==2)
470         :param n_sample: number of sample to use (int)
471         :return: the integral of the integrand (float)
472         """
473         np.random.seed(1)
474         # Sum elements and elements squared
475         total = 0.0
476         total_sq = 0.0
477         count_in_curve = 0
478 
479         def sampler():
480             while True:
481                 yield [np.random.uniform(b0, b1) for [b0, b1] in bornes]
482 
483         for x in itertools.islice(sampler(), n_sample):
484             f = integrand(x)
485             f_rn = np.random.uniform(domainsize[0], domainsize[1])
486             total += f
487             total_sq += (f ** 2)
488             count_in_curve += 1 if 0 <= f_rn <= f else 0
489         # Return answer
490         # sample_mean = total / n_sample
```

```python
        # sample_var = (total_sq - ((total / n_sample) ** 2) / n_sample) / (n_sample - 1.0)
        # return domainsize * sample_mean, domainsize * np.sqrt(sample_var / n_sample)
        v = np.prod([b1 - b0 for [b0, b1] in bornes]) * (domainsize[1] - domainsize[0])
        print(100 * (count_in_curve / n_sample), "%")
        return (count_in_curve / n_sample) * v

    @staticmethod
    def monte_carlo_integration_sph(integrand, bornes: list, domainsize=(0.0, 1.0), n_sample: int = int(1e6)):
        """
        Compute the integral with a spherical monte carlo integration algorithm
        :param integrand: the function to integrate (lambda or func)
        :param bornes:
        :param domainsize: (tuple of len==2)
        :param n_sample: number of sample to use (int)
        :return: the integral of the integrand (float)
        """
        # np.random.seed(1)
        # Sum elements and elements squared

        # def sampler():
        #     while True:
        #         yield [np.random.uniform(b0, b1) for [b0, b1] in bornes]

        samples = np.array([[np.random.uniform(b0, b1) for [b0, b1] in bornes] for _ in range(n_sample)])

        # f_samples = np.array([np.random.uniform(domainsize[0], domainsize[1]) for _ in range(n_sample)])
        # for x in itertools.islice(sampler(), n_sample):
        # f_values = integrand(samples)
        # @numba.generated_jit(nopython=True)
        def loop():
            count: int = int(0)
            for i in prange(len(samples)):
                f = integrand(samples[i])
                # f = f_values[i]
                f_rn = np.random.uniform(domainsize[0], domainsize[1])
                # if (f > 0 and 0 <= f_samples[i] <= f) or (f < 0 and f <= f_samples[i] <= 0):
                if (f > 0 and 0 <= f_rn <= f) or (f < 0 and f <= f_rn <= 0):
                    count += 1
            return count

        count_in_curve = loop()
        # Return answer
        # sample_mean = total / n_sample
        # sample_var = (total_sq - ((total / n_sample) ** 2) / n_sample) / (n_sample - 1.0)
        # return domainsize * sample_mean, domainsize * np.sqrt(sample_var / n_sample)
        # v = ((4*np.pi*(bornes[0][1] - bornes[0][0])**3)/3) * (domainsize[1] - domainsize[0])
        # print(sp.integrate((4/3)*sp.pi*sp.Symbol('r')**3, (sp.Symbol('z'), domainsize[0], domainsize[1])))
        v = sp.integrate((4 / 3) * sp.pi * sp.Symbol('r') ** 3, (sp.Symbol('z'), domainsize[0], domainsize[1])).evalf(
            subs={'r': bornes[0][1] - bornes[0][0]})
        # print(100 * (count_in_curve / n_sample), "%")
        return (count_in_curve / n_sample) * v


if __name__ == '__main__':
    from Transitions import Transitions
    from QuantumState import QuantumState

    qs1 = QuantumState(n=2, ell=1, m_ell=1, s=0.5, m_s=0.5)
    qs2 = QuantumState(n=1, ell=0, m_ell=0, s=0.5, m_s=0.5)

    y1 = QuantumFactory.Y_ell_m_ell(3, 2)
    y2 = QuantumFactory.Y_ell_m_ell(2, 1)

    print(QuantumFactory.bracket_product(y1, y1))
    print(QuantumFactory.bracket_product(y1, y2))
    print(QuantumFactory.bracket_product(y2, y2))

```

File - QuantumState.py

```python
import numpy as np
import Constants as const
import sympy as sp
from QuantumFactory import QuantumFactory


class QuantumState:

    def __init__(self, n: int, ell: int, m_ell: int, s: float, m_s: float, hydrogen: bool = False):
        """
        QuantumState constructor
        :param n: orbital number (int)
        :param ell: angular momentum (int)
        :param m_ell: quantum number m_ell (int)
        :param s: spin (float)
        :param m_s: quantum number m_s (float)
        :param hydrogen : if the current quantum state refer to a hydrogen atom (bool)
        """
        self._n: int = n
        self._ell: int = ell
        self._m_ell: int = m_ell
        self._s: float = s
        self._m_s: float = m_s
        self.hydrogen: bool = hydrogen
        self.check_invariants()

    def check_invariants(self) -> None:
        """
        Check every invariant for a quantum state
        :return: None
        """
        assert self._n >= 1
        assert 0 <= self._ell < self._n
        assert -self._ell <= self._m_ell <= self._ell
        assert self._s >= 0.0
        assert -self._s <= self._m_s <= self._s

    def getState(self) -> np.ndarray:
        """
        Getter of the state vector of state numbers in order (n, ell, m_ell, s, m_s)
        :return: [n, ell, m_ell, s, m_s] (numpy.ndarray)
        """
        return np.array([self._n, self._ell, self._m_ell, self._s, self._m_s])

    def get_n(self) -> int:
        return self._n

    def get_ell(self) -> int:
        return self._ell

    def get_m_ell(self) -> int:
        return self._m_ell

    def get_s(self) -> float:
        return self._s

    def get_m_s(self) -> float:
        return self._m_s

    def get_state_energy(self, z=sp.Symbol("Z", real=True), mu=sp.Symbol('mu', real=True)):
        """
        Get the energy of the current quantum state
        :param z: atomic number
        :param mu: reduced mass
        :return: the energy of the current quantum state (float if z and mu are float else sympy object)
        """
        return QuantumFactory.get_state_energy_unperturbed(self._n, z, mu)

    def get_valid_transitions_state_to_n(self, other_n: int) -> list:
        """
        Get all of the valid transition of the current quantum state to another in the orbital other_n
        :param other_n: (int)
        :return: list of QuantumState
        """
        import warnings
        warnings.warn("Warning! This method seems to be not valid", DeprecationWarning)
        from Transition import Transition
        valid_transitions = list()
        next_states = set()
        for key, possibilities in Transition.TRANSITIONS_RULES.items():
            for num in possibilities:
                for key_prime, possibilities_prime in Transition.TRANSITIONS_RULES.items():
                    if key == key_prime:
                        continue
                    for num_prime in possibilities_prime:
                        (_, other_ell, other_m_ell, other_s, other_m_s) = tuple(self.getState())
                        if key == "Delta ell":
                            other_ell += num
                        elif key == "Delta m_ell":
                            other_m_ell += num
                        elif key == "Delta s":
                            other_s += num
                        elif key == "Delta m_s":
                            other_m_s += num

                        if key_prime == "Delta ell":
                            other_ell += num_prime
                        elif key_prime == "Delta m_ell":
```

File - QuantumState.py

```
                            other_m_ell += num_prime
                        elif key_prime == "Delta s":
                            other_s += num_prime
                        elif key_prime == "Delta m_s":
                            other_m_s += num_prime
                        try:
                            next_state = QuantumState(other_n, other_ell, other_m_ell, other_s, other_m_s)
                        except AssertionError:
                            continue
                        if Transition.possible(self, next_state):
                            if str(next_state) not in next_states:
                                valid_transitions.append(next_state)
                            next_states.add(str(next_state))
                        else:
                            continue

        return valid_transitions

    def __repr__(self) -> str:
        """
        show a representation of the current quantum state
        :return: string representation of self (str)
        """
        this_repr = f"(n: {self._n}, " \
                    f"ell: {self._ell}, " \
                    f"m_ell: {self._m_ell}, " \
                    f"s: {self._s}, " \
                    f"m_s: {self._m_s})"
        return this_repr

    def repr_without_spin(self) -> str:
        """
        show a representation of the current quantum state without s and m_s
        :return: string representation of self without orbital spin (str)
        """
        this_repr = f"(n: {self._n}, " \
                    f"ell: {self._ell}, " \
                    f"m_ell: {self._m_ell})"
        return this_repr

    def get_wave_function(self, z=sp.Symbol("Z", real=True), mu=sp.Symbol('mu', real=True)):
        """
        Get the wave function of the current quantum state as a sympy object
        :param z: atomic number
        :param mu: reduced mass (float)
        :return: the wave function (sympy object)
        """
        if self.hydrogen:
            return QuantumFactory.get_hydrogen_wave_function(self._n, self._ell, self._m_ell)

        u = QuantumFactory.u_n_ell(n=self._n, ell=self._ell, z=z, mu=mu)
        y = QuantumFactory.Y_ell_m_ell(self._ell, self._m_ell)
        return u*y

    def decay_number(self, T: float, z: int = const.Z_H, mu: float = const.mu_H):
        """
        Return the decay number of the current QuantumState.
        :param T: Current temperature (float)
        :param z: atomic number (int)
        :param mu: reduced mass (float)
        :return: a sympy expression of the decay number (sympy object)
        """
        alpha = sp.Symbol('alpha')  # proportional function
        return alpha*QuantumFactory.get_g_n(self._n)*sp.exp(-self.get_state_energy(z, mu)/(const.k_B*T))


if __name__ == '__main__':
    from Transition import Transition
    quantum_state = QuantumState(n=1, ell=0, m_ell=0, s=1 / 2, m_s=1 / 2)
    print(f"psi_{quantum_state} = {quantum_state.get_wave_function()}")
    print(f"E_{quantum_state} = {quantum_state.get_state_energy()}")
    print(Transition.possible(quantum_state, QuantumState(2, 0, 0, 1 / 2, 1 / 2)))
    print(Transition.possible(quantum_state, QuantumState(2, 1, 0, 1/2, 1 / 2)))

    valid_transitions_test_1_to_2 = [QuantumState(2, 1, -1, 1/2, 1/2),
                                     QuantumState(2, 1, 0, 1/2, 1/2),
                                     QuantumState(2, 1, 1, 1/2, 1/2)]

    for state in QuantumState(n=1, ell=0, m_ell=0, s=1/2, m_s=1/2).get_valid_transitions_state_to_n(other_n=2):
        print(state.getState())

    print('-'*175)

    valid_transitions_test_3_to_1 = [QuantumState(1, 0, 0, 1/2, 1/2)]

    for state in QuantumState(n=3, ell=1, m_ell=0, s=1/2, m_s=1/2).get_valid_transitions_state_to_n(other_n=1):
        print(state.getState())

    print('-'*175)

    print(QuantumFactory.get_valid_ell_with_n(3))

    print('-' * 175)

    for valid_transition in QuantumFactory.get_valid_transitions_n_to_n(1, 2):
        print(valid_transition)
```

File - Transition.py

```python
from QuantumState import QuantumState
import Constants as const
import sympy as sp
import numpy as np
import numba
from QuantumFactory import QuantumFactory
import os


class Transition:
    TRANSITIONS_RULES: dict = {"Delta ell": [-1, 1],
                               "Delta m_ell": [-1, 0, 1],
                               "Delta s": [0],
                               "Delta m_s": [0]}

    TRANSITIONS_RULES_SPIN: dict = {"Delta ell": [-1, 1],
                                    "Delta m_ell": [-1, 0, 1],
                                    "Delta j": [-1, 0, 1],
                                    "Delta m_j": [-1, 0, 1]}

    n_ell_m_ell_state_to_rs: dict = dict()

    transition_count: int = int(0)

    static_save_file = os.getcwd() + "/Transition_static_values.npy"

    def __init__(self, initial_quantum_state: QuantumState, ending_quantum_state: QuantumState):
        """
        Transition constructor
        :param initial_quantum_state: initial quantum state (QuantumState)
        :param ending_quantum_state: final quantum state (QuantumState)
        """
        self._initial_quantum_state = initial_quantum_state
        self._ending_quantum_state = ending_quantum_state
        self.check_invariant()
        self.spontaneous_decay_rate = None
        Transition.transition_count += 1
        self.load_static_values()

    def __del__(self):
        Transition.save_static_values()
        Transition.transition_count -= 1

    @staticmethod
    def save_static_values() -> None:
        if Transition.transition_count >= 1:
            np.save(Transition.static_save_file, Transition.n_ell_m_ell_state_to_rs)

    @staticmethod
    def load_static_values() -> None:
        if Transition.transition_count >= 1 and os.path.exists(Transition.static_save_file):
            Transition.n_ell_m_ell_state_to_rs = np.load(Transition.static_save_file, allow_pickle=True).item()

    @staticmethod
    def clear_static_values() -> None:
        os.remove(Transition.static_save_file)

    def check_invariant(self) -> None:
        """
        Check every invariant for a Transition
        :return: None
        """
        assert Transition.possible(self._initial_quantum_state, self._ending_quantum_state)

    def get_n_to_n_prime_couple(self) -> tuple:
        """
        Gives the tuple (initial_quantum_state.n, ending_quantum_state.n)
        :return: (tuple)
        """
        return self._initial_quantum_state.get_n(), self._ending_quantum_state.get_n()

    def __repr__(self) -> str:
        """
        show a representation of the current Transition
        :return: string representation of self (str)
        """
        this_repr = f"({self._initial_quantum_state} -> {self._ending_quantum_state})"
        return this_repr

    def repr_without_spin(self) -> str:
        """
        show a representation of the current Transition without s and m_s
        :return: string representation of self without orbital spin (str)
        """
        this_repr = f"({self._initial_quantum_state.repr_without_spin()} " \
                    f"-> {self._ending_quantum_state.repr_without_spin()})"
        return this_repr

    def get_spontaneous_decay_rate(self, z=sp.Symbol('Z', real=True), mu=sp.Symbol('mu', real=True), algo="auto"):
        """
        Get the spontaneous decay rate of the transition
        :param z: atomic number (int)
        :param mu: reduced mass (float)
        :return: the spontaneous decay rate (float)
        """
        # check if spontaneous_decay_rate is already calculated
        if self.spontaneous_decay_rate is not None:
            return self.spontaneous_decay_rate
```

File - Transition.py

```
 99         elif self.repr_without_spin() in Transition.n_ell_m_ell_state_to_rs.keys():
100             self.spontaneous_decay_rate = Transition.n_ell_m_ell_state_to_rs[self.repr_without_spin()]
101             return self.spontaneous_decay_rate
102 
103         r, theta, phi = sp.Symbol("r", real=True, positive=True), sp.Symbol("theta", real=True),\
104                         sp.Symbol("phi", real=True)
105         coeff = (4*const.alpha*(self.get_delta_energy(z, mu)**3))/(3*(const.hbar**3)*(const.c**2))
106         psi = self._initial_quantum_state.get_wave_function(z, mu)
107         psi_prime = self._ending_quantum_state.get_wave_function(z, mu)
108 
109         if algo == "auto":
110             if self._initial_quantum_state.hydrogen and self._ending_quantum_state.hydrogen:
111                 algo = "sympy_ns"
112             else:
113                 algo = "scipy_nquad"
114 
115         # \Vec{r} = x + y + z
116         r_operator = r * (sp.sin(theta)*sp.cos(phi) + sp.sin(theta)*sp.sin(phi) + sp.cos(theta))
117 
118         # Process the integral with algo
119         bracket_product = QuantumFactory.bracket_product(psi, psi_prime, r_operator, algo)
120 
121         bracket_product_norm_square = sp.Mul(sp.conjugate(bracket_product), bracket_product).evalf()
122         # print(bracket_product_norm_square)
123 
124         self.spontaneous_decay_rate = sp.Float(coeff * bracket_product_norm_square)
125 
126         # updating Transition static attribute
127         Transition.n_ell_m_ell_state_to_rs[self.repr_without_spin()] = self.spontaneous_decay_rate
128         return self.spontaneous_decay_rate
129 
130     def get_delta_energy(self, z=sp.Symbol('Z', real=True), mu=sp.Symbol('mu', real=True)):
131         """
132         Getter of the transition energy without any perturbation
133         :param z: atomic number (float)
134         :param mu: reduced mass (float)
135         :return: transition energy (float) or transition energy (sympy object)
136         """
137         return self._initial_quantum_state.get_state_energy(z, mu) - self._ending_quantum_state.get_state_energy(z, mu)
138 
139     def get_angular_frequency(self, z=sp.Symbol('Z', real=True), mu=sp.Symbol('mu', real=True)):
140         """
141         Getter of the transition angular frequency without any perturbation
142         :param z: atomic number (float)
143         :param mu: reduced mass (float)
144         :return: angular frequency (float) or angular frequency (sympy object)
145         """
146         return self.get_delta_energy(z, mu)/const.hbar
147 
148     @staticmethod
149     def possible(initial_quantum_state: QuantumState, ending_quantum_state: QuantumState) -> bool:
150         """
151         Check if the transition is possible with every transition rule selection
152         :param initial_quantum_state: initial quantum state (QuantumState)
153         :param ending_quantum_state: final quantum state (QuantumState)
154         :return: a boolean representation of the capability of the transition (bool)
155         """
156         able = True
157 
158         if initial_quantum_state.get_n() == ending_quantum_state.get_n():
159             able = False
160 
161         if (initial_quantum_state.get_ell() - ending_quantum_state.get_ell()) \
162                 not in Transition.TRANSITIONS_RULES["Delta ell"]:
163             able = False
164 
165         if (initial_quantum_state.get_m_ell() - ending_quantum_state.get_m_ell()) \
166                 not in Transition.TRANSITIONS_RULES["Delta m_ell"]:
167             able = False
168 
169         if (initial_quantum_state.get_s() - ending_quantum_state.get_s()) \
170                 not in Transition.TRANSITIONS_RULES["Delta s"]:
171             able = False
172 
173         if (initial_quantum_state.get_m_s() - ending_quantum_state.get_m_s()) \
174                 not in Transition.TRANSITIONS_RULES["Delta m_s"]:
175             able = False
176 
177         return able
178 
179 
180 if __name__ == '__main__':
181     import time
182     start_time = time.time()
183 
184     qs1 = QuantumState(n=2, ell=1, m_ell=0, s=0.5, m_s=0.5, hydrogen=True)
185     qs2 = QuantumState(n=1, ell=0, m_ell=0, s=0.5, m_s=0.5, hydrogen=True)
186 
187     print(f"psi_1{qs1} = {qs1.get_wave_function()}")
188     print(f"psi_2{qs2} = {qs2.get_wave_function()}")
189 
190     rs_mean_norm_coeff = ((const.alpha ** 5) * (const.c ** 2) * const.mu_H) / const.hbar
191     omega_normalized_coeff = ((const.alpha ** 2) * const.mu_H * (const.c ** 2)) / (2 * const.hbar)
192 
193     trans = Transition(qs1, qs2)
194 
195     print(Transition.n_ell_m_ell_state_to_rs)
196 
```

File - Transition.py

```
197     print('-' * 175)
198     rs_normalized = trans.get_spontaneous_decay_rate(z=const.Z_H, mu=const.mu_H, algo="auto") / rs_mean_norm_coeff
199     reel_rs_normalized = 0.002746686
200     print(f"Transition: {trans}")
201     print(f"R^s sympy = {rs_normalized}")
202     print(f"reel R^s = {reel_rs_normalized},"
203           f" deviation: {100*(np.abs(rs_normalized-reel_rs_normalized))} %")
204     print('-'*175+f'\n elapse time: {time.time()-start_time}')
205
206     start_time = time.time()
207
208     print(Transition.transition_count)
209     del(trans)
210     print(Transition.transition_count)
211
212
213
```

File - Transitions.py

```python
from Transition import Transition
import Constants as const
import numpy as np
import sympy as sp
import tqdm


class Transitions(list):
    def __init__(self, n: int = None, n_prime: int = None, hydrogen: bool = False):
        """
        Transitions constructor. Transitions is a container of Transition object
         that can calculate some stats on these transitions
        :param n: initial orbital number n (int)
        :param n_prime: final orbital number n (int)
        :param hydrogen : if the current container of quantum states refer to hydrogen atoms (bool)
        """
        super().__init__()
        assert (n is None and n_prime is None) or (isinstance(n, int) and isinstance(n, int)),\
            "params n and n_prime must be integer or None"
        if isinstance(n, int) and isinstance(n, int):
            self.append_transitions_n_to_n(n, n_prime, hydrogen=hydrogen)
        self.spontaneous_decay_mean: float = None

    def append(self, transition: Transition) -> None:
        """
        Add a transition in the current container
        :param transition: Transition to be added (Transition)
        :return: None
        """
        super().append(transition)

    def append_transitions_n_to_n(self, n, n_prime, hydrogen: bool = False) -> None:
        """
        Add all the possible transition between orbital number n and n_prime
        :param n: initial orbital number n (int)
        :param n_prime: final orbital number n (int)
        :param hydrogen: if we want to cast quantum state in quantum hydrogen state (bool)
        :return: None
        """
        from QuantumFactory import QuantumFactory
        for trans in QuantumFactory.get_valid_transitions_n_to_n_prime(n, n_prime, hydrogen=hydrogen):
            self.append(trans)

    def get_spontaneous_decay_mean(self, z=const.Z_H, mu=const.mu_H, verbose: bool = False) -> float:
        """
        Get mean of the spontaneous decay rate of transitions in the current container
        :param z:  atomic number (float)
        :param mu: reduced mass (float)
        :param verbose: if True, will show elapse time (bool)
        :return: mean spontaneous decay rate (float)
        """
        if self.spontaneous_decay_mean is not None:
            return self.spontaneous_decay_mean

        if verbose:
            rs_vector: np.ndarray = np.array([trans.get_spontaneous_decay_rate(z=z, mu=mu) for trans in tqdm.tqdm(self
)])
        else:
            rs_vector: np.ndarray = np.array([trans.get_spontaneous_decay_rate(z=z, mu=mu) for trans in self])

        self.spontaneous_decay_mean: float = np.float(np.mean(rs_vector))
        return self.spontaneous_decay_mean

    @staticmethod
    def get_angular_frequency(n, n_prime, z=sp.Symbol("Z", real=True), mu=sp.Symbol('mu', real=True)) -> float:
        """
        Get the angular frequency between two states using the unperturbed energy.
        :param n: initial orbital number n (int)
        :param n_prime: final orbital number n (int)
        :param z: atomic number (float)
        :param mu: reduced mass (float)
        :return: angular frequency (float)
        """
        import warnings
        warnings.warn("Warning! This method seems to be replicated and not efficient", DeprecationWarning)
        e = (- (z ** 2) * (const.alpha ** 2) * mu * (const.c ** 2))/(2 * (n ** 2))
        e_prime = (- (z ** 2) * (const.alpha ** 2) * mu * (const.c ** 2))/(2 * (n_prime ** 2))
        return (e - e_prime)/const.hbar

    def __repr__(self) -> str:
        """
        String representation of the current object
        :return: string representation (str)
        """
        this_repr = "[ "
        for trans in self:
            this_repr += f"{trans}, \n"
        this_repr += "]"
        return this_repr

    def get_n_to_n_prime_couple(self) -> set:
        """
        gives a set of every tuple (n, n_prime) of the current container
        :return: tuple (n, n_prime) (set)
        """
        couple_set: set = set()
        for trans in self:
            couple_set.add(trans.get_n_to_n_prime_couple())
```

File - Transitions.py

```
 98         return couple_set
 99
100     def intensity_of_the_beam(self, T: float, z: int = const.Z_H, mu: float = const.mu_H) -> np.ndarray:
101         from QuantumFactory import QuantumFactory
102         I: list = list()
103         alpha = sp.Symbol('alpha')  # proportional function
104         for couple in self.get_n_to_n_prime_couple():
105             omega = QuantumFactory.get_transition_angular_frequency_unperturbed(couple[0], couple[1], z, mu)
106             N = QuantumFactory.decay_number(couple[0], T, z, mu)
107             Rs_mean = self.get_spontaneous_decay_mean(z, mu)
108             I.append(alpha*N*omega*Rs_mean)
109         return np.array(I)
110
111     def relative_intensity_of_the_beam(self, T: float, z: int = const.Z_H, mu: float = const.mu_H):
112         I_ratio: list = list()
113         for couple in self.get_n_to_n_prime_couple():
114             I_ratio.append(self.intensity_of_the_beam(T, z, mu))
115
116     def save(self):
117         raise NotImplemented("This method is not implemented yet")
118
119
120 if __name__ == '__main__':
121     n, n_prime = 6, 2
122
123     transitions_n_to_n_prime = Transitions(n=n, n_prime=n_prime, hydrogen=True)
124     print(transitions_n_to_n_prime)
125
126
```

File - main.py

```python
from Transitions import Transitions
from QuantumFactory import QuantumFactory
import Constants as const
from geometric_calculus import Geometric_calculus
import numpy as np
import time


def tab_cell(n, n_prime):
    rs_answ = {"(3, 2)": 0.0027466865777777777777777777778,
               "(4, 2)": 0.00052436,
               "(5, 2)": 0.000157596,
               "(6, 2)": 0.000060611}

    om_answ = {"(3, 2)": 5/36,
               "(4, 2)": 3/16,
               "(5, 2)": 21/100,
               "(6, 2)": 2/9}

    rs_mean_normalized_coeff = ((const.alpha ** 5) * const.mu_H * (const.c ** 2)) / const.hbar
    omega_normalized_coeff = ((const.alpha ** 2) * const.mu_H * (const.c ** 2)) / (2 * const.hbar)

    print(f'-'*75)

    print(f"Transition ({n} -> {n_prime}) : \n")

    transitions_n_to_n_prime = Transitions(n=n, n_prime=n_prime, hydrogen=True)
    # print(f"Transitions: {transitions_n_to_n_prime} \n")

    transitions_n_to_n_prime_rs_mean = transitions_n_to_n_prime.get_spontaneous_decay_mean() / rs_mean_normalized_coeff
    print(f"R^s_mean / rs_mean_normalized_coeff  = {transitions_n_to_n_prime_rs_mean:.5e}")

    reel_rs_mean = rs_answ[str((n, n_prime))] if str((n, n_prime)) in rs_answ else 0.0

    print(f"reference R^s  / rs_mean_normalized_coeff = {reel_rs_mean:.5e} \n")

    omega = QuantumFactory.get_transition_angular_frequency_unperturbed(n, n_prime, const.Z_H, const.mu_H)
    omega_normalized = omega / omega_normalized_coeff

    print(f"omega / omega_normalized_coeff  =  {omega_normalized:.5e}")

    reel_omega_mean = om_answ[str((n, n_prime))] if str((n, n_prime)) in om_answ else 0.0

    print(f"reference omega  / omega_normalized_coeff = {reel_omega_mean:.5e}")

    print(f'-' * 75)


if __name__ == '__main__':
    problem_variable = {
        "theta": np.pi/4,  # [rad]
        "a": 2e-2,  # [m]
        "L": 50e-2,  # [m]
        "B": 1.4580,  # [?]
        "C": 0.00354e-12,  # [m^2]
        "T": 1_000,  # [K]
    }
    couples = [(3, 2), (4, 2), (5, 2), (6, 2)]

    start_time = time.time()

    # Problem 2.a
    print("\n Problem 2.a \n")
    for transition in couples:
        tab_cell(transition[0], transition[1])

    # Problem 2.b
    print("\n Problem 2.b \n")
    omega_c = QuantumFactory.get_transition_angular_frequency_unperturbed(4, 2, const.Z_H, const.mu_H)
    geometric_engine = Geometric_calculus(omega_c,
                                          problem_variable["theta"], problem_variable["a"], problem_variable["L"],
                                          problem_variable["B"], problem_variable["C"])

    for transition in couples:
        relative_intensity = QuantumFactory.relative_intensity_of_the_beam(transition[0], transition[1],
                                                                           4, 2, T=problem_variable["T"])
        print(f"---")
        print(f" I_{transition} / I_(4, 2) = {relative_intensity:.2e} [-]")
        omega = QuantumFactory.get_transition_angular_frequency_unperturbed(transition[0], transition[1], const.Z_H,
const.mu_H)
        print(f"x position {transition}: {geometric_engine.get_delta_x(omega):.2e} [m]")
        print(f"---")

    # Problem 2.c
    print("\n Problem 2.c \n")
    # Balmer series (n' = 2)
    # refrence: https://fr.wikipedia.org/wiki/Spectre_de_l%27atome_d%27hydrog%C3%A8ne?fbclid=
IwAR1Yy7gA_6GgFECMvbQgK51SCjNKGy3UQS7OM1EWvxUKtHODyz-0LM_Azjk
    Balmer_series = [(3, 2), (4, 2), (5, 2), (6, 2)]

    for transition in Balmer_series:
        relative_intensity = QuantumFactory.relative_intensity_of_the_beam(transition[0], transition[1],
                                                                           4, 2, T=problem_variable["T"])
        print(f"---")
        print(f" I_{transition} / I_(4, 2) = {relative_intensity:.2e} [-]")
        omega = QuantumFactory.get_transition_angular_frequency_unperturbed(transition[0], transition[1], const.Z_H,
const.mu_H)
        print(f"x position {transition}: {geometric_engine.get_delta_x(omega):.2e} [m]")
```

File - main.py

```
 96         print(f"---")
 97
 98     print(f"--- elapse time : {time.time() - start_time:.2f} s ---")
 99
100
```