

## File - Transitions.py

```

1 from Transition import Transition
2 import Constants as const
3 import numpy as np
4 import sympy as sp
5 import tqdm
6
7
8 class Transitions(list):
9     def __init__(self, n: int = None, n_prime: int = None, hydrogen: bool = False):
10         """
11             Transitions constructor. Transitions is a container of Transition object
12             that can calculate some stats on these transitions
13             :param n: initial orbital number n (int)
14             :param n_prime: final orbital number n (int)
15             :param hydrogen : if the current container of quantum states refer to hydrogen atoms (bool)
16         """
17         super().__init__()
18         assert (n is None and n_prime is None) or (isinstance(n, int) and isinstance(n_prime, int)), \
19             "params n and n_prime must be integer or None"
20         if isinstance(n, int) and isinstance(n_prime, int):
21             self.append_transitions_n_to_n(n, n_prime, hydrogen=hydrogen)
22             self.spontanious_decay_mean: float = None
23
24     def append(self, transition: Transition) -> None:
25         """
26             Add a transition in the current container
27             :param transition: Transition to be added (Transition)
28             :return: None
29         """
30         super().append(transition)
31
32     def append_transitions_n_to_n(self, n, n_prime, hydrogen: bool = False) -> None:
33         """
34             Add all the possible transition between orbital number n and n_prime
35             :param n: initial orbital number n (int)
36             :param n_prime: final orbital number n (int)
37             :param hydrogen: if we want to cast quantum state in quantum hydrogen state (bool)
38             :return: None
39         """
40         from QuantumFactory import QuantumFactory
41         for trans in QuantumFactory.get_valid_transitions_n_to_n_prime(n, n_prime, hydrogen=hydrogen):
42             self.append(trans)
43
44     def get_spontanious_decay_mean(self, z=const.Z_H, mu=const.mu_H, verbose: bool = False) -> float:
45         """
46             Get mean of the spontanious decay rate of transitions in the current container
47             :param z: atomic number (float)
48             :param mu: reduced mass (float)
49             :param verbose: if True, will show elapse time (bool)
50             :return: mean spontanious decay rate (float)
51         """
52         if self.spontanious_decay_mean is not None:
53             return self.spontanious_decay_mean
54
55         if verbose:
56             rs_vector: np.ndarray = np.array([trans.get_spontanious_decay_rate(z=z, mu=mu) for trans in tqdm.tqdm(self
57             )])
58         else:
59             rs_vector: np.ndarray = np.array([trans.get_spontanious_decay_rate(z=z, mu=mu) for trans in self])
60
61         self.spontanious_decay_mean: float = np.float(np.mean(rs_vector))
62         return self.spontanious_decay_mean
63
64     @staticmethod
65     def get_angular_frequency(n, n_prime, z=sp.Symbol("Z", real=True), mu=sp.Symbol('mu', real=True)) -> float:
66         """
67             Get the angular frequency between two states using the unperturbeted energy.
68             :param n: initial orbital number n (int)
69             :param n_prime: final orbital number n (int)
70             :param z: atomic number (float)
71             :param mu: reduced mass (float)
72             :return: angular frequency (float)
73         """
74         import warnings
75         warnings.warn("Warning! This method seems to be replicated and not efficient", DeprecationWarning)
76         e = (- (z ** 2) * (const.alpha ** 2) * mu * (const.c ** 2))/(2 * (n ** 2))
77         e_prime = (- (z ** 2) * (const.alpha ** 2) * mu * (const.c ** 2))/(2 * (n_prime ** 2))
78         return (e - e_prime)/const.hbar
79
80     def __repr__(self) -> str:
81         """
82             String representation of the current object
83             :return: string representation (str)
84         """
85         this_repr = "[ "
86         for trans in self:
87             this_repr += f"{trans}, \n"
88         this_repr += "]"
89         return this_repr
90
91     def get_n_to_n_prime_couple(self) -> set:
92         """
93             gives a set of every tuple (n, n_prime) of the current container
94             :return: tuple (n, n_prime) (set)
95         """
96         couple_set: set = set()
97         for trans in self:
98             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_unperturbated(couple[0], couple[1], z, mu)
106            N = Quantumfactory.decay_number(couple[0], T, z, mu)
107            Rs_mean = self.get_spontanious_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 - Constants.py

```
1 import numpy
2 from scipy import constants
3
4 mu0: float = 1.2566e-6 # [kg m A^-2 s^-2]
5 c: float = 2.998e8 # [m/s]
6 alpha: float = 1/137.03599
7 hbar: float = 1.054571817e-34 # [j*s]
8 h: float = 6.602607015e-34 # [j*s]
9 Z_H: int = 1
10 q_e: float = 1.6021766208e19 # [As]
11 m_e: float = 9.109e-31 # [kg]
12 m_n: float = 1.6889e-27 # [kg]
13 m_p: float = 1.6725e-27 # [kg]
14 mu_B: float = (q_e * hbar)/(2 * m_e)
15 g_ell: float = 1.0
16 g_s: float = 2.0
17 s_H: numpy.ndarray = numpy.array([1/2]) # possible s for Hydrogen
18 k_B: float = constants.value("Stefan-Boltzmann constant") # [W m^-2 K^-4] Boltzmann constant
19 a0: float = constants.value("Bohr radius")
20
21
22 def mu_mag(L, S) -> float:
23     """
24         return the magnetic field with quantum number L and S
25         :param L: quantume number L (float)
26         :param S: quantum number S (S)
27         :return: (float)
28     """
29     return -(mu_B/hbar)*(g_ell*L + g_s*S)
30
31
32 def mu_mass(m_1, m_2) -> float:
33     """
34         return the reduced mass
35         :param m_1: mass 1 (float)
36         :param m_2: mass 2 (float)
37         :return: reduced mass (float)
38     """
39     return (m_1*m_2)/(m_1+m_2)
40
41
42 mu_H = mu_mass(m_p, m_e) # reduced mass of hydrogen
43
44
45 if __name__ == '__main__':
46     print(f"alpha = {alpha}, \n"
47           f"hbar = {hbar}, \n"
48           f"c = {c}, \n"
49           f"a0 = {a0}, \n"
50           f"mu_H = {mu_H}")
```

## File - Transition.py

```

1 from QuantumState import QuantumState
2 import Constants as const
3 import sympy as sp
4 import numpy as np
5 import numba
6 from QuantumFactory import QuantumFactory
7 import os
8
9
10 class Transition:
11     TRANSITIONS_RULES: dict = {"Delta ell": [-1, 1],
12                                 "Delta m_ell": [-1, 0, 1],
13                                 "Delta s": [0],
14                                 "Delta m_s": [0]}
15
16     TRANSITIONS_RULES_SPIN: dict = {"Delta ell": [-1, 1],
17                                      "Delta m_ell": [-1, 0, 1],
18                                      "Delta j": [-1, 0, 1],
19                                      "Delta m_j": [-1, 0, 1]}
20
21     n_ell_m_ell_state_to_rs: dict = dict()
22
23     transition_count: int = int(0)
24
25     static_save_file = os.getcwd() + "/Transition_static_values.npy"
26
27     def __init__(self, initial_quantum_state: QuantumState, ending_quantum_state: QuantumState):
28         """
29             Transition constructor
30             :param initial_quantum_state: initial quantum state (QuantumState)
31             :param ending_quantum_state: final quantum state (QuantumState)
32         """
33         self._initial_quantum_state = initial_quantum_state
34         self._ending_quantum_state = ending_quantum_state
35         self.check_invariant()
36         self.spontaneous_decay_rate = None
37         Transition.transition_count += 1
38         self.load_static_values()
39
40     def __del__(self):
41         Transition.save_static_values()
42         Transition.transition_count -= 1
43
44     @staticmethod
45     def save_static_values() -> None:
46         if Transition.transition_count >= 1:
47             np.save(Transition.static_save_file, Transition.n_ell_m_ell_state_to_rs)
48
49     @staticmethod
50     def load_static_values() -> None:
51         if Transition.transition_count >= 1 and os.path.exists(Transition.static_save_file):
52             Transition.n_ell_m_ell_state_to_rs = np.load(Transition.static_save_file, allow_pickle=True).item()
53
54     @staticmethod
55     def clear_static_values() -> None:
56         os.remove(Transition.static_save_file)
57
58     def check_invariant(self) -> None:
59         """
60             Check every invariant for a Transition
61             :return: None
62         """
63         assert Transition.possible(self._initial_quantum_state, self._ending_quantum_state)
64
65     def get_n_to_n_prime_couple(self) -> tuple:
66
67         Gives the tuple (initial_quantum_state.n, ending_quantum_state.n)
68         :return: (tuple)
69
70         return self._initial_quantum_state.get_n(), self._ending_quantum_state.get_n()
71
72     def __repr__(self) -> str:
73
74         show a representation of the current Transition
75         :return: string representation of self (str)
76
77         this_repr = f"{{self._initial_quantum_state}} -> {{self._ending_quantum_state}}"
78         return this_repr
79
80     def repr_without_spin(self) -> str:
81
82         show a representation of the current Transition without s and m_s
83         :return: string representation of self without orbital spin (str)
84
85         this_repr = f"{{self._initial_quantum_state.repr_without_spin()}} " \
86                     f"-> {{self._ending_quantum_state.repr_without_spin()}}"
87         return this_repr
88
89     def get_spontaneous_decay_rate(self, z=sp.Symbol('Z', real=True), mu=sp.Symbol('mu', real=True), algo="auto"):
90
91         Get the spontaneous decay rate of the transition
92         :param z: atomic number (int)
93         :param mu: reduced mass (float)
94         :return: the spontaneous decay rate (float)
95
96         # check if spontaneous_decay_rate is already calculated
97         if self.spontaneous_decay_rate is not None:
98             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.spontanious_decay_rate = Transition.n_ell_m_ell_state_to_rs[self.repr_without_spin()]
101         return self.spontanious_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_fonction(z, mu)
107     psi_prime = self._ending_quantum_state.get_wave_fonction(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.cos(theta) * (sp.cos(phi) + sp.sin(phi) + 1)
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.spontanious_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.spontanious_decay_rate
128     return self.spontanious_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_fonction()}")
188     print(f"psi_2{qs2} = {qs2.get_wave_fonction()}")
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_spontanious_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 - QuantumFactory.py

```

1 import itertools
2
3 import mpmath
4 import numpy as np
5 import scipy as sc
6 import sympy as sp
7 from numba import prange
8 from scipy import integrate
9
10 import Constants as const
11
12
13 class QuantumFactory:
14     """
15         QuantumFactory is a module to combine a bunch of static method util in quantum mechanics
16     """
17
18     @staticmethod
19     def zeta_n(n=sp.Symbol("n", real=True), r=sp.Symbol("r", real=True, positive=True),
20               z=sp.Symbol("z", real=True), mu=sp.Symbol('mu', real=True)):
21         """
22             return the zeta_n function
23             :param n: orbital number n (int)
24             :param r: rayon variable (sympy object)
25             :param z: atomic number (int)
26             :param mu: reduced mass
27             :return: zeta_n function (sympy object)
28         """
29
30         return (2 * z * const.alpha * mu * const.c * r) / (n * const.hbar)
31
32     @staticmethod
33     def u_n_ell(n=sp.Symbol("n", real=True), ell=sp.Symbol("ell", real=True),
34                 z=sp.Symbol("z", real=True), mu=sp.Symbol('mu', real=True)):
35         """
36             return the u_{n \ell} function
37             :param n: orbital number n (int)
38             :param ell: kinetic momentum (int)
39             :param z: atomic number (int)
40             :param mu: reduced mass (float)
41             :return: (sympy object)
42         """
43
44         u_coeff_norm = sp.sqrt(((2 * z * const.alpha * mu * const.c) / (n * const.hbar)) ** 3)
45         u_coeff_fact = sp.sqrt(np.math.factorial(n - ell - 1) / (2 * n * np.math.factorial(n + ell)))
46         u_coeff = u_coeff_norm * u_coeff_fact
47
48         exp_term = sp.exp(-QuantumFactory.zeta_n(n=n, z=z, mu=mu) / 2)
49         laguerre_term = sp.assoc_laguerre(n - ell - 1, 2 * ell + 1, QuantumFactory.zeta_n(n=n, z=z, mu=mu))
50
51         return u_coeff * exp_term * (QuantumFactory.zeta_n(n=n, z=z, mu=mu) ** ell) * laguerre_term
52
53     @staticmethod
54     def Y_ell_m_ell(ell: int, m_ell: int):
55         """
56             Return the spherical harmonic function
57             :param ell: kinetic momentum (int)
58             :param m_ell: quantum number m_ell (int)
59             :return: (sympy object)
60
61             from sympy.functions.special.spherical_harmonics import Ynm
62             theta, phi = sp.Symbol("theta", real=True), sp.Symbol("phi", real=True)
63             # return sp.FU['TR8'](Ynm(ell, m_ell, theta, phi).expand(func=True))
64             # return Ynm(ell, m_ell, theta, phi)
65             return Ynm(ell, m_ell, theta, phi).expand(func=True)
66
67     @staticmethod
68     def get_hydrogen_wave_function(n: int, ell: int, m_ell: int):
69         """
70             Give the hydrogen wave function of sympy
71             :param n: quantum number n (int)
72             :param ell: quantum number ell (int)
73             :param m_ell: quantum number m_ell (int)
74             :return: hydrogen wave function (sympy object)
75
76             from sympy.physics import hydrogen
77             r, theta, phi = sp.Symbol("r", real=True, positive=True), \
78                           sp.Symbol("theta", real=True), sp.Symbol("phi", real=True)
79             return hydrogen.Psi_nl(n, ell, m_ell, r, phi, theta, Z=1 / const.a0)
80
81     @staticmethod
82     def get_valid_ell_with_n(n: int) -> np.ndarray:
83         """
84             Gives the possible ell with a given n
85             :param n: quantum number n (int)
86             :return: array of possible quantum number ell (np.ndarray)
87
88             return np.array([i for i in np.arange(start=0, stop=n, step=1)])
89
90     @staticmethod
91     def get_valid_m_ell_with_ell(ell: int) -> np.ndarray:
92         """
93             Gives the possible m_ell with a given ell
94             :param ell: quantum number ell (int)
95             :return: array of possible m_ell (np.ndarray)
96
97             return np.array([i for i in np.arange(start=-ell, stop=ell + 1, step=1)])
98
99     @staticmethod
100    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: (int)
111             :param n_prime: (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
123         return valid_transitions
124
125     @staticmethod
126     def get_valid_transitions_n_to_n_prime(n: int, n_prime: int, hydrogen: bool = False) -> list:
127         """
128             Get a Transitions(list) container of all of the valid transition of n to n_prime
129             :param n: initial quantum number n (int)
130             :param n_prime: final quantum number n (int)
131             :param hydrogen: if we want to cast quantum state in quantum hydrogen state (bool)
132             :return: Transitions(list) of Transition object of the initial state and final state (Transitions)
133         """
134         from Transition import Transition
135         from Transitions import Transitions
136         valid_transitions: Transitions = Transitions()
137         for init_quantum_state in QuantumFactory.get_valid_quantum_state_for_n(n, hydrogen=hydrogen):
138             for end_quantum_state in QuantumFactory.get_valid_quantum_state_for_n(n_prime, hydrogen=hydrogen):
139                 if Transition.possible(init_quantum_state, end_quantum_state):
140                     valid_transitions.append(Transition(init_quantum_state, end_quantum_state))
141
142         return valid_transitions
143
144     @staticmethod
145     def get_valid_quantum_state_for_n(n, s_array: np.ndarray = const.s_H, hydrogen: bool = False) -> np.ndarray:
146         """
147             Get all the valid quantum state for the orbital number n
148             :param n: orbital number n (int)
149             :param s_array: array of possible spin number s (numpy.ndarray)
150             :param hydrogen: if we want to cast quantum state in quantum hydrogen state (bool)
151             :return: array of valid quantum state (numpy.ndarray[QuantumState])
152         """
153         from QuantumState import QuantumState
154         valid_states: list = list()
155         for ell in QuantumFactory.get_valid_ell_with_n(n):
156             for m_ell in QuantumFactory.get_valid_m_ell_with_ell(ell):
157                 for s in s_array:
158                     for m_s in QuantumFactory.get_valid_m_s_with_s(s):
159                         valid_states.append(QuantumState(n, ell, m_ell, s, m_s, hydrogen))
160
161         return np.array(valid_states)
162
163     @staticmethod
164     def get_g_n(n: int) -> int:
165         """
166             Get the degenesrence number of the level n
167             :param n: orbital level (int)
168             :return: degenescence number (int)
169         """
170         deg_dico = dict()
171         for quantum_state in QuantumFactory.get_valid_quantum_state_for_n(n):
172             energy = quantum_state.get_state_energy()
173             if energy in deg_dico.keys():
174                 deg_dico[energy] += 1
175             else:
176                 deg_dico[energy] = 1
177
178         return int(np.sum(list(deg_dico.values())))
179
180     @staticmethod
181     def get_state_energy_unperturbed(n: int, z=sp.Symbol("Z", real=True), mu=sp.Symbol('mu', real=True)):
182         """
183             Get the energy of the current quantum state
184             :param n: quantum number n (int)
185             :param z: atomic number (float)
186             :param mu: reduced mass (float)
187             :return: the energy of the current quantum state (float if z and mu are float else sympy object)
188         """
189         numerator = - (z ** 2) * (const.alpha ** 2) * mu * (const.c ** 2)
190         denominator = 2 * (n ** 2)
191         return numerator / denominator
192
193     @staticmethod
194     def get_delta_energy_unpertubered(n: int, n_prime: int,
195                                         z=sp.Symbol('Z', real=True), mu=sp.Symbol('mu', real=True)):
196         """
197             Getter of the transition energy without any perturbation
198             :param n: initial orbital number n (int)
199             :param n_prime: final orbital number n (int)
200             :param z: atomic number (float)

```

File - QuantumFactory.py

```

197     :param mu: reduced mass (float)
198     :return: transition energy (float) or transition energy (sympy object)
199     """
200     e = QuantumFactory.get_state_energy_unperturbed(n, z, mu)
201     e_prime = QuantumFactory.get_state_energy_unperturbed(n_prime, z, mu)
202     return e - e_prime
203
204     @staticmethod
205     def get_transition_angular_frequency_unperturbated(n: int, n_prime: int,
206                                                       z=sp.Symbol('Z', real=True), mu=sp.Symbol('mu', real=True)):
207         """
208             Getter of the transition angular frequency without any perturbation
209             :param n: initial orbital number n (int)
210             :param n_prime: final orbital number n (int)
211             :param z: atomic number (float)
212             :param mu: reduced mass (float)
213             :return: angular frequency (float) or angular frequency (sympy object)
214         """
215         return QuantumFactory.get_delta_energy_unpertubered(n, n_prime, z, mu) / const.hbar
216
217     @staticmethod
218     def decay_number(n: int, T: float, z: int = const.Z_H, mu: float = const.mu_H):
219         """
220             Return the decay number of the level n for unperturbeted energy.
221             :param n: orbital number n (int)
222             :param T: Current temperature (float)
223             :param z: atomic number (int)
224             :param mu: reduced mass (float)
225             :return: a sympy expression of the decay number (sympy object)
226         """
227         g = QuantumFactory.get_g_n(n)
228         return g * sp.exp(-QuantumFactory.get_state_energy_unperturbeted(n, z, mu) / (const.k_B * T))
229
230     @staticmethod
231     def decay_number_ratio(n: int, n_prime: int, T: float, z: int = const.Z_H, mu: float = const.mu_H):
232         """
233             Return the ratio of decay number of the levels n to n_prime for unperturbeted energy.
234             :param n: initial orbital number n (int)
235             :param n_prime: final orbital number n (int)
236             :param T: Current temperature (float)
237             :param z: atomic number (int)
238             :param mu: reduced mass (float)
239             :return: the ration of decay number (float)
240         """
241         N_i = QuantumFactory.decay_number(n, T, z, mu)
242         N_j = QuantumFactory.decay_number(n_prime, T, z, mu)
243         return (N_i / N_j).evalf()
244
245     @staticmethod
246     def intensity_of_the_beam(n: int, n_prime: int, T: float, z: int = const.Z_H, mu: float = const.mu_H):
247         """
248             Gives the intensity of the beam with a proportional function alpha
249             :param n: initial orbital number n (int)
250             :param n_prime: final orbital number n (int)
251             :param T: Current temperature (float)
252             :param z: atomic number (int)
253             :param mu: reduced mass (float)
254             :return: intensity of the beam (sympy object)
255         """
256         from Transitions import Transitions
257         transitions = Transitions(n, n_prime)
258         alpha = sp.Symbol('alpha') # proportional function
259         omega = QuantumFactory.get_transition_angular_frequency_unperturbated(n, n_prime, z, mu)
260         N = QuantumFactory.decay_number(n, T, z, mu)
261         Rs_mean = transitions.get_spontanious_decay_mean(z, mu)
262         return alpha * N * omega * Rs_mean
263
264     @staticmethod
265     def relative_intensity_of_the_beam(n: int, n_prime: int, i: int, j: int,
266                                      T: float, z: int = const.Z_H, mu: float = const.mu_H):
267         """
268             Gives the intensity of the beam normalized with the intensity of another transition beam
269             :param n: initial orbital number n (int)
270             :param n_prime: final orbital number n (int)
271             :param i: initial orbital number n of the base transition (int)
272             :param j: final orbital number n of the base transition (int)
273             :param T: Current temperature (float)
274             :param z: atomic number (int)
275             :param mu: reduced mass (float)
276             :return: intensity of the beam normalized (float)
277         """
278         I1 = QuantumFactory.intensity_of_the_beam(n, n_prime, T, z, mu)
279         I2 = QuantumFactory.intensity_of_the_beam(i, j, T, z, mu)
280         return (I1 / I2).evalf()
281
282     @staticmethod
283     def bracket_product(wave_function, wave_function_prime, operator=None, algo="mcint"):
284         """
285             Call the algo function to make the scalar product <bra/operator|ket>
286             :param wave_function: bra
287             :param wave_function_prime: ket
288             :param operator: operator
289             :param algo: algo to use to compute the integral (str) element of
290                         {sympy, sympy_ns, scipy_nquad, scipy_tplquad, mcint}
291             :return: the result of algo(wave_function, wave_function_prime, operator)
292         """
293         algos = {"sympy": QuantumFactory.bracket_product_sympy,
294                  "sympy_ns": QuantumFactory.bracket_product_sympy_ns,

```

```

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

```

## File - QuantumFactory.py

```

393     integral_core_expansion = sp.expand(jacobian * integral_core, func=True).simplify()
394
395     # Process the integral with scipy
396     integral_core_lambdify = sp.lambdify((theta, r, phi), integral_core_expansion, modules="numpy")
397     print(lambdastr((theta, r, phi), integral_core_expansion))
398     bracket_product = QuantumFactory.complex_quadrature(sc.integrate.tplquad, integral_core_lambdify,
399                                         0, 2 * np.pi, lambda b_phi: 0, lambda b_phi: np.inf,
400                                         lambda b_theta, b_r: 0, lambda b_theta, b_r: np.pi)[0]
401     # bracket_product = sc.integrate.tplquad(integral_core_lambdify,
402     #                                         0, 2 * np.pi, lambda b_phi: 0, lambda b_phi: np.inf,
403     #                                         lambda b_theta, b_r: 0, lambda b_theta, b_r: np.pi)[0]
404
405     return bracket_product
406
407 @staticmethod
408 def complex_quadrature(integrate_func, func, *args, **kwargs):
409     """
410         Gives the complex quadrature of a function
411         :param integrate_func: the integration function
412         :param func: the actual function to integrate
413         :param args: args to pass to integrate_func
414         :param kwargs: kwargs to pass to integrate_func
415         :return: (the complex quadrature, real error, imag error) (tuple)
416     """
417
418     def real_func(x, y, z):
419         return sc.real(func(x, y, z))
420
421     def imag_func(x, y, z):
422         return sc.imag(func(x, y, z))
423
424     real_integral = integrate_func(real_func, *args, **kwargs)
425     imag_integral = integrate_func(imag_func, *args, **kwargs)
426     return real_integral[0] + 1j * imag_integral[0], real_integral[1:], imag_integral[1:]
427
428 @staticmethod
429 def bracket_product_mcint(wave_function, wave_function_prime, operator=None):
430     """
431         Compute the scalar product <bra/operator|ket> with a monte carlo integration algorithm
432         :param wave_function: bra
433         :param wave_function_prime: ket
434         :param operator: operator
435         :return: bracket product (float)
436     """
437     r, theta, phi = sp.Symbol("r", real=True), sp.Symbol("theta", real=True), sp.Symbol("phi", real=True)
438     jacobian = (r ** 2) * sp.sin(theta)
439     integral_core = sp.conjugate(wave_function) * (operator if operator is not None else 1) * wave_function_prime
440
441     # integral_core_expansion = sp.expand(sp.FU["TR8"])(jacobian * integral_core), func=True).simplify()
442     integral_core_expansion = sp.expand(jacobian * integral_core, func=True).simplify()
443
444     integral_core_lambdify = sp.lambdify((theta, r, phi), integral_core_expansion, modules="numpy")
445
446     def integrand(x):
447         theta, r, phi = x[0], x[1], x[2]
448         return integral_core_lambdify(theta, r, phi)
449
450     domainsize = (0, 1)
451     result = QuantumFactory.monte_carlo_integration_sph(integrand, [[0, 1.15], [0, np.pi], [0, 2 * np.pi]],
452                                                       domainsize, 1_000_000)
453
454     return result
455
456 @staticmethod
457 def monte_carlo_integration(integrand, bornes: list, domainsize=(0.0, 1.0), n_sample: int = int(1e6)):
458     """
459         Compute the integral with a monte carlo integration algorithm
460         :param integrand: the function to integrate (lambda or func)
461         :param bornes: the
462         :param domainsize: (tuple of len==2)
463         :param n_sample: number of sample to use (int)
464         :return: the integral of the integrand (float)
465     """
466     np.random.seed(1)
467     # Sum elements and elements squared
468     total = 0.0
469     total_sq = 0.0
470     count_in_curve = 0
471
472     def sampler():
473         while True:
474             yield [np.random.uniform(b0, b1) for [b0, b1] in bornes]
475
476     for x in itertools.islice(sampler(), n_sample):
477         f = integrand(x)
478         f_rn = np.random.uniform(domainsize[0], domainsize[1])
479         total += f
480         total_sq += (f ** 2)
481         count_in_curve += 1 if 0 <= f_rn <= f else 0
482
483     # Return answer
484     # sample_mean = total / n_sample
485     # sample_var = (total_sq - ((total / n_sample) ** 2) / n_sample) / (n_sample - 1.0)
486     # return domainsize * sample_mean, domainsize * np.sqrt(sample_var / n_sample)
487     v = np.prod([b1 - b0 for [b0, b1] in bornes]) * (domainsize[1] - domainsize[0])
488     print(100 * (count_in_curve / n_sample), "%")
489     return (count_in_curve / n_sample) * v
490
491 @staticmethod
492 def monte_carlo_integration_sph(integrand, bornes: list, domainsize=(0.0, 1.0), n_sample: int = int(1e6)):
493     """

```

File - QuantumFactory.py

```

491     Compute the integral with a spherical monte carlo integration algorithm
492     :param integrand: the function to integrate (lambda or func)
493     :param bornes: the
494     :param domainsize: (tuple of len==2)
495     :param n_sample: number of sample to use (int)
496     :return: the integral of the integrand (float)
497     """
498     # np.random.seed(1)
499     # Sum elements and elements squared
500
501     # def sampler():
502     #     while True:
503     #         yield [np.random.uniform(b0, b1) for [b0, b1] in bornes]
504
505     samples = np.array([[np.random.uniform(b0, b1) for [b0, b1] in bornes] for _ in range(n_sample)])
506
507     # f_samples = np.array([np.random.uniform(domainsize[0], domainsize[1]) for _ in range(n_sample)])
508     # for x in itertools.islice(sampler(), n_sample):
509     #     f_values = integrand(samples)
510     #     @numba.generated_jit(nopython=True)
511     def loop():
512         count: int = int(0)
513         for i in prange(len(samples)):
514             f = integrand(samples[i])
515             # f = f_values[i]
516             f_rn = np.random.uniform(domainsize[0], domainsize[1])
517             # if (f > 0 and 0 <= f_samples[i] <= f) or (f < 0 and f <= f_samples[i] <= 0):
518             if (f > 0 and 0 <= f_rn <= f) or (f < 0 and f <= f_rn <= 0):
519                 count += 1
520         return count
521
522     count_in_curve = loop()
523     # Return answer
524     # sample_mean = total / n_sample
525     # sample_var = (total_sq - ((total / n_sample) ** 2) / n_sample) / (n_sample - 1.0)
526     # return domainsize * sample_mean, domainsize * np.sqrt(sample_var / n_sample)
527     # v = ((4*np.pi*(bornes[0][1] - bornes[0][0])**3)/3) * (domainsize[1] - domainsize[0])
528     # print(sp.integrate((4 / 3) * sp.pi * sp.Symbol('r')**3, (sp.Symbol('z'), domainsize[0], domainsize[1])))
529     v = sp.integrate((4 / 3) * sp.pi * sp.Symbol('r') ** 3, (sp.Symbol('z'), domainsize[0], domainsize[1])).evalf(
530     subs={'r': bornes[0][1] - bornes[0][0]})
531     # print(100 * (count_in_curve / n_sample), "%")
532     return (count_in_curve / n_sample) * v
533
534
535 if __name__ == '__main__':
536     from Transitions import Transitions
537     from QuantumState import QuantumState
538
539     qs1 = QuantumState(n=2, ell=1, m_ell=1, s=0.5, m_s=0.5)
540     qs2 = QuantumState(n=1, ell=0, m_ell=0, s=0.5, m_s=0.5)
541
542     y1 = QuantumFactory.Y_ell_m_ell(3, 2)
543     y2 = QuantumFactory.Y_ell_m_ell(2, 1)
544
545     print(QuantumFactory.bracket_product(y1, y1))
546     print(QuantumFactory.bracket_product(y1, y2))
547     print(QuantumFactory.bracket_product(y2, y2))
548

```

## File - QuantumState.py

```

1 import numpy as np
2 import Constants as const
3 import sympy as sp
4 from QuantumFactory import QuantumFactory
5
6
7 class QuantumState:
8
9     def __init__(self, n: int, ell: int, m_ell: int, s: float, m_s: float, hydrogen: bool = False):
10         """
11             QuantumState constructor
12             :param n: orbital number (int)
13             :param ell: angular momentum (int)
14             :param m_ell: quantum number m_ell (int)
15             :param s: spin (float)
16             :param m_s: quantum number m_s (float)
17             :param hydrogen : if the current quantum state refer to a hydrogen atom (bool)
18         """
19         self._n = n
20         self._ell = ell
21         self._m_ell = int(m_ell)
22         self._s = float(s)
23         self._m_s = float(m_s)
24         self.hydrogen = bool(hydrogen)
25         self.check_invariants()
26
27     def check_invariants(self) -> None:
28         """
29             Check every invariant for a quantum state
30             :return: None
31         """
32         assert self._n >= 1
33         assert 0 <= self._ell < self._n
34         assert -self._ell <= self._m_ell <= self._ell
35         assert self._s >= 0.0
36         assert -self._s <= self._m_s <= self._s
37
38     def getState(self) -> np.ndarray:
39         """
40             Getter of the state vector of state numbers in order (n, ell, m_ell, s, m_s)
41             :return: [n, ell, m_ell, s, m_s] (numpy.ndarray)
42         """
43         return np.array([self._n, self._ell, self._m_ell, self._s, self._m_s])
44
45     def get_n(self) -> int:
46         return self._n
47
48     def get_ell(self) -> int:
49         return self._ell
50
51     def get_m_ell(self) -> int:
52         return self._m_ell
53
54     def get_s(self) -> float:
55         return self._s
56
57     def get_m_s(self) -> float:
58         return self._m_s
59
60     def get_state_energy(self, z=sp.Symbol("z", real=True), mu=sp.Symbol('mu', real=True)):
61         """
62             Get the energy of the current quantum state
63             :param z: atomic number
64             :param mu: reduced mass
65             :return: the energy of the current quantum state (float if z and mu are float else sympy object)
66         """
67         return QuantumFactory.get_state_energy_unperturbed(self._n, z, mu)
68
69     def get_valid_transitions_state_to_n(self, other_n: int) -> list:
70         """
71             Get all of the valid transition of the current quantum state to another in the orbital other_n
72             :param other_n: (int)
73             :return: list of QuantumState
74         """
75         import warnings
76         warnings.warn("Warning! This method seems to be not valid", DeprecationWarning)
77         from Transition import Transition
78         valid_transitions = list()
79         next_states = set()
80         for key, possibilities in Transition.TRANSITIONS_RULES.items():
81             for num in possibilities:
82                 for key_prime, possibilities_prime in Transition.TRANSITIONS_RULES.items():
83                     if key == key_prime:
84                         continue
85                     for num_prime in possibilities_prime:
86                         (_ , other_ell, other_m_ell, other_s, other_m_s) = tuple(self.getState())
87                         if key == "Delta ell":
88                             other_ell += num
89                         elif key == "Delta m_ell":
90                             other_m_ell += num
91                         elif key == "Delta s":
92                             other_s += num
93                         elif key == "Delta m_s":
94                             other_m_s += num
95
96                         if key_prime == "Delta ell":
97                             other_ell += num_prime
98                         elif key_prime == "Delta m_ell":

```

```

99             other_m_ell += num_prime
100            elif key_prime == "Delta s":
101                other_s += num_prime
102            elif key_prime == "Delta m_s":
103                other_m_s += num_prime
104        try:
105            next_state = QuantumState(other_n, other_ell, other_m_ell, other_s, other_m_s)
106        except AssertionError:
107            continue
108        if Transition.possible(self, next_state):
109            if str(next_state) not in next_states:
110                valid_transitions.append(next_state)
111                next_states.add(str(next_state))
112            else:
113                continue
114
115    return valid_transitions
116
117 def __repr__(self) -> str:
118     """
119     show a representation of the current quantum state
120     :return: string representation of self (str)
121     """
122     this_repr = f"(n: {self._n}, " \
123                 f"ell: {self._ell}, " \
124                 f"m_ell: {self._m_ell}, " \
125                 f"s: {self._s}, " \
126                 f"m_s: {self._m_s})"
127     return this_repr
128
129 def repr_without_spin(self) -> str:
130     """
131     show a representation of the current quantum state without s and m_s
132     :return: string representation of self without orbital spin (str)
133     """
134     this_repr = f"(n: {self._n}, " \
135                 f"ell: {self._ell}, " \
136                 f"m_ell: {self._m_ell})"
137     return this_repr
138
139 def get_wave_fonction(self, z=sp.Symbol("z", real=True), mu=sp.Symbol('mu', real=True)):
140     """
141     Get the wave function of the current quantum state as a sympy object
142     :param z: atomic number
143     :param mu: reduced mass (float)
144     :return: the wave function (sympy object)
145     """
146     if self.hydrogen:
147         return QuantumFactory.get_hydrogen_wave_function(self._n, self._ell, self._m_ell)
148
149     u = QuantumFactory.u_n_ell(n=self._n, ell=self._ell, z=z, mu=mu)
150     y = QuantumFactory.Y_ell_m_ell(self._ell, self._m_ell)
151     return u*y
152
153 def decay_number(self, T: float, z: int = const.Z_H, mu: float = const.mu_H):
154     """
155     Return the decay number of the current QuantumState.
156     :param T: Current temperature (float)
157     :param z: atomic number (int)
158     :param mu: reduced mass (float)
159     :return: a sympy expression of the decay number (sympy object)
160     """
161     alpha = sp.Symbol('alpha') # proportional function
162     return alpha*QuantumFactory.get_g_n(self._n)*sp.exp(-self.get_state_energy(z, mu)/(const.k_B*T))
163
164
165 if __name__ == '__main__':
166     from Transition import Transition
167     quantum_state = QuantumState(n=1, ell=0, m_ell=0, s=1 / 2, m_s=1 / 2)
168     print(f"\npsi_{quantum_state} = {quantum_state.get_wave_fonction()}")
169     print(f"E_{quantum_state} = {quantum_state.get_state_energy()}")
170     print(Transition.possible(quantum_state, QuantumState(2, 0, 0, 1 / 2, 1 / 2)))
171     print(Transition.possible(quantum_state, QuantumState(2, 1, 0, 1/2, 1 / 2)))
172
173     valid_transitions_test_1_to_2 = [QuantumState(2, 1, -1, 1/2, 1/2),
174                                     QuantumState(2, 1, 0, 1/2, 1/2),
175                                     QuantumState(2, 1, 1, 1/2, 1/2)]
176
177     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):
178         print(state.getState())
179
180     print('*'*175)
181
182     valid_transitions_test_3_to_1 = [QuantumState(1, 0, 0, 1/2, 1/2)]
183
184     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):
185         print(state.getState())
186
187     print('*'*175)
188
189     print(QuantumFactory.get_valid_ell_with_n(3))
190
191     print('*' * 175)
192
193     for valid_transition in QuantumFactory.get_valid_transitions_n_to_n(1, 2):
194         print(valid_transition)

```

File - geometric\_calculus.py

```

1 import numpy as np
2
3 import Constants as const
4
5
6 class Geometric_calculus:
7     """
8         This class wraps up methods that calculate multiple geometric
9             information such as angles and distances
10        """
11
12     def __init__(self, w_c, angle, a, L, B, C) -> None:
13         """
14             Constructor
15
16             Parameters
17             -----
18             w_c : float
19                 Calibrating frequency in Hz
20             angle : float
21                 Angle of the beam producer in rad
22             a : float
23                 Length of a side of the prism in meters
24             L : float
25                 Distance between the center of the prism and the screen in meter
26             B : float
27                 First Cauchy law's constant
28             C : float
29                 Second Cauchy law's constant in meters^2
30
31             Attributes
32             -----
33             self.w_c: float
34                 Calibrating frequency in Hz of the system
35             self.angle: float
36                 Angle of the beam producer in rad
37             self.phi : float
38                 Angle of the prism
39             self.a: float
40                 Length of a side of the prism
41             self.L: float
42                 Distance between the center of the prism and the screen in meter
43             self.B : float
44                 First Cauchy law's constant
45             self.C : float
46                 Second Cauchy law's constant in meters^2
47             self._x_ref : float
48                 Position of the beam on the screen for the frequency w_c in meters
49
50             self._w_c = w_c
51             self._angle = angle
52             self._a = a
53             self._L = L
54             self._B = B
55             self._C = C
56             self._phi = self._get_phi(self._w_c)
57             self._x_ref = self.get_x_for_frequency(self._w_c)
58
59     def _get_phi(self, w_c) -> float:
60         """
61             This methods calculates the value of
62                 the angle of the rotation of the prism which
63                 ensures that the beam with a frequency of w_c
64                 is perpendicular to the screen
65
66             Parameters
67             -----
68             w_c : float
69                 Calibrating frequency in Hz
70             angle : float
71                 Angle of the beam producer in rad
72             B : float
73                 First Cauchy law's constant
74             C : float
75                 Second Cauchy law's constant in meters^2
76             Returns
77             -----
78             phi: float
79                 value of the angle of rotation of the prism in rad
80
81             n = self._get_refraction_value(w_c)
82             sin_argument = np.pi / 3 - np.arcsin((np.sin(self._angle + np.pi / 6) / n))
83             arcsin_argument = n * np.sin(sin_argument)
84             phi = np.arcsin(arcsin_argument) - np.pi / 6
85             return phi
86
87     def _get_refraction_value(self, w) -> float:
88         """
89             This methods calculates the value of
90                 the angle of the rotation of the prism which
91                 ensures that the beam with a frequency of w_c
92                 is perpendicular to the screen
93
94             Parameters
95             -----
96             w : float
97                 Calibrating frequency in frequency of the beam
98             Returns
99             -----
100            value: float
101                The value of the refraction index for the given frequency

```

## File - geometric\_calculus.py

```

99      """  
100     c = const.c  
101     value = self._B + self._C / (((2 * np.pi * c) / w) ** 2)  
102     return value  
103  
104 def get_x_for_frequency(self, w) -> float:  
105     """This methods calculates the value of  
106         the position of the beam on the screen  
107  
108     Parameters  
109     -----  
110     w : float  
111         frequency of the beam in Hz  
112  
113     Returns  
114     -----  
115     value: float  
116         The value of the position of the beam on the screen in meters  
117  
118     x = self.get_h_of_first_refraction_of_beam(w) - self.get_differences_of_h(w)  
119     return x  
120  
121 def get_h_of_first_refraction_of_beam(self, w) -> float:  
122     """This methods calculates the value of  
123         the position of the beam on the screen if  
124         it was refracted only once  
125  
126     Parameters  
127     -----  
128     w : float  
129         frequency of the beam in Hz  
130  
131     Returns  
132     -----  
133     value: float  
134         The value of the position of the beam on the screen in meters if the beam was refracted only once  
135  
136     n = self._get_refraction_value(w)  
137     arcsin_argument = np.sin(np.pi / 6 + self._angle) / n  
138     h_top = (self._L + (self._a / 3) * np.cos(self._phi)) * (br/>139         np.sin(np.arcsin(arcsin_argument) - np.pi / 6 + self._phi))  
140     return h_top  
141  
142 def get_differences_of_h(self, w) -> float:  
143     """This methods calculates the value of  
144         the difference of height between one and two diffraction  
145  
146     Parameters  
147     -----  
148     w : float  
149         frequency of the beam in Hz  
150  
151     Returns  
152     -----  
153     value: float  
154         the height differences in m  
155  
156     n = self._get_refraction_value(w)  
157     H = (self._L + (self._a / 3) * np.cos(self._phi)) / np.cos(  
158         np.arcsin((np.sin(np.pi / 6 + self._angle)) / n) - np.pi / 6 + self._phi)  
159     d = (np.sqrt(3) * (self._a / 3)) / np.sin(np.arcsin((np.sin(np.pi / 6 + self._angle)) / n) + np.pi / 6)  
160     den = np.sin(  
161         np.pi / 2 + np.abs(np.arcsin(np.sin(np.pi / 6 + self._angle) / n) - np.pi / 6 + self._phi) - np.abs(  
162             np.arcsin(np.sin(np.pi / 6 + self._angle) / n) - np.arcsin(  
163                 n * np.sin(np.pi / 3 - np.arcsin(np.sin(np.pi / 6 + self._angle) / n))))  
164     nim = np.sin(np.arcsin((np.sin(np.pi / 6 + self._angle)) / n) - np.arcsin(  
165         n * np.sin(np.pi / 3 - np.arcsin((np.sin(np.pi / 6 + self._angle)) / n))))  
166     diff_h = ((H - d) * nim) / den  
167     return diff_h  
168  
169 def get_delta_x(self, w) -> float:  
170     """This methods calculates the value of  
171         the distance between the beam and the beam of frequency of w_c  
172         on the screen  
173  
174     Parameters  
175     -----  
176     w : float  
177         frequency of the beam in Hz  
178  
179     Returns  
180     -----  
181     value: float  
182         The value of the position of the beam on the screen in meters  
183  
184     delta_x = self.get_x_for_frequency(w) - self._x_ref  
185     return delta_x  
186  
187  
188 if __name__ == "__main__":  
189     G = Geometric_calculus(3.5e15, np.pi / 4, 2e-2, 50e-2, 1.4580, 0.00354e-12)  
190     print(G.get_delta_x(2.0e15))  
191     print(G.get_delta_x(2.5e15))  
192     print(G.get_delta_x(3e15))  
193     print(G.get_delta_x(3.5e15))  
194     print(G.get_delta_x(4e15))  
195     print(G.get_delta_x(4.5e15))  
196     print(G.get_delta_x(5e15))

```

## File - main.py

```

1 from Transitions import Transitions
2 from QuantumFactory import QuantumFactory
3 import Constants as const
4 import numpy as np
5 import time
6
7
8 def tab_cell(n, n_prime):
9     rs_anw = {"(3, 2)": 0.00274668657777777777777777777778,
10                 "(4, 2)": 0.00052436,
11                 "(5, 2)": 0.000157596,
12                 "(6, 2)": 0.000060611}
13
14     om_anw = {"(3, 2)": 5/36,
15                 "(4, 2)": 3/16,
16                 "(5, 2)": 21/100,
17                 "(6, 2)": 2/9}
18
19     rs_mean_normalized_coeff = ((const.alpha ** 5) * const.mu_H * (const.c ** 2)) / const.hbar
20     omega_normalized_coeff = ((const.alpha ** 2) * const.mu_H * (const.c ** 2)) / (2 * const.hbar)
21
22     print(f'-'*50)
23
24     print(f"Transition ({n} -> {n_prime}) : \n")
25
26     transitions_n_to_n_prime = Transitions(n=n, n_prime=n_prime, hydrogen=True)
27     # print(f"Transitions: {transitions_n_to_n_prime} \n")
28
29     transitions_n_to_n_prime_rs_mean = transitions_n_to_n_prime.get_spontaneous_decay_mean() / rs_mean_normalized_coeff
30     print(f"R^s_mean / rs_mean_normalized_coeff = {transitions_n_to_n_prime_rs_mean:.5e}")
31
32     reel_rs_mean = rs_anw[str((n, n_prime))] if str((n, n_prime)) in rs_anw else 0.0
33
34     print(f"reel R^s / rs_mean_normalized_coeff = {reel_rs_mean:.5e} \n")
35
36     omega = QuantumFactory.get_transition_angular_frequency_unperturbed(n, n_prime, const.Z_H, const.mu_H)
37     omega_normalized = omega / omega_normalized_coeff
38
39     print(f"omega / omega_normalized_coeff = {omega_normalized:.5e}")
40
41     reel_omega_mean = om_anw[str((n, n_prime))] if str((n, n_prime)) in om_anw else 0.0
42
43     print(f"reel omega / omega_normalized_coeff = {reel_omega_mean:.5e}")
44
45     print(f'-' * 50)
46
47
48 if __name__ == '__main__':
49     problem_variable = {
50         "theta": np.pi/4,    # [rad]
51         "a": 2e-2,          # [m]
52         "L": 50e-2,          # [m]
53         "B": 1.4580,          # [?]
54         "C": 0.00354e-6,      # [m^2]
55         "T": 1_000,          # [K]
56     }
57     couples = [(3, 2), (4, 2), (5, 2), (6, 2)]
58
59     start_time = time.time()
60
61     # Problem 2.a
62     for couple in couples:
63         tab_cell(couple[0], couple[1])
64
65     # Problem 2.b
66     for couple in couples:
67         relative_intensity = QuantumFactory.relative_intensity_of_the_beam(couple[0], couple[1],
68                                         4, 2, T=problem_variable["T"])
69         print(f" I_{couple} / I_(4, 2) = {relative_intensity:.5f}")
70
71     print(f"--- elapse time : {time.time() - start_time:.2f} s ---")
72
73

```