MPhys Lab Book:

Quantum coherence of radical pairs in avian magnetoreception

Date: 21-27/09/2020

Author: Kieran Dominy (KAD212@exeter.ac.uk (mailto:KAD212@exeter.ac.uk))

In [1]:

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import odeint, solve_ivp
import multiprocessing as mp
from semi_classical import sphere_sampling_func
import timeit
import time
```

Initial setup

I spent some time getting a good workflow going for this project as I knew it would be important for doing efficent work on the project. This involved setting up:

- An Anaconda backend to ensure scientific python modules were up to date
- Setting up a GitHub Repository so that work can be efficently collaborated with other project members
- Pycharm as an IDE so that I can effectivly utilise the Git version control
- Using Jupyter notebook as a lab book so that python code which makes up a large portion of the project can be presented usefully
- Setting up Jupyter notebook extensions such as Latex_env (https://jupyter-contrib-nbextensions.readthedocs.io/en/latest/nbextensions/latex_envs/README.html) so that LaTeX citations and BibTex can be used in the lab book

Brief Introduction

It has been known for some time that night migratory birds such as the european robin must navigate using the earth's magnetic field. There is a considerable amount of evidence showing that night migratory birds react to magnetic cues[1] [2]. The birds must navigate cloudy starless nights during the first year of their life without having done the trip before. So the question is not “Can birds navigate using magnetoreception?” but “How do birds navigate using magnetoreception?”. This question still has no solid answer as the underlying mechanism has eluded discovery for some time now, however the current theory that radical pairs are the key to our avian friends compass sense has a sizable amount of evidence backing it [1] [2].

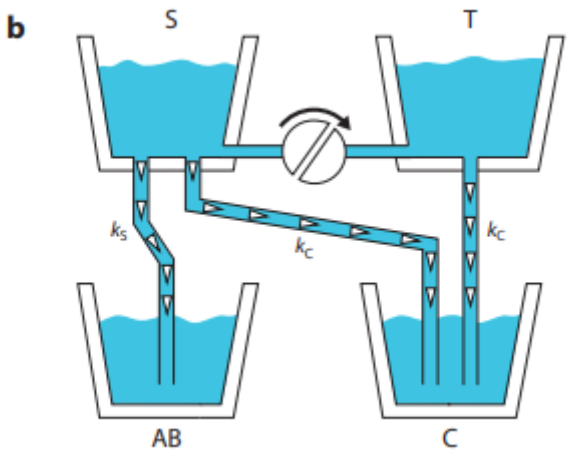
The aim of this project is to work out some bounds on the coherence lifetimes of radical pairs within the bird.

In order to understand the aim of this project one must first understand what exactly a radical pair is. A radical is a molecule with an unpaired outer electron [1]. Radicals pairs are pairs of radicals that are created simultaneously through a chemical reaction. Due to the quantum mechanical property of spin that electrons have, a magnetic field can affect the spin of the electrons and therefore the spin dynamics in a radical pair. The spins of the lone electrons in the radical pair are quantum mechanically linked through a property known as entanglement (also known as coherence) where the state of one electron will determine the state of the other electron irrespective of distance. An electron has a spin of a 1/2 and thus the pair of electrons in a radical pair can either be in a singlet state where the total spin angular momentum of the system is 0 or a triplet state where the total spin angular momentum of the system is one. The states are known as singlet and triplet as there is only one way to produce the singlet state but 3 ways to produce the triplet state using the bases of the states.

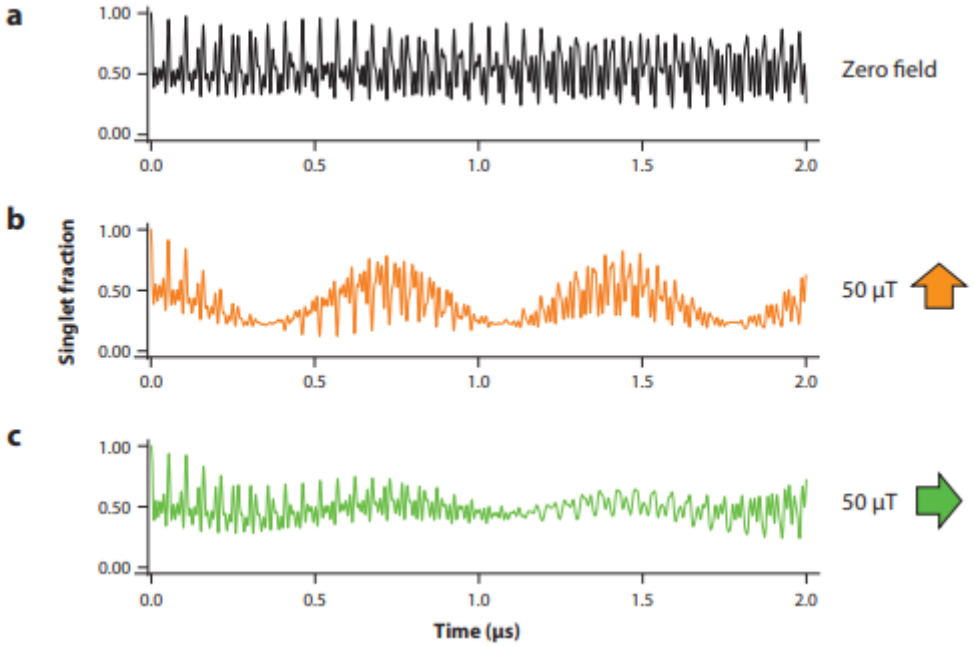
The electrons in the radical pair also interact with the other particles in the radical pair such as the protons and neutrons which have their own quantum mechanical spin. These interactions are known as hyperfine interactions [1]. The arrangement of particles in the radical pair is not isotropic so therefore the strength of these interactions will be anisotropic and depend on the geometry of the molecule. This is key to how the magnetic compass works and will be explained later as I must cover more background material first.

For a normal pair of entangled electrons the spin state they are formed in will remain there state but in a radical pair the hyperfine interactions caused by the spins of other particles interacting with the electrons cause the state of the electron pair to change between singlet and triplet states in a complicated process known as singlet-triplet interconversion.

The ways in which radical pairs can recombine into their parent molecule is the beginning of the process for how birds sense a magnetic field. Being in a singlet or a triplet state impacts the products produced by a radical pair recombination meaning that changing the fraction of radical pairs that are in one state or the other means the yields of their recombination products change. A helpful diagram (taken from Hore et al [1]) can be seen below which shows an analogy of the radical pair system with the top two buckets representing the singlet and triplet radical pairs. The pipe between them represents the singlet-triplet interconversion. The singlet states feed into both final products where as the triplet states feed into only product C. It is easy to see from this analogy how changing the rate at which singlet triplet interconversion happens can cause different amounts of either product to be produced.



The singlet triplet interconversion will cause the states of the electrons to oscillate. These oscillations are affected by magnetic fields as this changes the spin dynamics of the radical pair system. Due to the anisotropy of the hyperfine interactions the direction of the magnetic field can cause a significant impact on the fraction of states. An example of this is shown below in a graph taken from Hore et al [1]. The fact that the direction of the magnetic field can change the yields of the radical pair recombination products shows how biological pathways could be built to detect this change in products and therefore produce a magnetic compass sense.



Singlet triplet interconversion eventually ceases as over a large enough time scale as the entangled electrons reach an equilibrium with the surrounding spins causing them to be in a permanent singlet/triplet state until their recombination. This is in essence the main goal of the project by figuring out the limits and lifetimes of these radical pairs and their interactions with magnetic fields so that it can be better understood whether this is the true mechanism by which birds have magnetoreception.

Cryptochromes are a type of protein that are used in biological signaling in large amounts of animals and plants, they can also produce radical pairs which makes them the ideal candidate for the molecule responsible for the magnetoreception of birds [1].

Simulating the interaction of the spins in the radical pair introduces a computational problem with the fact that cryptochrome proteins can have thousands of nuclear spins and producing exact quantum mechanical solutions requires exponentially more computational time for increasing numbers of spins [3].

In order to simulate the spin dynamics of these large molecules a semiclassical approximation is required which captures most of the quantum behaviour to a fair accuracy whilst also only increasing linearly in computational time required [3].

For small numbers of spins the full quantum mechanical calculation can be used [3] but when there are large numbers of nuclear spins in the environment a different approach must be used. One such approach is outlined in the next section.

Semi-Classical Theory

This project seeks to find the limits of coherence of radical pairs used in the magnetorecpetion of birds as limits on how long quantum effects can dominate in this regime are placed by themodynamically warm and messy environment of a living organism. To start with semi-classical methods [3] will be used to simulate the spin dynamics of the radical pairs.

To start with I will attempt to reproduce the left hand column in Fig 2 from Manolopoulos et al [3]. The first step to this is solving the coupled differential equations for nuclear and electron spin vectors:

$$\frac{d}{dt}\vec{S}_i(t) = \left[\vec{\omega}_i + \sum_{k=1}^{N_i} a_{ik}\vec{I}_{ik}(t)\right] \times \vec{S}_i(t) \tag{1}$$

$$\frac{d}{dt}\vec{I}_{ik}(t) = a_{ik}\vec{S}_i(t) \times \vec{I}_{ik}(t) \tag{2}$$

Where i is an indice for the radical of which the electron belongs, $\vec{S}_i$ is the electron spin angular momentum vector, $\vec{I}_{ik}$ is the nuclear spin angular momentum vector for k-th nuclear spin, a_{ik} is the hyperfine coupling constant for the k-th nuclear spin and electron spin, N_i is the the total number of nuclear spins and $\vec{\omega}_i$ factors in the effect of an applied magnetic field.

I will attempt to solve these dynamamically coupled eqauations using scipy below:

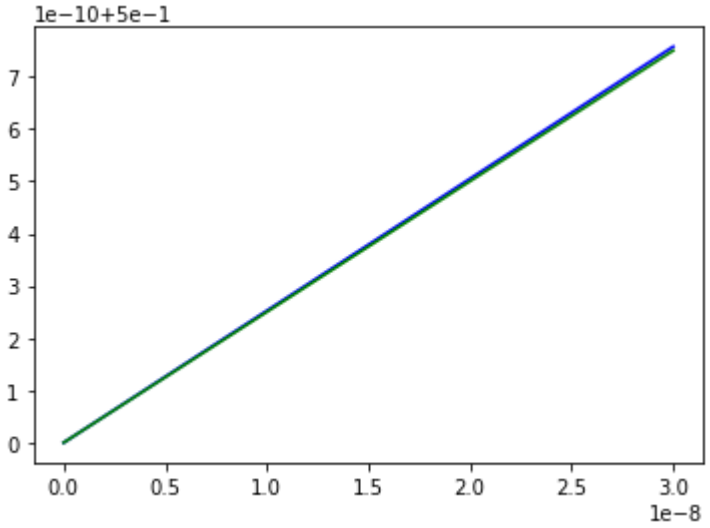
For my first attempt I tried to just get some code down so I could adapt it when I fully understood the equations mentioned above. I first attempted to solve the dynamically coupled differential equations.

In [2]:

```
from scipy.integrate import odeint

B = 0.5e-3
gamma_i = 1
omega = B * gamma_i
a = 0.1

def semi_classical(t, omega, a):

    def ddt(y,t):

        s,i = y
        dydt = [(omega + a*s)*i, a*s*i]

        return dydt

    y0 = [0.5, 0.5]

    sol = odeint(ddt, y0, t)

    return sol[:, 0], sol[:, 1]

t = np.linspace(0, 30e-9, 200)

s,i = semi_classical(t, omega, a)
plt.plot(t, s, 'b', t, i, 'g')
```

Out[2]:

```
[<matplotlib.lines.Line2D at 0x27eae2aae50>,
 <matplotlib.lines.Line2D at 0x27eae2aae80>]
```



Date: 28/09-04/10/2020

Aims

- This week I aim to fully reproduce the left hand column in Fig 2 from Manolopoulos et al [3].

Useful Recap

As discussed last week, 2 import equations in the semi-classical approach are:

$$\frac{d}{dt}\vec{S}_i(t) = \left[\vec{\omega}_i + \sum_{k=1}^{N_i} a_{ik}\vec{I}_{ik}(t)\right] \times \vec{S}_i(t) \tag{3}$$

$$\frac{d}{dt}\vec{I}_{ik}(t) = a_{ik}\vec{S}_i(t) \times \vec{I}_{ik}(t) \tag{4}$$

Which are the dynamically coupled differential equations for electron spin and nuclear spin. Last week I attempted to write a function that could solve a simplified version of the equations above whilst I spent time setting up my workflow for the project as well as reading the Manolopoulos paper in more detail to get to grips with the equations better.

Rewriting the initial code

To start with this week I decided my first step was to rewrite coupled differential equation solver function from last week so that it correctly solves (3) and (4)

In [3]:

```
def semi_classical_ode(time, omega, a, initial):

    def ddt(y, t):
        s_x, s_y, s_z, i_x, i_y, i_z = y
        s = np.array([s_x, s_y, s_z])
        i = np.array([i_x, i_y, i_z])
        dsdt = np.cross((omega + a * s), i)
        didt = a * np.cross(s, i)
        return [dsdt[0], dsdt[1], dsdt[2], didt[0], didt[1], didt[2]]

    sol = odeint(ddt, initial, time)

    return sol[:, :3], sol[:, 3:]
```

To adapt my code from last time I had to rewrite $\vec{S}$ and $\vec{I}$ as vectors in the code. Due to the nature of the SciPy odeint syntax I had to pass in the individual components of the vectors then rewrite them as vectors to compute the cross product and then return them as individual components. I will experiment at a further time on whether doing the cross product myself is more efficient in terms of computation time in order to speed up the whole program

In order to solve the differential equations for $\vec{S}$ and $\vec{I}$ random initial vectors with lengths $S\sqrt{S+1}$ and $I\sqrt{I+1}$ must be generated. The code below generates these vectors:

In [4]:

```
def random_vec(r_vals):

    vals = []
    coords = []
    for r in r_vals:
        length = np.sqrt(r*(r+1))
        u = (np.random.random_sample()-0.5)*2
        theta = np.random.random_sample()*2*np.pi
        vec = np.array([np.sqrt(1-u**2)*np.cos(theta), np.sqrt(1-u**2)*np.sin(theta), u])
        vec = length*vec
        for val in vec:
            vals.append(val)
        coords += [r, u, theta]

    return vals, coords
```

In this function a random point on the sphere is picked using the equations from (5)-(8) [4] and a 3D vector of the required length is generated.

$$x = \sqrt{1-u^2}\cos\theta \tag{5}$$
$$y = \sqrt{1-u^2}\sin\theta \tag{6}$$
$$z = u \tag{7}$$
$$u \in [-1, 1], \theta \in [0, 2\pi] \tag{8}$$

The equations in (5)-(8) are used to get a uniform distribution of points as picking points based of $\theta \in [0, 2\pi]$ and $\phi \in [0, \pi]$ gives a non-uniform distribution of points on the surface of the sphere as there will be more clustered on around the poles of the sphere.

The next step to implementing the semi-classical method is performing a Monte Carlo integral [3] on equation (9).

$$R^{(i)}_{\alpha\beta} \approx \frac{1}{2\pi}\int d\Omega_S \prod_{k=1}^{N_i}\frac{1}{4\pi}\int d\Omega_{I_{ik}} S_{i\alpha}(0)S_{i\beta}(t) \tag{9}$$

Equation (9) approximates the electron spin correlation tensor where $S_{i\alpha}$ and $S_{i\beta}$ are the components of the electron spin vector calculated using (3) and (4).

Equation (9) can be simplified to equation (10) for approximating for one nuclear spin.

$$R_{\alpha\beta} \approx \frac{1}{8\pi^2}\int d\Omega_S \int d\Omega_I S_\alpha(0)S_\beta(t) \tag{10}$$

In [5]:

```
def monte_carlo_int(ode_solver, sampling_func, time, resolution=int(1e4)):

    spin_corr = np.zeros([3, 3, len(time)])
    count = 0
    for _ in range(resolution):
        y0 = sampling_func((1/2, 1/2))
        s, i = ode_solver(time, omega, a, y0[0])
        for alpha in range(len(spin_corr)):
            for beta in range(len(spin_corr[0])):
                spin_corr[alpha][beta] += (np.sin(y0[1][2]))*(1/(8*np.pi**2))*s[0][alpha]*s

    return (16*np.pi**2)*spin_corr/resolution
```

A Monte Carlo integral is a sum performed over a large sample of N values as shown in the equations below.

$$I = \int_{\Omega} f(\bar{x}) d\bar{x} \tag{11}$$

$$V = \int_{\Omega} d\bar{x} \tag{12}$$

$$I \approx V \frac{1}{N} \sum_{i=1}^{N} f(\bar{x}_i) \tag{13}$$

For the Monte Carlo integral random vectors are picked from the surfaces $d\Omega_S$ and $d\Omega_{I_{ik}}$ in order to get the samples required for the integral. The code above then performs the summation in equation (13) as required whilst also looping over all components of the spin correlation tensor.

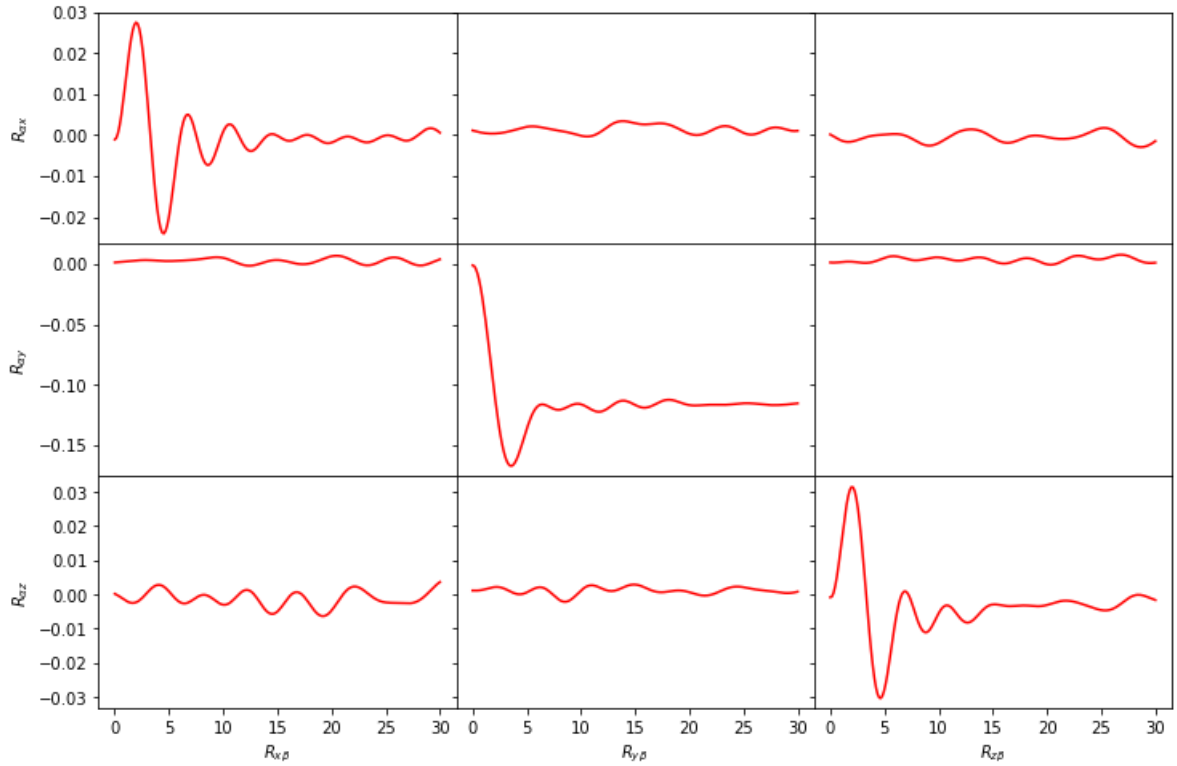
In [6]:

```
t = np.linspace(0, 30, 200)
B = 0.5e-3
omega = np.array([0, 0, -2*B])
a = -0.999985

s1 = time.time()
R = monte_carlo_int(semi_classical_ode, random_vec, t)
e1 = time.time()
print("Time taken:", round(e1-s1, 2), "s")
```

```
Time taken: 401.05 s
```

In [7]:

```
fig, axs = plt.subplots(3, 3, sharex='col', sharey='row', gridspec_kw={'hspace': 0, 'wspace
labels = ('x', 'y', 'z')
for i in range(3):
    for j in range(3):
        axs[i, j].plot(t, R[i][j], 'r-')
        if j == 0:
            axs[i, j].set_ylabel(r'$R_{\alpha '+str(labels[i]) + r'}$')
        if i == 2:
            axs[i, j].set_xlabel(r'$R_{ '+str(labels[j])+r'\beta}$')
```



Running the Monte Carlo integral for 10000 samples gives a converging spin correlation tensor as shown in the graph above however the data it has produced is not correct as according to the Manolopoulos paper [3] the tensor will have the following conditions:

$$R_{xx} = R_{yy} \tag{14}$$
$$R_{xy} = -R_{yx} \tag{15}$$
$$R_{xz} = R_{yz} = R_{zx} = R_{zy} = 0 \tag{16}$$

As can be seen in the graph above the spin correlation tensor does not match the expected result as $R_{xx} \neq R_{yy}$ and R_{xy} and R_{yx} are supposed to not have a constant value of 0 through time.

Another thing that should be noted is the slow time of the Monte Carlo integral function, this obviously needs lots of optimisation to run more efficiently.

Name	Call Count	Time (ms)	Own Time (ms)
cross	967508	52867 80.9%	17697 27.1%
normalize_axis_tuple	5805048	14627 22.4%	9668 14.8%
moveaxis	2902524	27681 42.3%	9258 14.2%
ddt	483754	60184 92.1%	4082 6.2%
<built-in method numpy.core._mu	3893369	54439 83.3%	3056 4.7%
<built-in method numpy.core._mu	8708704	2386 3.6%	2386 3.6%
<built-in method scipy.integrate._o	1000	62543 95.7%	2358 3.6%
<built-in method numpy.array>	3900117	2120 3.2%	2120 3.2%
<listcomp>	5805048	3183 4.9%	1995 3.1%
moveaxis	2902524	31475 48.1%	1739 2.7%
<built-in method builtins.len>	18454409	1675 2.6%	1675 2.6%
<method 'transpose' of 'numpy.nd	2902524	1238 1.9%	1238 1.9%
<built-in method builtins.sorted>	2904822	1080 1.7%	1080 1.7%
cross	967508	55070 84.2%	678 1.0%
<built-in method _operator.index>	5805049	635 1.0%	635 1.0%
<listcomp>	2902524	630 1.0%	630 1.0%

The screenshot of the table above shows the results of running the program through a profiler which identifies the time spent on various function calls. As can be seen above the NumPy cross function uses up a significant amount of computational time as well as various NumPy array methods such as normalize_axis_tuple and moveaxis which appear to be called in the creation of the arrays for ODE solver.

It appears I have made several mistakes throughout the program that I will go through next week and attempt to fix them as well as making the program more efficient.

Date: 05-11/10/2020

Aims

My aims this week include:

- Finding ways to generate faster results through optimisation of code
- Find correct parameters to pass to differential equation solver

Optimisation through computational inexpensive functions

To first optimise my code I will attempt to find various ways of solving the coupled differential equations featured in last weeks work from Manolopoulos [3]. First I will define some basic constants that form the basis of the inputs needed to solve the ODE's. I have also included the sampling function used as initial values must be sampled from the surface of the respective sphere surfaces of the electron and nuclear spins to solve the ODE's.

It is important to optimise this part of the code as although they will appear to be fast they are called thousands of times for the Monte Carlo integration so small decreases in run time for the function can snowball into large time saves for the Monte Carlo integration.

In [8]:

```
import timeit

t = np.linspace(0, 30, 200)
mag_field = 0.5
gamma = 2.002319
omega = np.array([0, 0, gamma*mag_field])
a = -0.999985

def sphere_sampling_func(r_vals):
    vals = []
    for r in r_vals:
        length = np.sqrt(r * (r + 1))
        u = (np.random.random_sample() - 0.5) * 2
        theta = np.random.random_sample() * 2 * np.pi
        vec = np.array([np.sqrt(1 - u ** 2) * np.cos(theta), np.sqrt(1 - u ** 2) * np.sin(t
        vec = length * vec
        for val in vec:
            vals.append(val)

    return np.array(vals)

y0_vals = np.zeros([100, 6])
for y0 in range(len(y0_vals)):
    y0_vals[y0] += sphere_sampling_func([1/2, 1/2])

solver_times = []
solver_times_error = []
```

The code below represents the initial attempt to solve the ODE's using SciPy's ODE solver:

In [9]:

```
def basic_ode_solver(t, omega_vec, a, initial):

    def ddt(y, t):
        s_x, s_y, s_z, i_x, i_y, i_z = y
        s = np.array([s_x, s_y, s_z])
        i = np.array([i_x, i_y, i_z])
        dsdt = np.cross((omega_vec + a * i), s)
        didt = np.cross(a * s, i)
        return [dsdt[0], dsdt[1], dsdt[2], didt[0], didt[1], didt[2]]

    sol = odeint(ddt, initial, t)

    return sol[:, :3]
```

In [10]:

```
lap_times = []
tt = timeit.default_timer()
for y0 in y0_vals:
    lt = timeit.default_timer()
    s = basic_ode_solver(t, omega, a, y0)
    lap_times.append(timeit.default_timer()-lt)
print("Time taken:", round(timeit.default_timer() - tt, 5), "s")
solver_times.append((timeit.default_timer() - tt)/len(y0_vals))
solver_times_error.append(np.std(lap_times))
```

```
Time taken: 5.94986 s
```

This code represents a good baseline to compare other results to with as it is my first initial attempt at solving the problem.

My next attempt at solving the problem involved hardcoding the cross product to try and reach maximum efficiency:

In [11]:

```
def no_np_ode_solver(t, omega_vec, a, initial):
    def ddt(y, t):
        s_x, s_y, s_z, i_x, i_y, i_z = y

        return [
            ((a*i_y*s_z)-((a*i_z+omega_vec[-1])*s_y)),
            -((a*i_x*s_z)-((a*i_z+omega_vec[-1])*s_x)),
            ((a*i_x*s_y) - (a*i_y*s_x)),
            a*((s_y*i_z) - (s_z*i_y)),
            -a*((s_x*i_z) - (s_z*i_x)),
            a*((s_x*i_y) - (s_y*i_x))
        ]

    sol = odeint(ddt, initial, t)

    return sol[:, :3]
```

In [12]:

```
lap_times = []
tt = timeit.default_timer()
for y0 in y0_vals:
    lt = timeit.default_timer()
    s = no_np_ode_solver(t, omega, a, y0)
    lap_times.append(timeit.default_timer()-lt)
print("Time taken:", round(timeit.default_timer() - tt, 5), "s")
solver_times.append((timeit.default_timer() - tt)/len(y0_vals))
solver_times_error.append(np.std(lap_times))
```

```
Time taken: 0.59559 s
```

As can be seen above this code is incredibly fast compared to my first attempt however it is not flexible at all so when I eventually adapt the code to work for a greater number of spins this approach will probably die out.

Next I tried rewriting the basic solver so that it was neater and easier to follow in the hopes that this might make a difference in runtime:

In [13]:

```
def np_cross_ode_solver(t, omega_vec, a, initial):

    def ddt(y, t):
        s = y[:3]
        i = y[3:]
        dsdt = np.cross((omega_vec + a * i), s)
        didt = np.cross(a * s, i)

        return np.concatenate((dsdt, didt), axis=None)

    sol = odeint(ddt, initial, t)

    return sol[:, :3]
```

In [14]:

```
lap_times = []
tt = timeit.default_timer()
for y0 in y0_vals:
    lt = timeit.default_timer()
    s = np_cross_ode_solver(t, omega, a, y0)
    lap_times.append(timeit.default_timer()-lt)
print("Time taken:", round(timeit.default_timer() - tt, 5), "s")
solver_times.append((timeit.default_timer() - tt)/len(y0_vals))
solver_times_error.append(np.std(lap_times))
```

```
Time taken: 5.68804 s
```

It did not make a difference however.

My next approach to the problem was to try using a different blackbox ODE solver and see if it was faster than ODE_int:

In [15]:

```
def np_cross_ivp_solver(t, omega_vec, a, initial):

    def ddt(t, y):
        s = y[:3]
        i = y[3:]
        dsdt = np.cross((omega_vec + a * i), s)
        didt = np.cross(a * s, i)

        return np.concatenate((dsdt, didt), axis=None)

    sol = solve_ivp(ddt, [0, 30], initial, t_eval=t)

    return sol
```

In [16]:

```
lap_times = []
tt = timeit.default_timer()
for y0 in y0_vals:
    lt = timeit.default_timer()
    s = np_cross_ivp_solver(t, omega, a, y0)
    lap_times.append(timeit.default_timer()-lt)
print("Time taken:", round(timeit.default_timer() - tt, 5), "s")
solver_times.append((timeit.default_timer() - tt)/len(y0_vals))
solver_times_error.append(np.std(lap_times))
```

```
Time taken: 3.45605 s
```

This solver was indeed faster when given the numpy cross function so I decided to try it with the hardcoded cross function:

In [17]:

```
def no_np_ivp_solver(t, omega_vec, a, initial):

    def ddt(t, y):
        s_x, s_y, s_z, i_x, i_y, i_z = y

        return [
            ((a*i_y*s_z)-((a*i_z+omega_vec[-1])*s_y)),
            -((a*i_x*s_z)-((a*i_z+omega_vec[-1])*s_x)),
            ((a*i_x*s_y) - (a*i_y*s_x)),
            a*((s_y*i_z) - (s_z*i_y)),
            -a*((s_x*i_z) - (s_z*i_x)),
            a*((s_x*i_y) - (s_y*i_x))
        ]

    sol = solve_ivp(ddt, [0, 30], initial, t_eval=t)

    return sol
```

In [18]:

```
lap_times = []
tt = timeit.default_timer()
for y0 in y0_vals:
    lt = timeit.default_timer()
    s = no_np_ivp_solver(t, omega, a, y0)
    lap_times.append(timeit.default_timer()-lt)
print("Time taken:", round(timeit.default_timer() - tt, 5), "s")
solver_times.append((timeit.default_timer() - tt)/len(y0_vals))
solver_times_error.append(np.std(lap_times))
```

```
Time taken: 1.13365 s
```

Interestingly this was actually slower than using ODE_int.

My next attempt was to convert one of the vectors into a matrix and use the numpy dot product as this should be significantly faster:

In [19]:

```
def skew(x):
    return np.array([[0, -x[2], x[1]],
                     [x[2], 0, -x[0]],
                     [-x[1], x[0], 0]])

def np_matrix_ode_solver(t, omega_vec, a, initial):

    def ddt(y, t):
        s = y[:3]
        i = y[3:]

        i_m = skew(omega_vec + a * i)
        dsdt = np.dot(i_m, s)
        s_m = skew(a*s)
        didt = np.dot(s_m, i)

        return np.concatenate((dsdt, didt), axis=None)

    sol = odeint(ddt, initial, t)

    return sol[:, :3]
```

In [20]:

```
lap_times = []
tt = timeit.default_timer()
for y0 in y0_vals:
    lt = timeit.default_timer()
    s = np_matrix_ode_solver(t, omega, a, y0)
    lap_times.append(timeit.default_timer()-lt)
print("Time taken:", round(timeit.default_timer() - tt, 5), "s")
solver_times.append((timeit.default_timer() - tt)/len(y0_vals))
solver_times_error.append(np.std(lap_times))
```

```
Time taken: 1.30965 s
```

It was fast but still not as good as the hardcoded version however this approach is definitely one that can be altered for multiple spins so more experimentation is required.

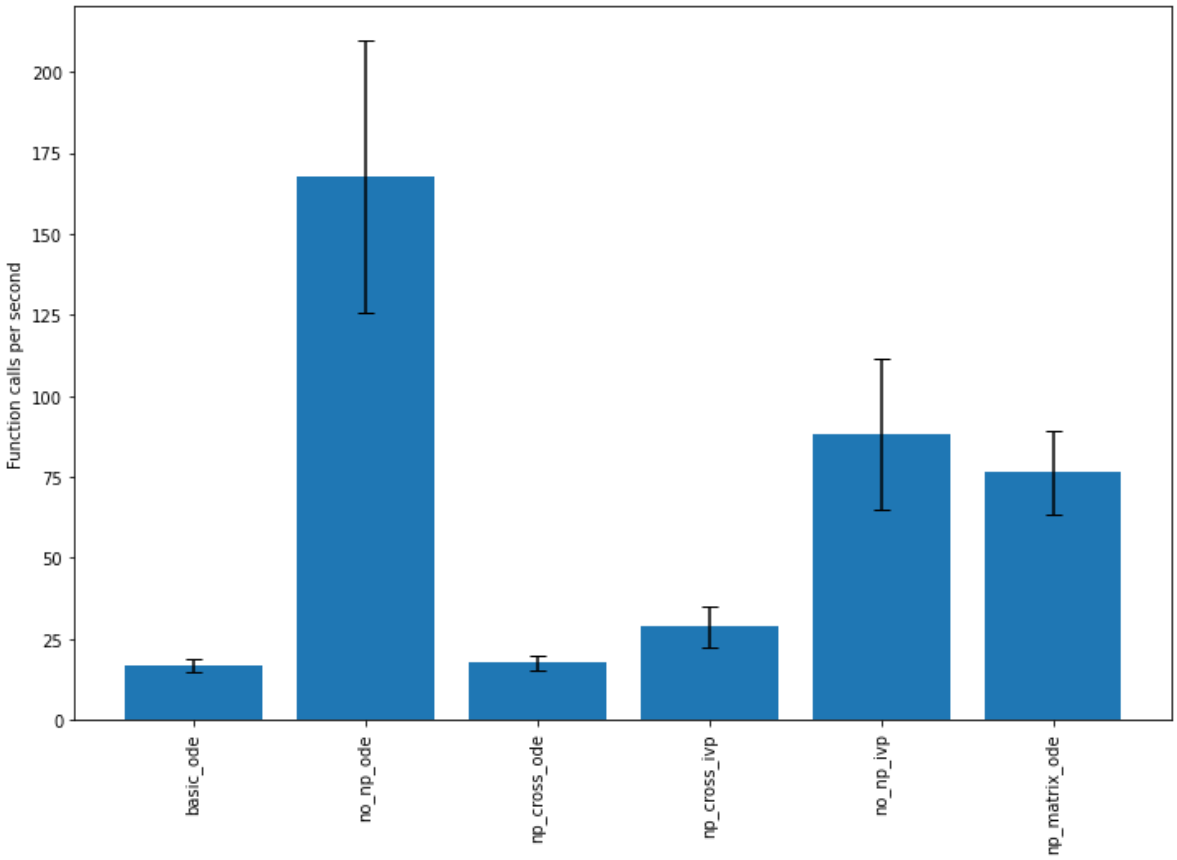
Analysis

In [21]:

```
plt.figure(figsize=(12,8))
recip_solver_times = [1/x for x in solver_times]
recip_errors = [solver_times_error[i]/(solver_times[i])**2 for i in range(len(solver_times)
solver_names = ["basic_ode", "no_np_ode", "np_cross_ode", "np_cross_ivp", "no_np_ivp", "np_

plt.bar(solver_names, recip_solver_times, yerr=recip_errors, capsize=5)
plt.ylabel('Function calls per second')
plt.xticks(rotation=90)
```

Out[21]:

```
([0, 1, 2, 3, 4, 5], <a list of 6 Text major ticklabel objects>)
```



As can be seen in the graph above the hardcoded cross function in the no_np_ode solver performs the best along with it's IVP solver companion. The error in this approach is quite large however even at the lower end it still performs better than any other solver being an order of magnitude faster than the original solver. Next week

when I change the code so that calculation works for multiple nuclear spins I will have to see how the other solvers hold up.

Correct parameters

Upon closer inspection of the paper I realised I had made some errors as I had introduced some $\sin(\theta)$ functions that were incorrect as well as misunderstanding the unit system they has used through out the paper.

In [22]:

```
def monte_carlo_int(ode_solver, sampling_func, time, resolution=int(1e4)):

    spin_corr = np.zeros([3, 3, len(time)])
    count = 0
    for _ in range(resolution):
        y0 = sampling_func((1/2, 1/2))
        s = ode_solver(time, omega, a, y0)
        for alpha in range(len(spin_corr)):
            for beta in range(len(spin_corr[0])):
                spin_corr[alpha][beta] += (1/(8*np.pi**2))*s[0][alpha]*s[:, beta]

    return (16*np.pi**2)*spin_corr/resolution
```

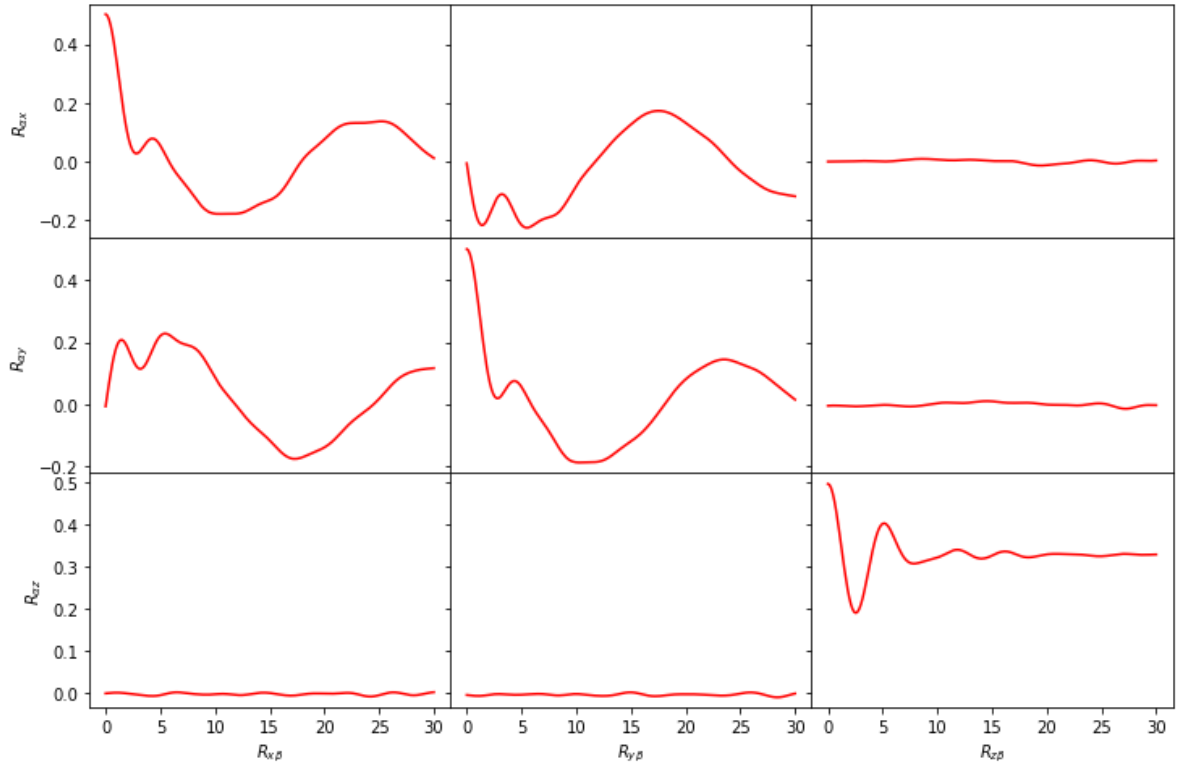
In [23]:

```
t = np.linspace(0, 30, 200)
B = -0.5
gamma = 1
omega = np.array([0, 0, -gamma*B])
a = -0.999985

s1 = timeit.default_timer()
R = monte_carlo_int(no_np_ode_solver, sphere_sampling_func, t)
print("Time taken:", round(timeit.default_timer()-s1, 2), "s")
```

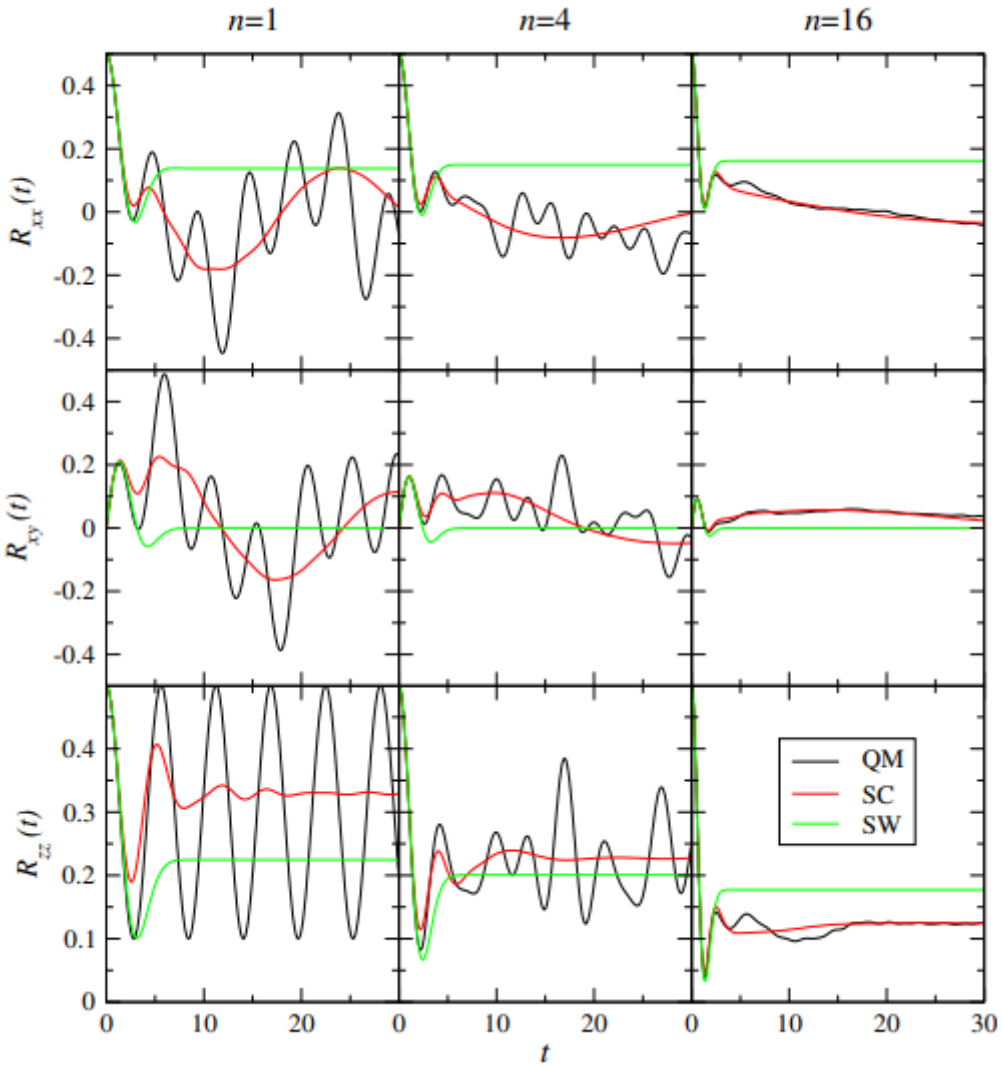
```
Time taken: 53.74 s
```

In [24]:

```
fig, axs = plt.subplots(3, 3, sharex='col', sharey='row', gridspec_kw={'hspace': 0, 'wspace
labels = ('x', 'y', 'z')
for i in range(3):
    for j in range(3):
        axs[j, i].plot(t, R[i][j], 'r-')
        if j == 0:
            axs[i, j].set_ylabel(r'$R_{\alpha '+str(labels[i]) + r'}$')
        if i == 2:
            axs[i, j].set_xlabel(r'$R_{ '+str(labels[j])+r'\beta}$')
```



With the now corrected function one can see that it has perfectly replicated the relevant components from the figure in the left hand column in Fig 2 from Manolopoulos et al [3] which can be seen below.



Date: 12-18/10/2020

Aims

- Add multiprocessing functionality to the Monte Carlo integration
- Generalise the Monte Carlo integration and ODE solver to work for k number of spins

K Number of spins

Following on from my work last week I have written some ODE solvers that are more generalised and work for any number of spins.

In [25]:

```
def np_cross_ode_solver(t, omega_vec, a, initial):
    def cross(x, y):
        return np.array([
            (x[1] * y[2]) - (x[2] * y[1]),
            -((x[0] * y[2]) - (x[2] * y[0])),
            (x[0] * y[1]) - (x[1] * y[0])
        ])

    def ddt(y, t):
        new_vec = np.zeros([len(y) // 3, 3])

        s = y[:3]
        i = y[3:]
        i = i.reshape(len(i) // 3, 3)
        omega = omega_vec + np.sum(a * i, axis=0)

        new_vec[0] += cross(omega, s)

        new_vec[1:] += np.cross(a * s, i)

        return new_vec.reshape(len(y))

    sol = odeint(ddt, initial, t)

    return sol[:, :3]
```

In [26]:

```
def einsum_ode_solver(t, omega_vec, a, initial):

    eijk = np.zeros((3, 3, 3))
    eijk[0, 1, 2] = eijk[1, 2, 0] = eijk[2, 0, 1] = 1
    eijk[0, 2, 1] = eijk[2, 1, 0] = eijk[1, 0, 2] = -1
    index = np.array([val for val in range((len(initial)-3)//3)])

    def ddt(y, t):
        new_vec = np.zeros([len(y) // 3, 3])

        s = y[:3]
        i = y[3:]

        i = i.reshape(len(i) // 3, 3)
        omega = omega_vec + np.sum(a * i, axis=0)


        u = omega[None, :]
        v = s[None, :]
        new_vec[0] += np.einsum('ijk,uj,vk->uvi', eijk, u, v, optimize=True)[0, 0]

        u = (a*s).reshape(i.shape)
        v = i
        i_vecs = np.einsum('ijk,uj,vk->uvi', eijk, u, v, optimize=True)
        new_vec[1:] += i_vecs[index, index]

        return new_vec.reshape(len(y))

    sol = odeint(ddt, initial, t)

    return sol[:, :3]
```

In [27]:

```
def nested_einsum_ode_solver(t, omega_vec, a, initial):

    eijk = np.zeros((3, 3, 3))
    eijk[0, 1, 2] = eijk[1, 2, 0] = eijk[2, 0, 1] = 1
    eijk[0, 2, 1] = eijk[2, 1, 0] = eijk[1, 0, 2] = -1
    index = np.array([val for val in range((len(initial)-3)//3)])

    def ddt(y, t):
        new_vec = np.zeros([len(y) // 3, 3])

        s = y[:3]
        i = y[3:]

        i = i.reshape(len(i) // 3, 3)
        omega = omega_vec + np.sum(a * i, axis=0)


        u = omega[None, :]
        v = s[None, :]
        new_vec[0] += np.einsum('iuk,vk->uvi', np.einsum('ijk,uj->iuk', eijk, u), v)[0, 0]

        u = (a*s).reshape(i.shape)
        v = i
        i_vecs = np.einsum('iuk,vk->uvi', np.einsum('ijk,uj->iuk', eijk, u), v)
        new_vec[1:] += i_vecs[index, index]

        return new_vec.reshape(len(y))

    sol = odeint(ddt, initial, t)

    return sol[:, :3]
```

In [28]:

```
def np_matrix_ode_solver(t, omega_vec, a, initial):

    def skew(x):
        return np.array([[0, -x[2], x[1]],
                         [x[2], 0, -x[0]],
                         [-x[1], x[0], 0]])

    def ddt(y, t):
        new_vec = np.zeros([len(y) // 3, 3])
        s = y[:3]
        i = y[3:]

        i = i.reshape(len(i) // 3, 3)
        omega = omega_vec + np.sum(a * i, axis=0)

        o_m = skew(omega)
        new_vec[0] += np.dot(o_m, s)
        for index, a_val in enumerate(a):
            s_m = skew(a_val*s)
            new_vec[index+1] += np.dot(s_m, i[index])

        return new_vec.reshape(len(y))

    sol = odeint(ddt, initial, t)

    return sol[:, :3]
```

The four ODE solvers are all variations of the same approach with different ways of calculating the cross product.

The np_cross function uses the default numpy cross function which despite underperforming last week may do much better this week as it can use numpy vectorization of arrays to calculate multiple cross products faster than just looping through each set of cross products.

The einsum and nested einsum functions use Einstein summation to calculate the cross product using the levi-civita symbols [5].

The matrix approach from the last one attempts to capitalise on the speed of the matrix dot product method of calculating a cross product from last week.

In [31]:

```
# Testing function
N_spins = 5
tests = 10
t = np.linspace(0, 30, 200)
mag_field = -0.5
gamma = 1
omega = np.array([0, 0, -gamma * mag_field])
a = np.random.rand(N_spins).reshape(N_spins, 1)

test_funcs = [np_cross_ode_solver, nested_einsum_ode_solver]
time_results = np.zeros([N_spins, len(test_funcs)])
error = np.zeros([N_spins, len(test_funcs)])

for n in range(1,N_spins+1):

    samples = np.zeros([tests, (n + 1) * 3])
    for sample in range(len(samples)):
        samples[sample] += sphere_sampling_func(np.repeat(0.5, n))

    for index, func in enumerate(test_funcs):
        lap_times = []
        tt = timeit.default_timer()
        for sample in samples:
            lt = timeit.default_timer()
            s = func(t, omega, a[:n], sample)
            lap_times.append(timeit.default_timer()-lt)
        time_results[n-1, index] += (timeit.default_timer()-tt)/tests
        error[n-1, index] += np.std(lap_times)


save_name = f"{str(N_spins)}_{str(tests)}_funcs_{len(test_funcs)}"
np.savez(save_name, results=time_results, error=error, func_names=[str(x.__name__) for x in
```

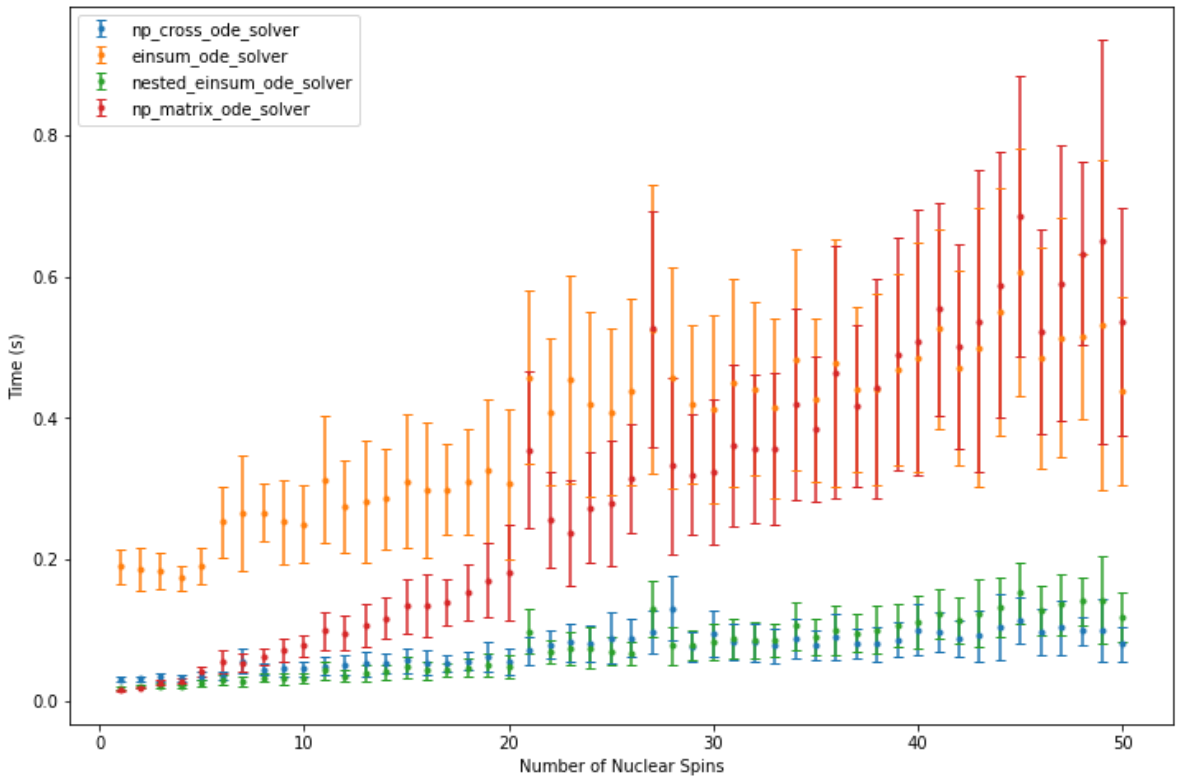
I tested all four functions for an increasing number of spins with 20 trajectories each in order to compare their efficiencies. The results of this can be seen below:

In [32]:

```
plt.figure(figsize=(12,8))
data = np.load("50_20_funcs_4.npz")
results = data["results"]
error = data["error"]

for n in range(len(data["func_names"])):
    plt.errorbar(range(1, len(results)+1), results[:, n], marker="o", markersize=3, linesty
plt.legend(data["func_names"])
plt.xlabel("Number of Nuclear Spins")
plt.ylabel("Time (s)")
```

Out[32]:

```
Text(0, 0.5, 'Time (s)')
```



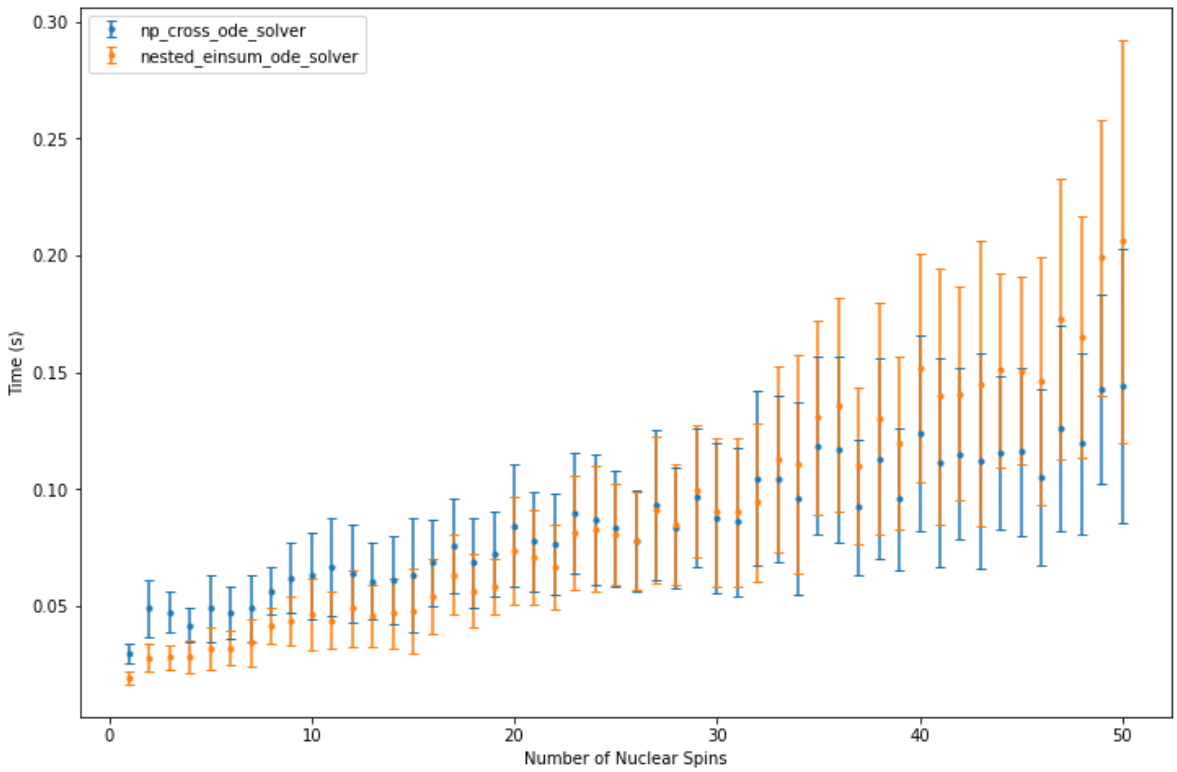
As can be seen in the graph above the matrix dot method and einsum method do not scale well at all with increasing numbers of nuclear spins. There is quite large variation in how long each solve takes as the number of spins increases, I am not sure why this happens.

In [33]:

```
plt.figure(figsize=(12,8))
data = np.load("50_20_funcs_2.npz")
results = data["results"]
error = data["error"]

for n in range(len(data["func_names"])):
    plt.errorbar(range(1, len(results)+1), results[:, n], marker="o", markersize=3, linesty
plt.legend(data["func_names"])
plt.xlabel("Number of Nuclear Spins")
plt.ylabel("Time (s)")
```

Out[33]:

```
Text(0, 0.5, 'Time (s)')
```



Taking a closer look at the numpy cross function and the nested einsum solver the einsum solver appears to have an advantage for a lower number of spins before the numpy cross function overtakes it at around 25 spins.

Multiprocessing

An extremely important feature to introduce early is the ability to multiprocess and take full advantage of the full computational power available. By using the python multiprocessing module I have written code that can uses multiple cores to perform the differential equation solves simultaneously. For a low number of trajectories the multiprocessing may take more time as there is an overhead involved in running the multiprocessing but for the necessary project work it will save significant amounts of time.

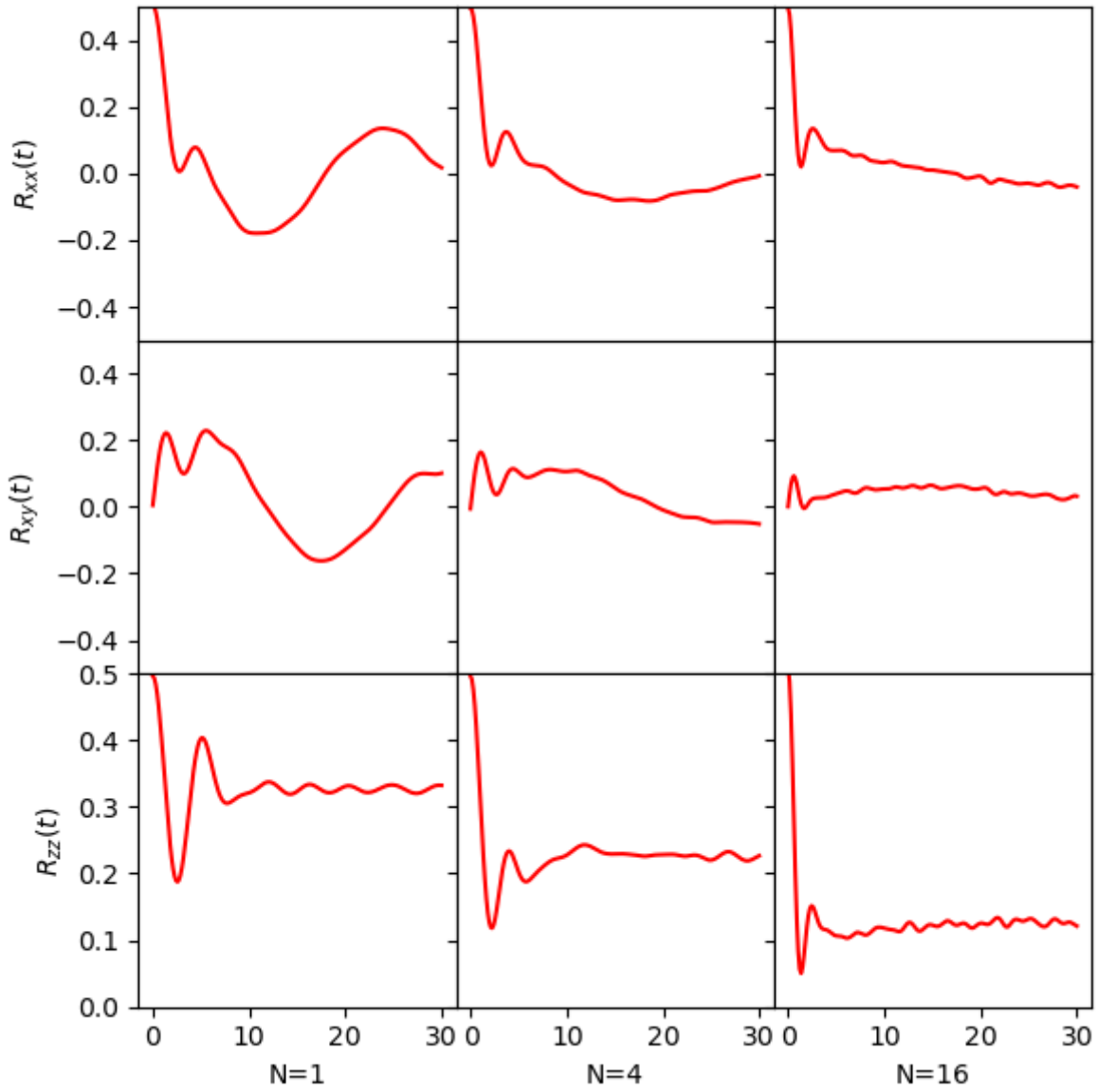
In [34]:

```
def monte_carlo_int_mp(y0, a):

    spin_corr = np.zeros([3, 3, len(t)])
    s = einsum_ode_solver(t, omega, a, y0)
    for alpha in range(len(spin_corr)):
        for beta in range(len(spin_corr[0])):
            spin_corr[alpha][beta] += 2*s[0][alpha]*s[:, beta]

    return spin_corr
```

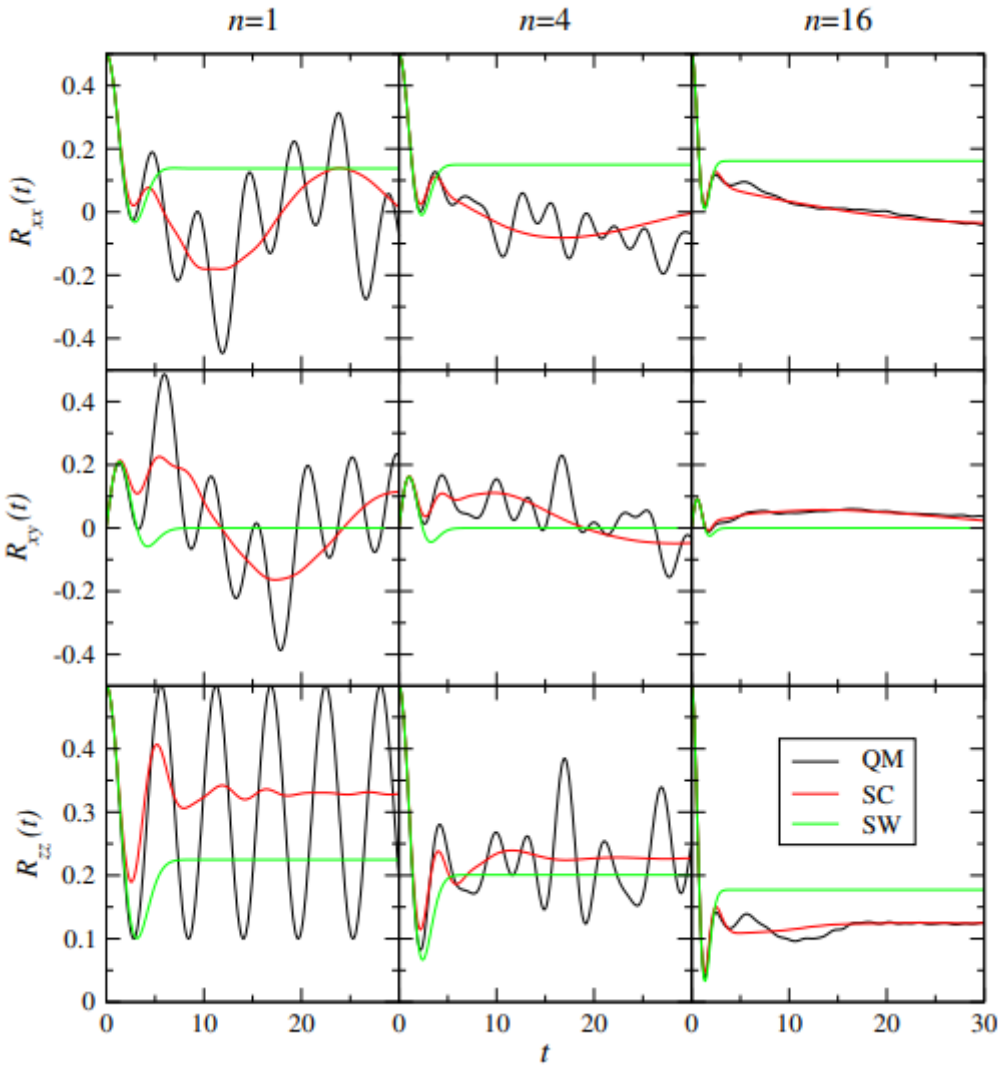
In [35]:

```
def integrate(a, spins, resolution=int(1e4)):
    st = timeit.default_timer()
    print(a)

    samples = np.zeros([resolution, (len(spins) + 1) * 3])
    a_vals = np.zeros(resolution, len(spins), 1)
    for index in range(len(samples)):
        sample = sphere_sampling_func(spins)
        samples[index] += sample
        a_vals[index] += a

    print(f"Processing {resolution} trajectories with: {mp.cpu_count() - 1} cores")
    with mp.Pool(mp.cpu_count() - 1) as pool:
        R_vec = pool.starmap(monte_carlo_int_mp, zip(samples, a_vals))

    R = np.zeros([3, 3, len(t)])
    for R_val in R_vec:
        R += R_val

    R = R / resolution

    print(f"Total time taken: {round(timeit.default_timer() - st, 2)} s")

    return R
```

I have now been able to almost perfectly recreate the figure 2 from Manolopoulos et al [3].



There are some fluctuations in the N=4 and more so in the N=16 graphs as I used the same number of samples meaning that I need to use more trajectories to caclculate a more accurate tensor for higher numbers of spins. For comparison the figure 2 from Manolopoulos et al [3] is shown below:



Date: 19-25/10/2020

Aims

- Alter differential equation solver such that it works for anisotropic hyperfine tensors

Hyperfine tensors

An important aspect of avian magnetoreception is that the hyperfine interactions are anisotropic as this allows the bird to have a directional sense rather than just a magnetic strength sensor. The anisotropic hyperfine interactions can be modeled using 3x3 matrices which means I must adapt my code to take these tensors.

I have done this below using matrix multiplication as well as the broadcasting features of numpy which allow me to combine arrays in ways which are much quicker than looping through.

In [36]:

```
def anisotropic_ode_solver(t, omega_vec, a, initial):

    def cross(x, y):
        return np.array([
            (x[1] * y[2]) - (x[2] * y[1]),
            -((x[0] * y[2]) - (x[2] * y[0])),
            (x[0] * y[1]) - (x[1] * y[0])
        ])

    def ddt(y, t):
        new_vec = np.zeros([len(y) // 3, 3])

        s = y[:3]
        i = y[3:]
        i = i.reshape(len(i) // 3, 3, 1)
        omega = omega_vec + np.sum((a@i).reshape((len(y)//3)-1, 3), axis=0)

        new_vec[0] += cross(omega, s)

        i = i.reshape((len(y)//3)-1, 3)
        new_vec[1:] += np.cross(a@s, i)

        return new_vec.reshape(len(y))

    sol = odeint(ddt, initial, t)

    return sol[:, :3]
```

In [37]:

```
def monte_carlo_int_mp(y0, a):

    spin_corr = np.zeros([3, 3, len(t)])
    s = anisotropic_ode_solver(t, omega, a, y0)
    for alpha in range(len(spin_corr)):
        for beta in range(len(spin_corr[0])):
            spin_corr[alpha][beta] += 2*s[0][alpha]*s[:, beta]

    return spin_corr
```

I have also adapted the multiprocessing function so that it can determine whether isotropic or anisotropic hyperfine tensors are being used.

In [38]:

```
def integrate(a, spins, resolution=int(1e4)):
    st = timeit.default_timer()
    print(a)

    samples = np.zeros([resolution, (len(spins) + 1) * 3])
    a_val_dimensions = [resolution]
    for dimension in a.shape:
        a_val_dimensions.append(dimension)
    a_vals = np.zeros(a_val_dimensions)
    for index in range(len(samples)):
        sample = sphere_sampling_func(spins)
        samples[index] += sample
        a_vals[index] += a

    print(f"Processing {resolution} trajectories with: {mp.cpu_count() - 1} cores")
    with mp.Pool(mp.cpu_count() - 1) as pool:
        R_vec = pool.starmap(monte_carlo_int_mp, zip(samples, a_vals))

    R = np.zeros([3, 3, len(t)])
    for R_val in R_vec:
        R += R_val

    R = R / resolution

    print(f"Total time taken: {round(timeit.default_timer() - st, 2)} s")

    return R
```

Date: 26/10-01/11/2020

Aims

- Simulate and study the electron spin correlation tensor for a simple Flavin molecule

Flavin semiquinone molecule

Flavin is a type of radical that is produced in cryptochromes and is therefore very key for the aim of the project. It has strong anisotropic hyperfine interactions due to the fact that it has nitrogen molecules with a nuclear spin of 1 [1].

Now equipped with the necessary tools I can begin using the semi classical simulation on actual real life molecules rather than hypothetical systems. Supplied with some data from my project supervisor I needed to convert it into a usable format to feed to the code.

In [39]:

```
N5 = [-2.41368, -0.0662465, -0.971492,
      -0.0662465, -2.44657, 0.0485258,
      -0.971492, 0.0485258, 43.5125]
N10 = [0.442319, 0.06085, 1.8016,
       0.06085, -0.0133137, -0.338064,
       1.8016, -0.338064, 23.1529]
H5 = [-2.38856, 1.8683, 0.514044,
       1.8683, -40.6401, 0.0339364,
       0.514044, 0.0339364, -27.8618]
H7 = [-0.39371, 0., 0.,
       0., -0.39371, 0.,
       0., 0., -0.39371]
H8 = [7.42385, 0., 0.,
      0., 7.42385, 0.,
      0., 0., 7.42385]
Hb1 = [10.3368, -0.86374, -1.23637,
      -0.86374, 7.56294, 0.373315,
      -1.23637, 0.373315, 6.9114]
Hb2 = [6.58082, 1.26851, -1.05455,
       1.26851, 3.84121, -0.306796,
      -1.05455, -0.306796, 3.24229]
H9 = [1.48621, 1.05252, 0.0516722,
      1.05252, 3.49893, 0.170705,
      0.0516722, 0.170705, -0.00498294]
H6 = [-2.24866, -1.73702, 0.0909162,
      -1.73702, -4.73778, -0.0762715,
       0.0909162, -0.0762715, -6.06812]
H3 = [-2.85739, 0.897874, 0.0128428,
       0.897874, 0.508656, 0.0599533,
       0.0128428, 0.0599533, -3.35597]
Hc = [1.65248, 0.140083, 0.145213,
      0.140083, -0.498108, -0.0304329,
      0.145213, -0.0304329, -0.30567]
```

My first step was to convert these lists into numpy arrays so that I could manipulate them easily.

In [40]:

```
a = np.array([N5, N10, H5, H7, H8, Hb1, Hb2, H9, H6, H3, Hc])
a.shape
```

Out[40]:

```
(11, 9)
```

Next I reshaped the array into an array of 3x3 matrices

In [41]:

```
a = a.reshape(11, 3, 3)
```

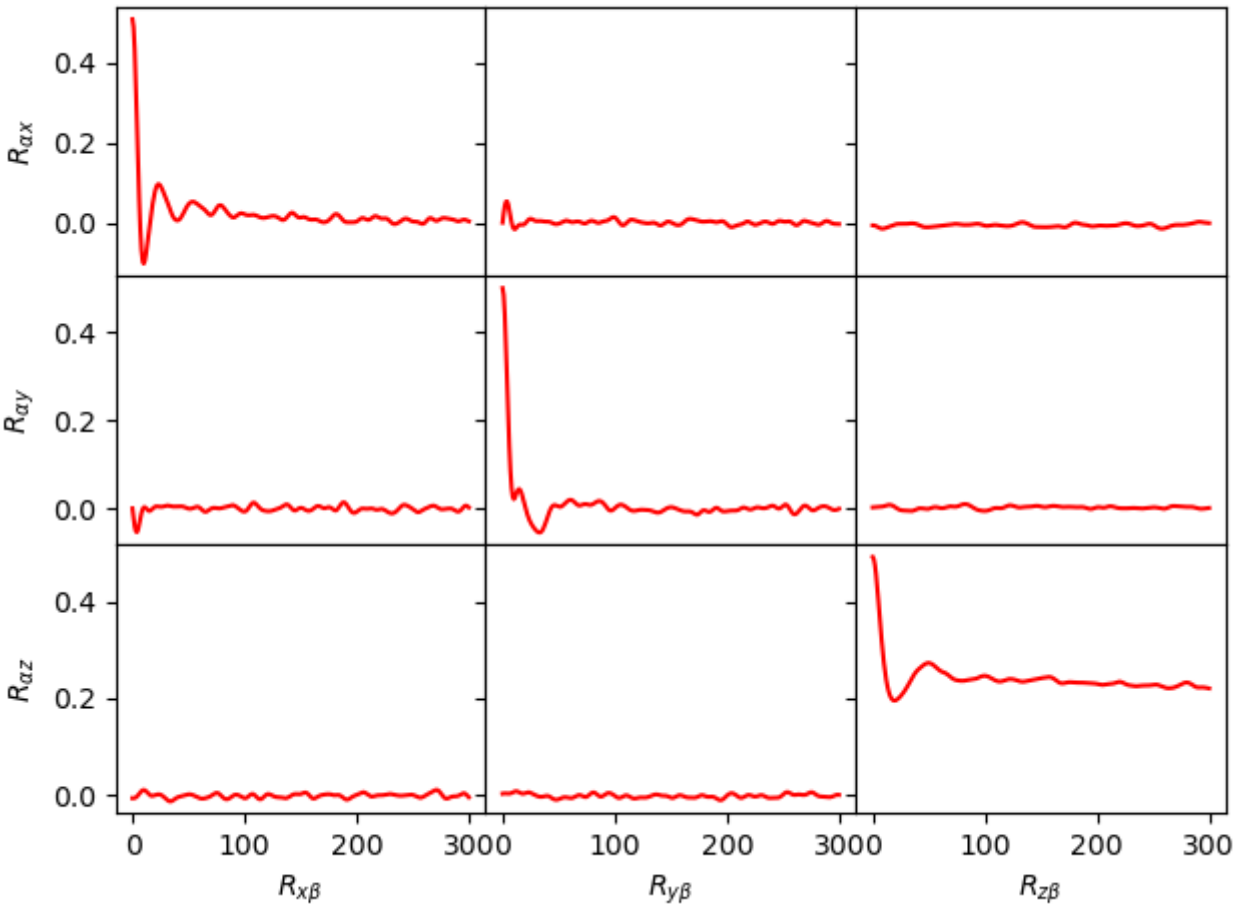
I then had to convert the units of the tensor from MHz to mT which I did by using the gyromagnetic ratio of an electron as shown below:

In [42]:

```
gamma_e = 1.76085963023e11
a *= 1e6
a /= gamma_e
a *= 1e3
print(a)
```

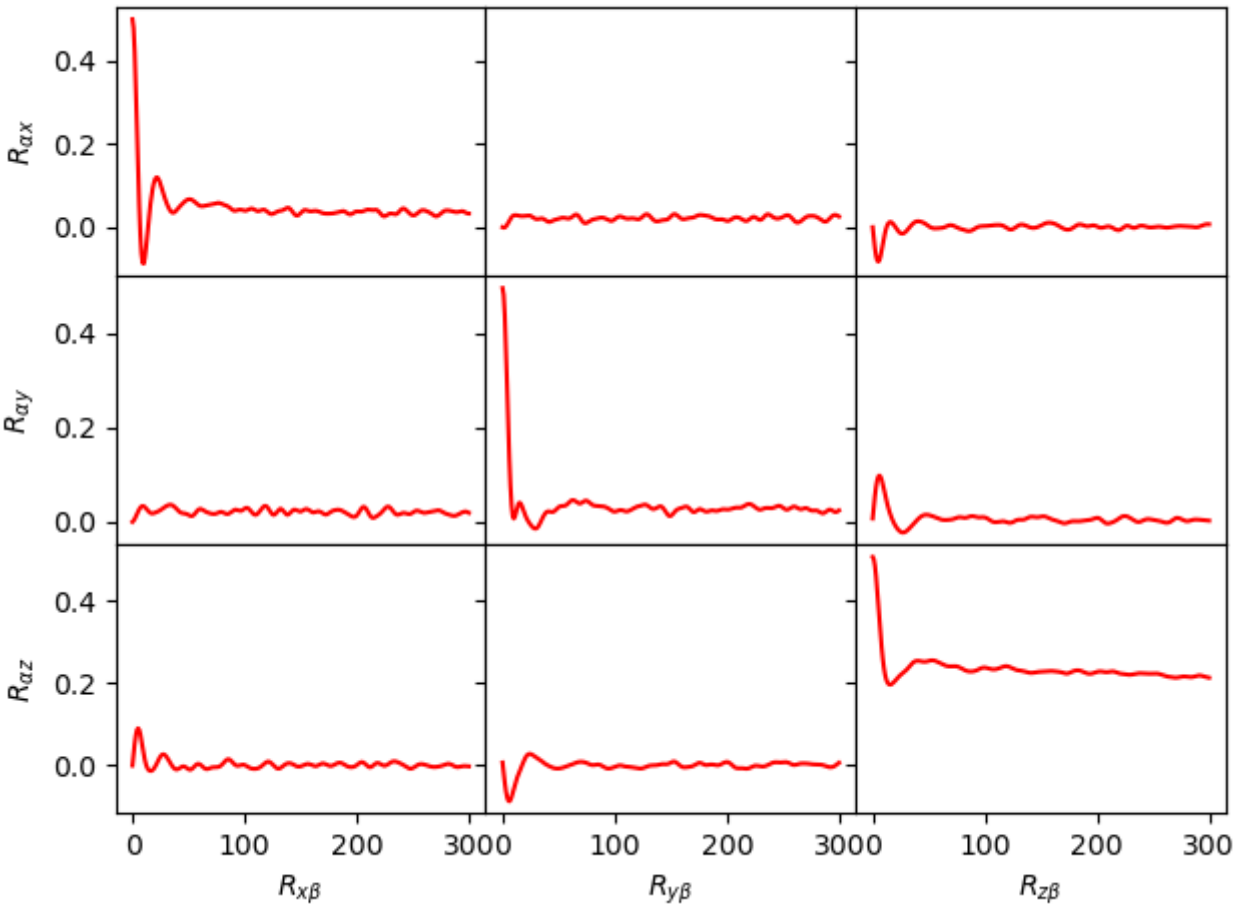
```
[[[-1.37073959e-02 -3.76216814e-04 -5.51714619e-03]
  [-3.76216814e-04 -1.38941796e-02  2.75580172e-04]
  [-5.51714619e-03  2.75580172e-04  2.47109419e-01]]

 [[ 2.51194923e-03  3.45569851e-04  1.02313664e-02]
  [ 3.45569851e-04 -7.56090933e-05 -1.91988046e-03]
  [ 1.02313664e-02 -1.91988046e-03  1.31486347e-01]]

 [[-1.35647383e-02  1.06101586e-02  2.91927869e-03]
  [ 1.06101586e-02 -2.30796932e-01  1.92726322e-04]
  [ 2.91927869e-03  1.92726322e-04 -1.58228399e-01]]

 [[-2.23589657e-03  0.00000000e+00  0.00000000e+00]
  [ 0.00000000e+00 -2.23589657e-03  0.00000000e+00]
  [ 0.00000000e+00  0.00000000e+00 -2.23589657e-03]]

 [[ 4.21603737e-02  0.00000000e+00  0.00000000e+00]
  [ 0.00000000e+00  4.21603737e-02  0.00000000e+00]
  [ 0.00000000e+00  0.00000000e+00  4.21603737e-02]]

 [[ 5.87031460e-02 -4.90521780e-03 -7.02140011e-03]
  [-4.90521780e-03  4.29502720e-02  2.12007246e-03]
  [-7.02140011e-03  2.12007246e-03  3.92501474e-02]]

 [[ 3.73727689e-02  7.20392460e-03 -5.98883626e-03]
  [ 7.20392460e-03  2.18144021e-02 -1.74230810e-03]
  [-5.98883626e-03 -1.74230810e-03  1.84131088e-02]]

 [[ 8.44025256e-03  5.97730780e-03  2.93448717e-04]
  [ 5.97730780e-03  1.98705788e-02  9.69441272e-04]
  [ 2.93448717e-04  9.69441272e-04 -2.82983374e-05]]

 [[-1.27702400e-02 -9.86461368e-03  5.16317135e-04]
  [-9.86461368e-03 -2.69060629e-02 -4.33149234e-04]
  [ 5.16317135e-04 -4.33149234e-04 -3.44611228e-02]]

 [[-1.62272446e-02  5.09906630e-03  7.29348313e-05]
  [ 5.09906630e-03  2.88868000e-03  3.40477452e-04]
  [ 7.29348313e-05  3.40477452e-04 -1.90587026e-02]]

 [[ 9.38450727e-03  7.95537575e-04  8.24671073e-04]
  [ 7.95537575e-04 -2.82877744e-03 -1.72829790e-04]
  [ 8.24671073e-04 -1.72829790e-04 -1.73591350e-03]]]
```

Using my multiprocessing code from last week I was able to simulate the flavin-semiquinone in a roughly earth strength magnetic field (~0.05 mT). Below is the spin correlation tensor:



For this calculation I used 10000 trajectories in the Monte Carlo integration with a timescale of $300mT$ or $1.7\mu s$ and a resolution of $1mT$ or $5.7ns$. To do proper analysis with this progress I need to use a time scale which is about $4\mu s$ and a resolution of $1ns$ however I cannot do this without access to a high performance computer.

I repeated the calculation but this time used a magnetic field going in both the X and Y direction to produce the following plot using the same statistics as before:



I will work out how to do some error analysis over the coming weeks in order to properly judge how good this data is.

Date: 02-08/11/2020

Aims

- Adapt code so that an electron spin correlation tensor can be calculated for a lone electron with zero spins

Zero spin electron spin correlation tensor

This week I fixed a small bug with the code where it did not work for calculating a spin correlation tensor of a lone electron.

I initially had to change the sphere sampling function so that it automatically inserted the electron spin so that it could be passed an empty array and only return the electron spin vector.

In [43]:

```
def sphere_sampling_func(r_vals):
    r_vals = np.insert(r_vals, 0, 0.5)  # for initial electron spin
    vals = []
    for r in r_vals:
        length = np.sqrt(r * (r + 1))
        u = (np.random.random_sample() - 0.5) * 2
        theta = np.random.random_sample() * 2 * np.pi
        vec = np.array([np.sqrt(1 - u ** 2) * np.cos(theta), np.sqrt(1 - u ** 2) * np.sin(t
        vec = length * vec
        for val in vec:
            vals.append(val)

    return np.array(vals)
```

By then feeding the Monte Carlo integration a blank array and one empty placeholder hyperfine tensor this spin correlation tensor can be calculated.

There is however an analytical way to solve this tensor:

Taking the differential equation for $\vec{S}$ is (17)

$$\frac{d}{dt}\vec{S}_i(t) = \left[\vec{\omega}_i + \sum_{k=1}^{N_i} a_{ik}\vec{I}_{ik}(t)\right] \times \vec{S}_i(t) \tag{17}$$

With zero nuclear spins this reduces to:

$$\frac{d}{dt}\vec{S}_i(t) = \omega \times \vec{S}_i(t) \tag{18}$$

Which can then be split into the following equations:

$$\frac{d}{dt}S_x = BS_y \tag{19}$$

$$\frac{d}{dt}S_y = -BS_x \tag{20}$$

$$S_z = 0 \tag{21}$$

Rearranging and substituting gives:

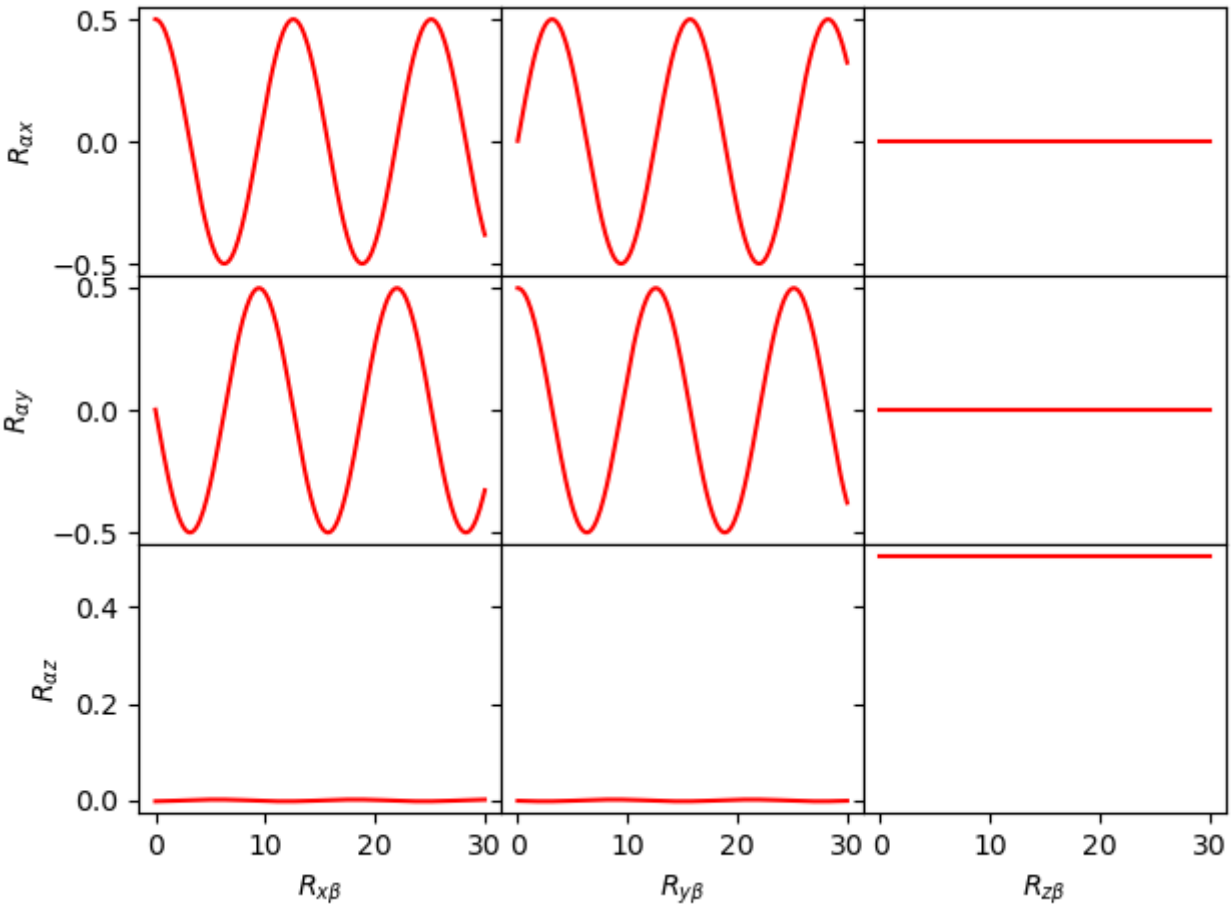
$$\frac{d^2}{dt^2}S_x = -B^2 S_x \tag{22}$$

$$\frac{d^2}{dt^2}S_y = -B^2 S_y \tag{23}$$

Which can be solved to give:

$$S_x = S_y = cos(Bt) \tag{24}$$

Looking at the output of the code below we can see that this correct:



Date: 09-15/11/2020

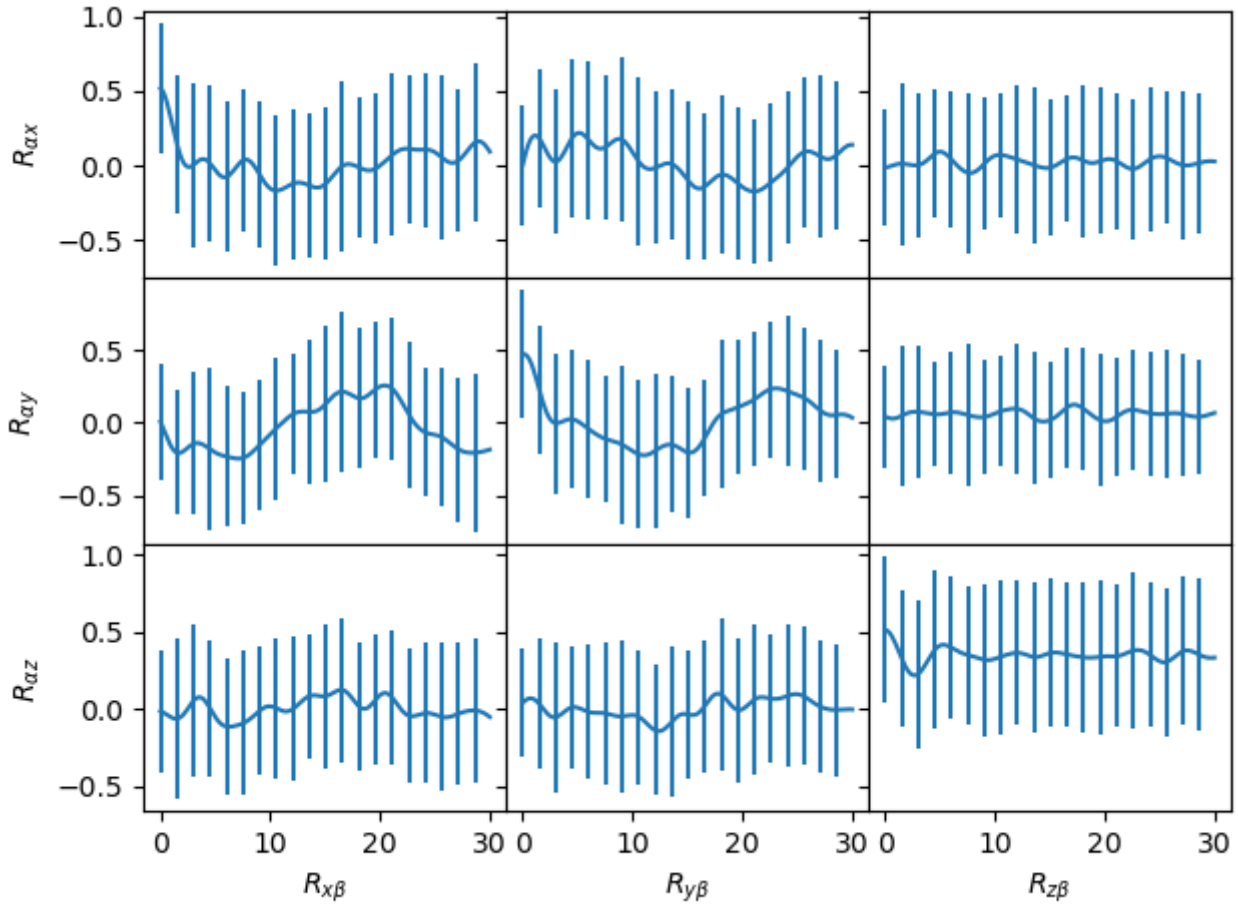
Aims

- Apply some error analysis to the spin correlation tensor code
- Look at varying error from increasing the number of spins

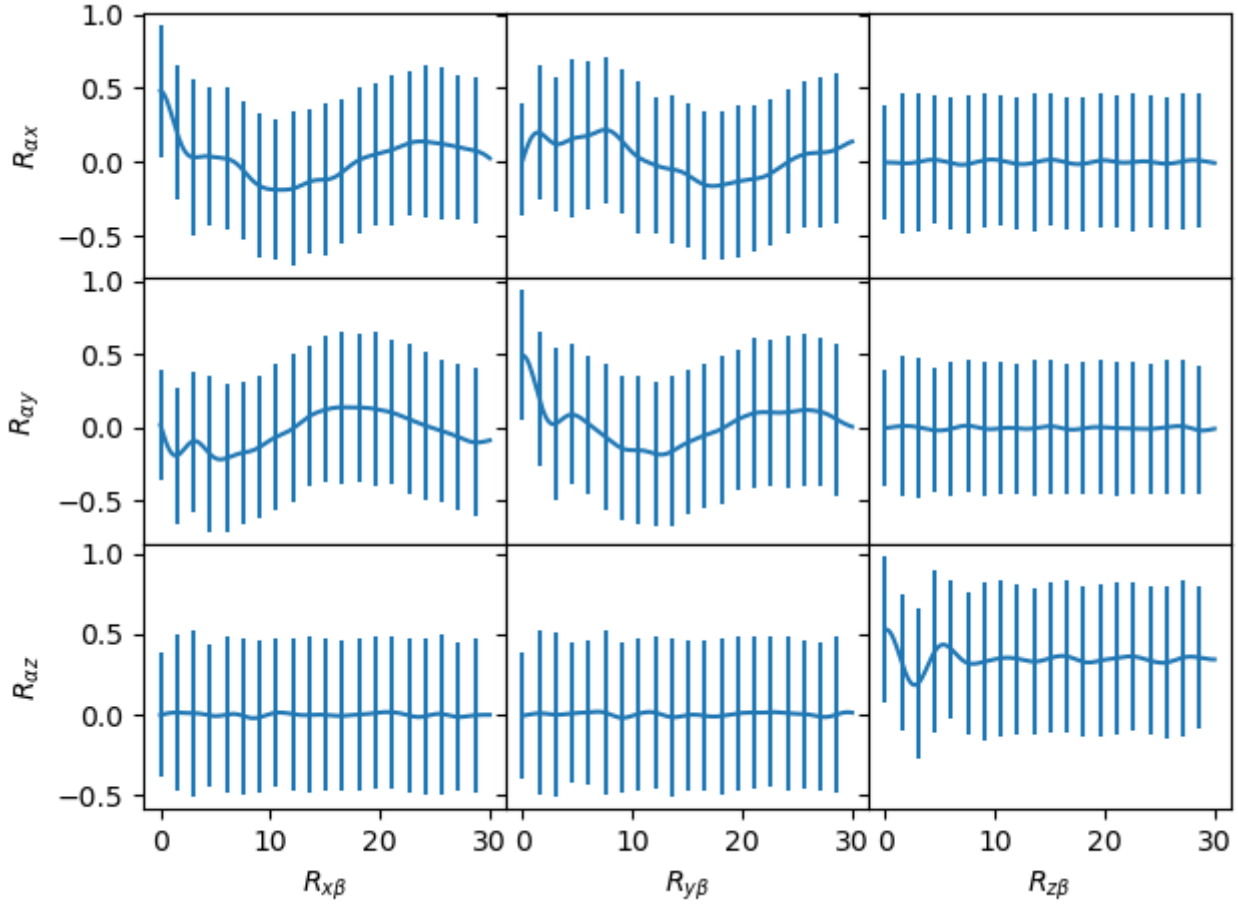
Error Analysis

My first idea to calculate the error involved taking the standard deviation of the different trajectories for different time points:

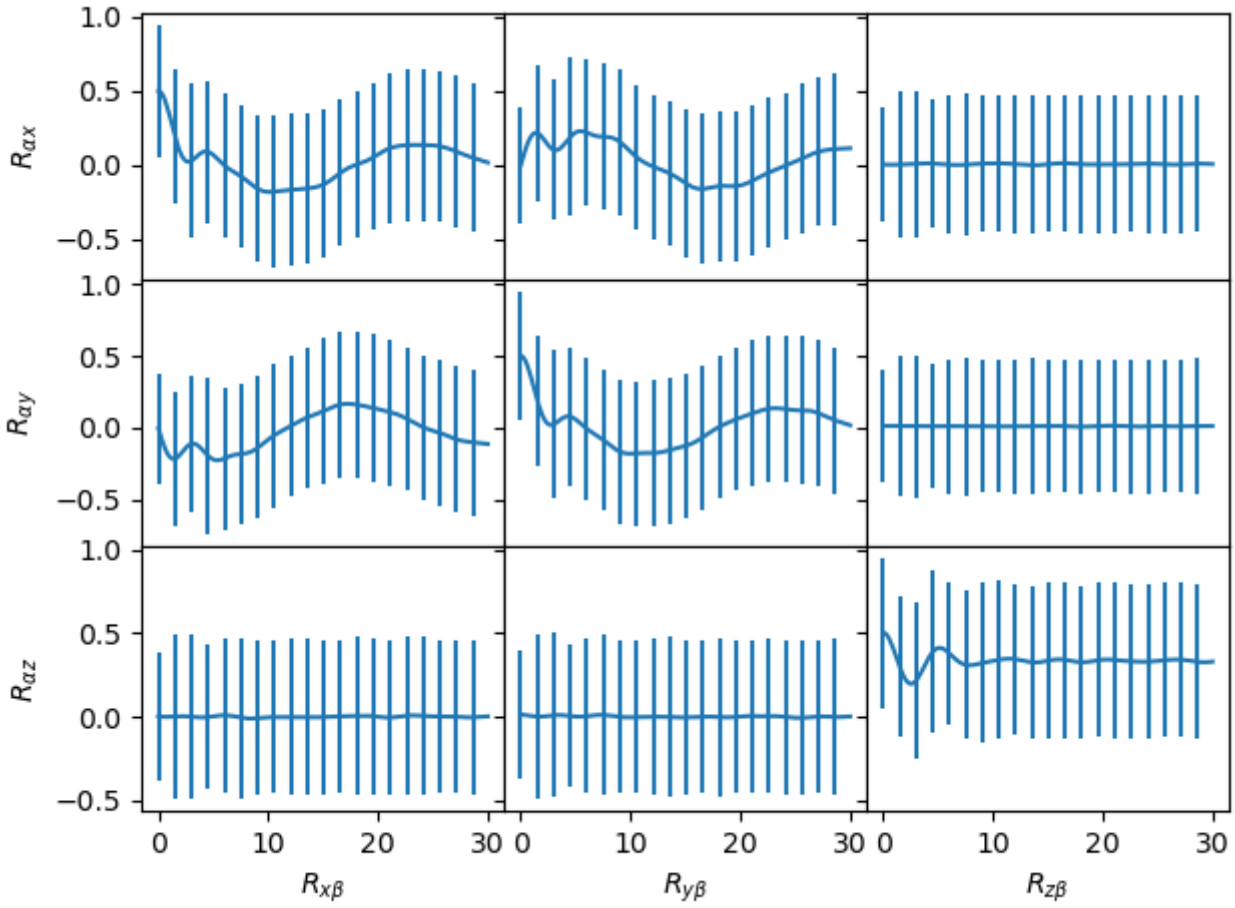
This is the result for 100 trajectories:



This is the result for 1000 trajectories:



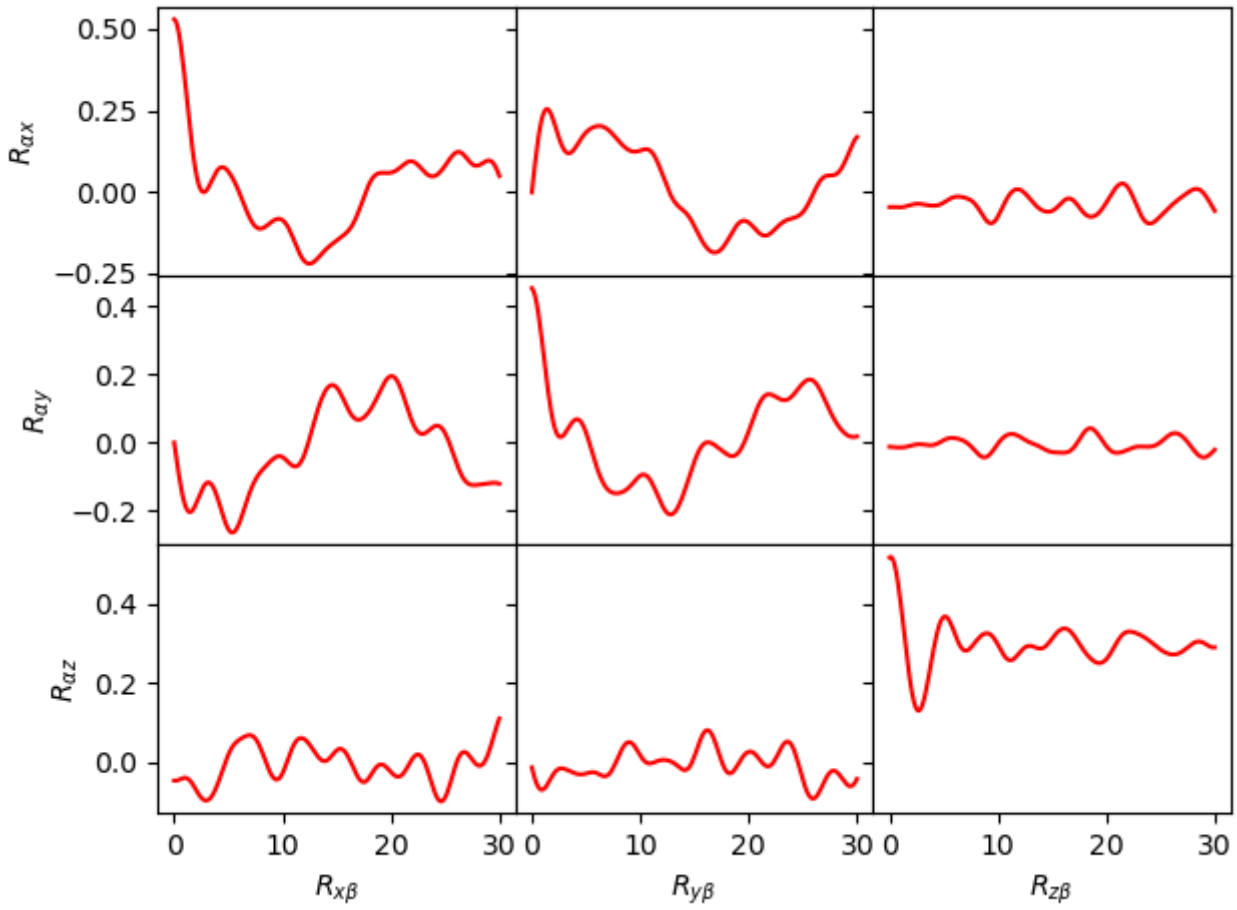
This is the result for 10000 trajectories:



As can be seen above this did not work how I intended it to work as looking at it from a more critical perspective I can see that the different trajectories will produce an equal standard deviation among them no matter how many there are unless the number is very low.

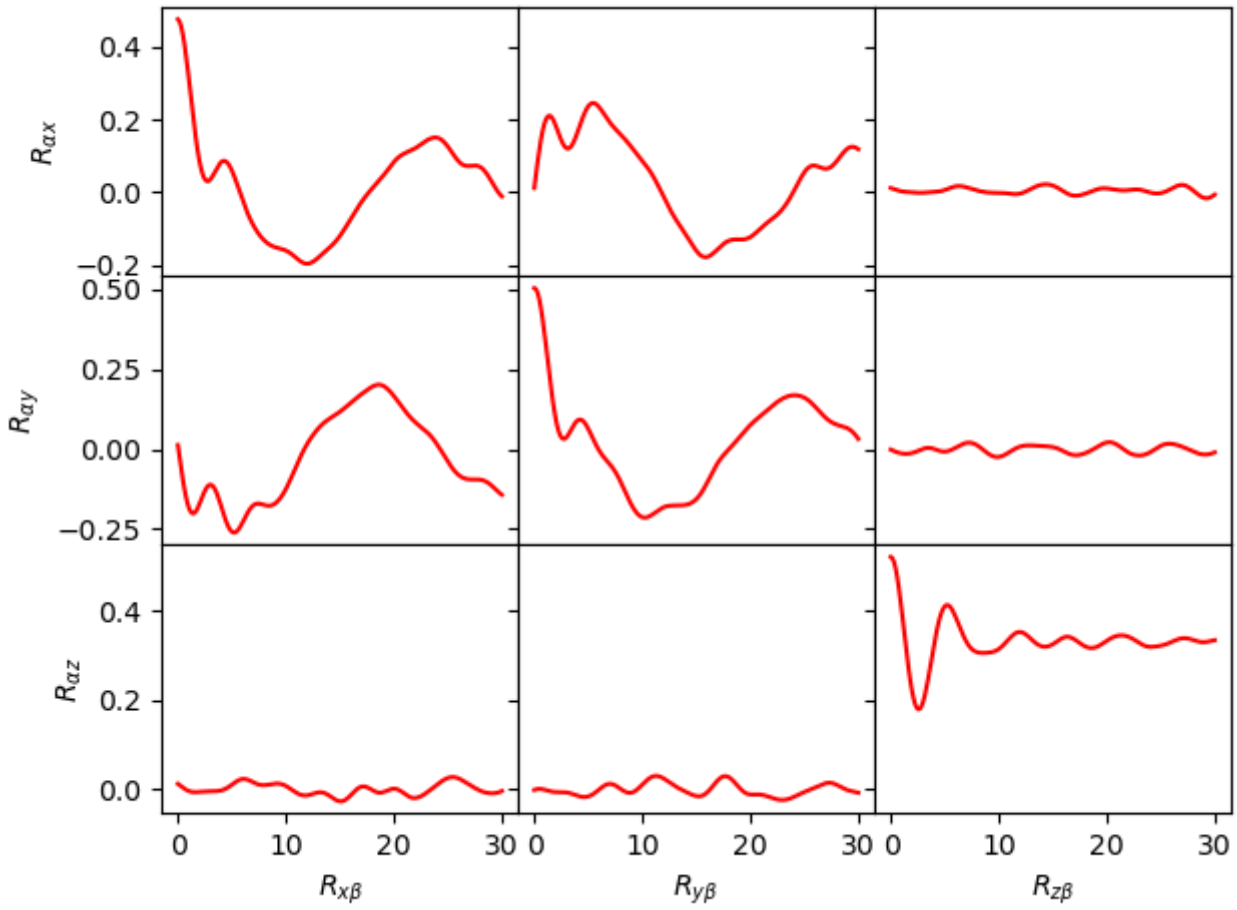
My next attempt was to calculate the standard deviation for all the points along one of the components of the tensor that should always be equal to 0.

This is the result for 100 trajectories:



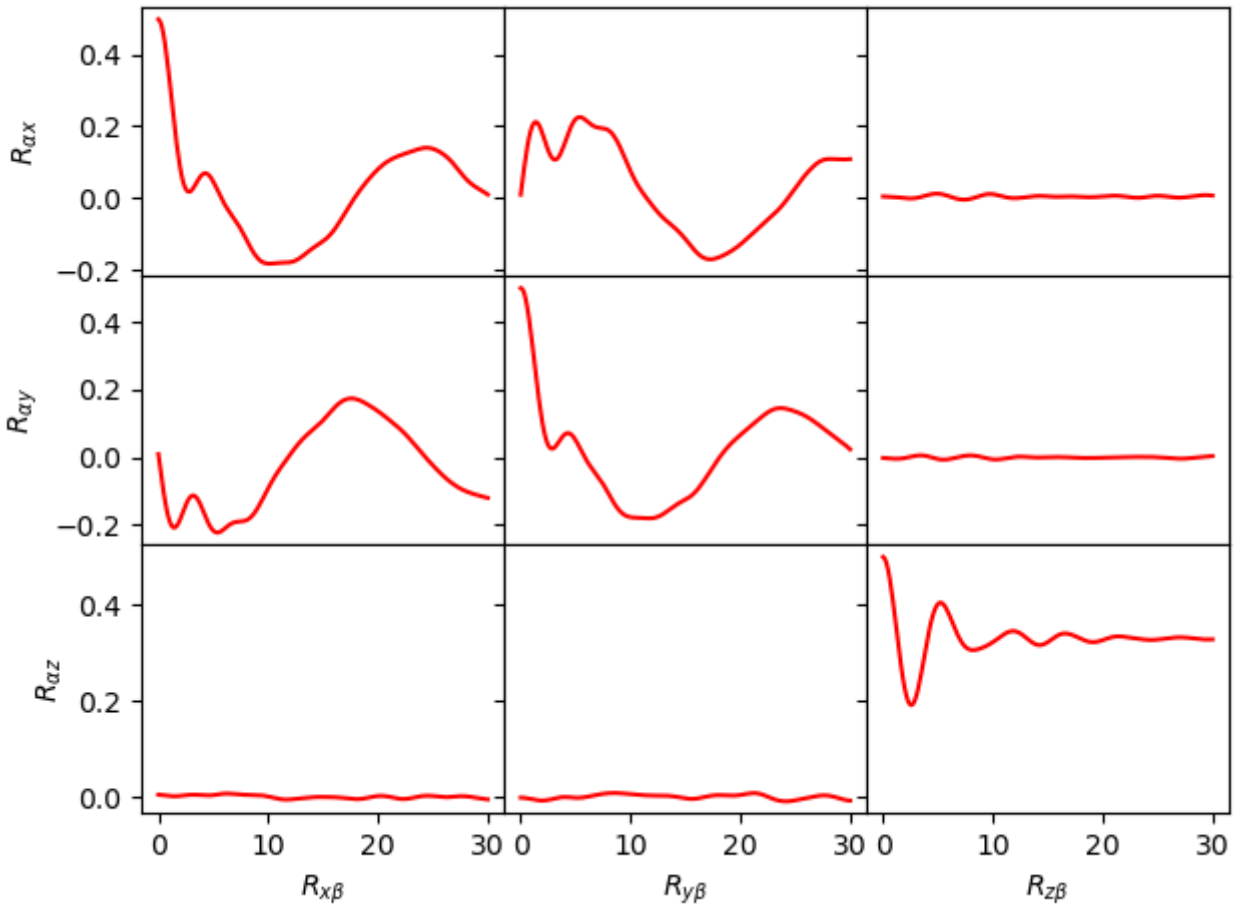
The standard deviation calculated for XZ component was 0.02919563730224244

This is the result for 1000 trajectories:



The standard deviation calculated for XZ component was 0.008528024594939284

This is the result for 10000 trajectories:



The standard deviation calculated for XZ component was 0.0034110684632754447

This appears to be a better way of measuring the error for now as can be seen with the error decreasing depending on the number of trajectories used in the calculation. However this approach will not work for when the magnetic field is in multiple directions.

References

[1] Hore P. J. and Mouritsen Henrik, The Radical-Pair Mechanism of Magnetoreception, , vol. 45, number 1, pp. 299--344, 2016. online (https://doi.org/10.1146/annurev-biophys-032116-094545)

[2] Rodgers Christopher T. and Hore P. J., Chemical magnetoreception in birds: The radical pair mechanism, , vol. 106, number 2, pp. 353--360, 2009. online (https://www.pnas.org/content/106/2/353)

[3] Manolopoulos David E. and Hore P. J., An improved semiclassical theory of radical pair recombination reactions, , vol. 139, number 12, pp. 124106, 2013. online (http://arxiv.org/abs/1407.2139)

[4] Eric W. Weisstein, Sphere Point Picking, . online (https://mathworld.wolfram.com/SpherePointPicking.html)

[5] , python - how to speed up a vector cross product calculation, . online (https://stackoverflow.com/questions/20908754/how-to-speed-up-a-vector-cross-product-calculation)