Fraser Birks, William Allison

Simulation methods for molecular beam scattering

MoBSTer: Molecular Beam Simulation using Trajectories

September 23, 2020

Contents

Part I
Methods

Chapter 1
Introduction

Abstract Successful apparatus design requires many components to work together in order to optimise performance and deliver useful signal. The components of an apparatus are often designed in isolation, often with assumptions about apparatus symmetry that may not be achieved in the laboratory. Here, we describe a trajectory-based framework for simulation of a complete molecular-beam apparatus including spin-propagation through magnetic fields. The framework, MoBSTer, is written within a MATLAB environment and uses Monte-Carlo methods to generate and propagate trajectories through the various components between beam-source and beam-detector. Part I provides an outline of the methods used and gives a "user-guide". Part II describes the characteristics and performance of individual components.

1.1 Get started tutorial

This section is a brief tutorial - written for any user who is using MoBSTer for the first time.

1.1.1 Downloading MoBSTer

You can find the latest version of MoBSTer on GitHub (https://github.com/FraserBirksStudent/MoBSTer). You will need to have the latest version of Matlab installed on you device. Once you have downloaded and unzipped the files, launch Matlab and navigate to the MoBSTer folder. Right click on the folder and hit 'add folders and sub folders to path'.

Congratulations! All the MoBSTer components, accessory functions, templates and examples are now accessible within your Matlab environment.

1.1.2 Your first simulation

The first step in any simulation is to launch the MoBSTer GUI. In the Matlab console window, type the command 'appdesigner'. Once the Matlab app designer has launched, click open and navigate to mobsterUI.mlapp - then hit run. Figure 1.1 shows the blank ComponentParameters tab.

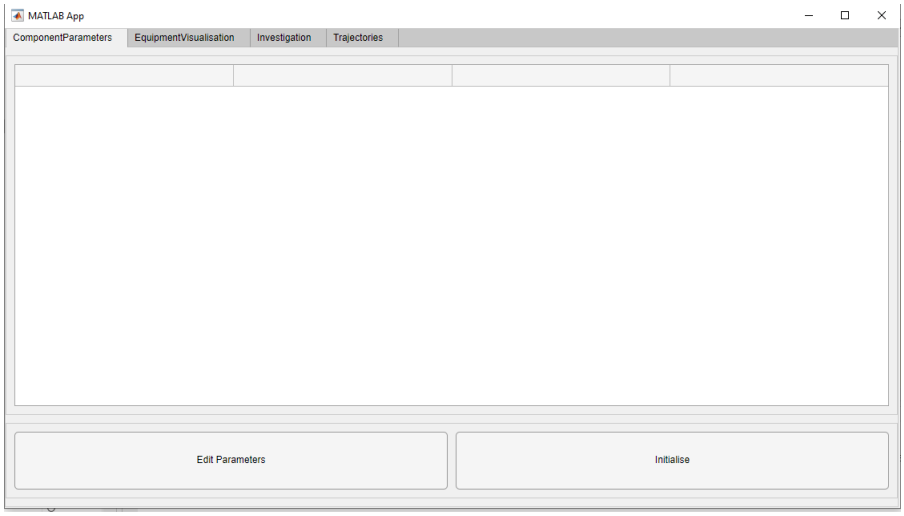


Fig. 1.1 The MoBSTerUI ComponentParameters tab. The edit parameters button opens the initialise.m Matlab file, and the initialise button loads the machine data specified in this file to the table.

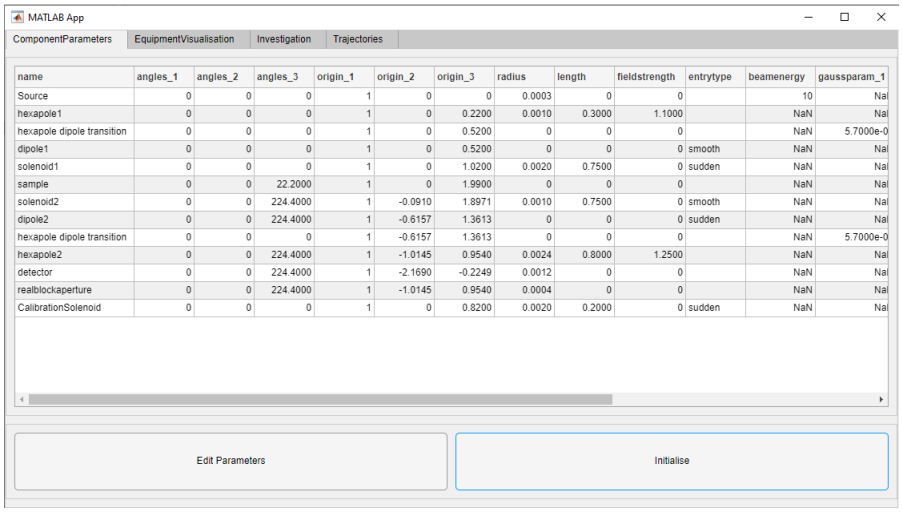
You'll notice on this opening screen that there are two buttons- Edit Parameters and initialise. Click on 'Edit Parameters'. This should open the Matlab file 'initialise.m'. This may look a little intimidating, but it's actually very simple- what's held in this file is the information about each instrument - it's location, angle of orientation, it's length, the strength of the magnetic field, as well as many other parameters. These parameters are all saved to a Matlab data structure which is passed to the simulation upon initialisation.

When you create your own simulation, you will need to enter the correct parameters for each instrument in this file- see section 2.1 of this manual for more information about initialising instruments correctly, including examples.

By default, the initialise file contains all the information about each instrument in the Cambridge helium spin echo machine [1]. This is the data you'll be using for your first simulation.

Head back to the GUI and press 'initialise'. What you should see is shown in

Figure 1.2 - all the data contained within the initialise function in a convenient and easy to digest table format.



name	angles_1	angles_2	angles_3	origin_1	origin_2	origin_3	radius	length	fieldstrength	entrytype	beamenergy	gaussparam_1
Source	0	0	0	1	0	0	0.0003	0	0		10	Nal
hexapole1	0	0	0	1	0	0.2200	0.0010	0.3000	1.1000		NaN	Nal
hexapole dipole transition	0	0	0	1	0	0.5200	0	0	0		NaN	5.7000e-0
dipole1	0	0	0	1	0	0.5200	0	0	0	smooth	NaN	Nal
solenoid1	0	0	0	1	0	1.0200	0.0020	0.7500	0	sudden	NaN	Nal
sample	0	0	22.2000	1	0	1.9900	0	0	0		NaN	Nal
solenoid2	0	0	224.4000	1	-0.0910	1.8971	0.0010	0.7500	0	smooth	NaN	Nal
dipole2	0	0	224.4000	1	-0.6157	1.3613	0	0	0	sudden	NaN	Nal
hexapole dipole transition	0	0	0	1	-0.6157	1.3613	0	0	0		NaN	5.7000e-0
hexapole2	0	0	224.4000	1	-1.0145	0.9540	0.0024	0.8000	1.2500		NaN	Nal
detector	0	0	224.4000	1	-2.1690	-0.2249	0.0012	0	0		NaN	Nal
realblockaperture	0	0	224.4000	1	-1.0145	0.9540	0.0004	0	0		NaN	Nal
CalibrationSolenoid	0	0	0	1	0	0.8200	0.0020	0.2000	0	sudden	NaN	Nal

Fig. 1.2 The machine data loaded into the simulation. In this form, it is easy for a user to check if it is as expected.

Figure 1.3 displays the next tab - which contains a 3D plot of the instrument configuration. When running a simulation it is important to check this to see if it matches what you expect - the majority of bugs in a simulation will be due to incorrect instrument placement and orientation!

Now, you've checked the machine is set up correctly, you're ready to run the first simulation! There are well commented example simulations provided- which are there to both help users learn the syntax and provide a platform that users can build on with their own simulations. The most basic possible simulation is just a single propagation through the machine. The file 'cambridgesinglerunexample.m' in the examples folder contains exactly this. Open this, select the first section of code and hit 'run and advance' to run the code instrument by instrument. In the Workspace you can notice the particles and trajectories structure changing size after every instrument - and if you view these structures in more detail you'll see how the data in the structures is changing during propagation.

A powerful element of MoBSTer is its ability to display the path taken by each successful molecule in the beam- this is a quick way to check for errors. Re-open the MoBSTer UI and press 'Plot Trajectories'. Figure 1.4 shows an example of what should be found on the next tab; a 3D plot of up to 1000 trajectories that represent the journey of the particles moving through the machine.

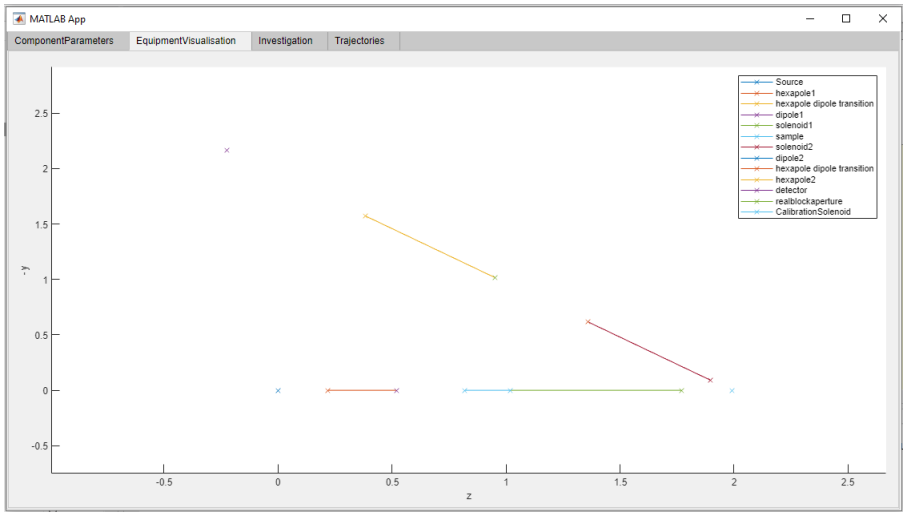


Fig. 1.3 A graphical representation of the machine contained within the initialise file. By default, this matches the Cambridge set-up.

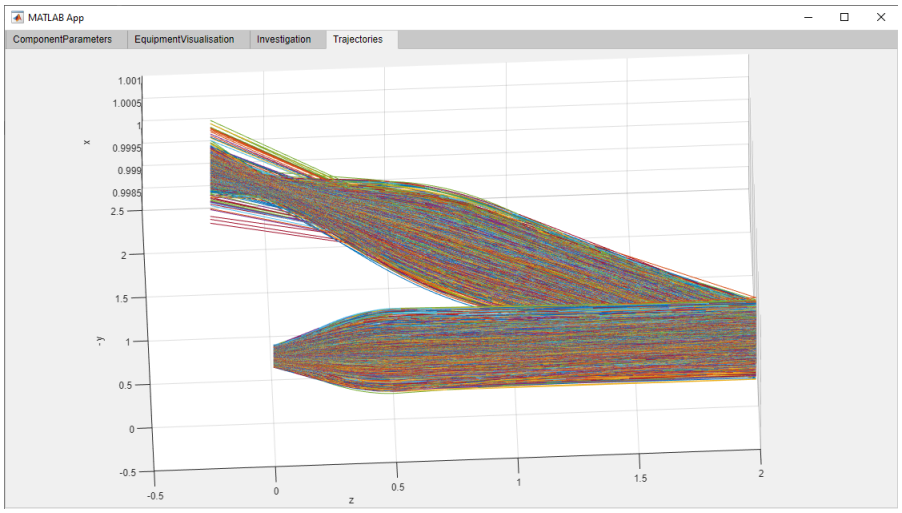


Fig. 1.4 A plot showing the path of up to 1000 trajectories through the machine - check there are no discontinuities.

Feel free to explore and read the other more complex examples - these are good showcases of how powerful the MoBSTer framework can be, and what sort of simulations it is possible to create.

You've now seen what MoBSTer can do - and I'm sure you're eager to simulate and investigate something yourself- perhaps you'd like to create an instrument and test it in the context of a whole machine? Or maybe you'd like to tilt a component through a range of angles and see how it changes the beam intensity or polarisation?

At this point, there are two options. Does the investigation you are planning involve adding instruments/ interacting with particles and trajectories in a new way that will require more functions? - if so, you are an 'advanced user' and it is highly recommended you read the entire manual in order to learn exactly how MoBSTer works and how to fully interface with the framework. However, if you simply want to manipulate and adjust the parameters of instruments already created, and subsequently don't need to edit or add any code apart from the top level programs, you should be set to go after reading the following sections:

- Section 1.2 on the syntax of the top level simulation file.
- Section 2.1 on coordinate systems and placing and orienting components.
- Section 3.1 on the structures used, and how to access the data within them.
- Section 3.3 on the data stored on each particle within the data structures
- Section 4.1 on the form of spin data in MoBSTer.
- Section 4.4 on how to measure the beam polarisation using MoBSTer. (This can also be seen in the examples provided.)

Additionally, it is recommended that all users refer to the section on each instrument function when creating a simulation in order to make sure the correct parameters are passed to each instrument.

1.2 The syntax of the top-level simulation program

The following details the syntax of the top level simulation program the user must follow. This syntax is very simple, and can be seen in any of the examples provided.

1.2.1 Beginning a simulation

All simulations should start with the same lines of code, that being those which clear the Workspace, and initialise the components:

```
clear
close all
```

```
N = 10000; %The number of particles, can be changed
[parameters,component] = initialise(); %initialise the components
```

After these lines of code, the data structures which store the simulation output should be initialised, for example, if a simulation was to store data on the polarisation of the beam each run, an initialisation of the the storage vector might look like:

```
pvector = zeros(601,1);
```

Doing this prevents dynamic memory allocation, increasing the efficiency of the simulation.

1.2.2 Running a simulation using components

A basic simulation is comprised of the instrument functions called one-by-one, in the order at which the particles would encounter them. For example, in the Cambridge set-up:

```
%% source
[particles,trajectories] = simplesourcev2(N,parameters(:,:,1),component(1).radius,
        100e-3,component(1).beamenergy);

%% hexapole 1
[particles,trajectories] = hexapole(particles,trajectories,parameters(:,:,2),
        component(2).radius,component(2).length,component(2).fieldstrength);

%% dipole 1
[particles,trajectories] = dipole(particles,trajectories,parameters(:,:,4),
        component(4).entrytype);

%% calibration solenoid
[particles,trajectories] = solenoid(particles,trajectories,parameters(:,:,13),
        component(13).radius,component(13).length,component(13).GyroMagneticRatio,
        component(13).fieldstrength,component(13).entrytype);

%% solenoid 1
[particles,trajectories] = solenoid(particles,trajectories,parameters(:,:,5),
        component(5).radius,component(5).length,component(5).GyroMagneticRatio,
        component(5).fieldstrength,component(5).entrytype);

%% sample
[particles,trajectories] = flatsample(particles,trajectories,parameters(:,:,6));

%% solenoid 2
[particles,trajectories] = solenoid(particles,trajectories,parameters(:,:,7),
        component(7).radius,component(7).length,component(7).GyroMagneticRatio,
        component(7).fieldstrength,component(7).entrytype);

%% dipole 2
[particles,trajectories] = dipole(particles,trajectories,parameters(:,:,8),
        component(8).entrytype);
```

```
%% hexapole 2
[particles,trajectories] = hexapole(particles,trajectories,parameters(:,:,10),
        component(10).radius,component(10).length,component(10).fieldstrength);

%% detector aperture
[particles,trajectories] = simpledetector(particles,trajectories,parameters(:,:,11),
        component(11).radius);
```

Using this format, a completely arbitrary machine can be constructed, built out of any instruments. Once the user has familiarised themselves with this basic simulation syntax (looking at the examples provided may help), they are already ready to utilise much of the power in the MoBSTer package.

In general, it is recommended that a maximum of 10^5 particles are used per individual simulation run, as any more than this may cause memory issues. More particles can be used if the trajectory structure is disabled, which is discussed in section 8.1.1.

1.2.3 Building more complex simulations

To create more complex simulations and investigations with the instruments provided, the user must iterate over at least one variable. The simplest way to do this is using a for loop, as can be seen in the examples. Each for loop iteration, the user should update whichever parameter they wish. If it is a parameter of one of the later components being changed, it is much faster to simulate the first part of the simulation once for a large number of particles, save the particles and trajectory structures, and then use them as the initial conditions to iterate over later components many times.

1.2.4 Harvesting data from simulations

The syntax of acquiring data from the structures is very simple. For example, to find the weight of the 1st particle, one just needs to type

```
particle(1).weight
```

Hence, to find the total weight of all the particles, the user can iterate over the structure, adding up all the weights. To see more detail about the structures, including the data they store, see chapter 3 and section 4.1. To see how to measure the beam polarisation from the beam data, see section 4.4 and examples 'cambridgespinechoexample.m' and 'CambridgeMachineTransmissionAnalysisExample.m'.

Once the user has familiarised themselves with the data available to read in MoBSTer

and it's propagation within simulations, they have complete free reign over what data to pull from a simulation and when.

Chapter 2
The Coordinate System and Controlling Components

2.1 Coordinates and the placing of components

Here we describe the coordinate systems used in the simulation and define their relationship. The main system is the "lab frame" specified by three Cartesian coordinates, (x, y, z). Each component in the simulation has an internal set of coordinates, (x', y', z') and, when placing a component in the lab frame, the user specifies the position, $P(x, y, z)$ and the orientation of the component (α, β, γ). Figure 4.5 illustrates the relationship of the coordinate systems. In the lab frame, unit vectors $\underline{\mathbf{n}}_{x'}(x, y, z)$ and $\underline{\mathbf{n}}_{z'}(x, y, z)$ specify the orientation of the component relative to the laboratory, while $P(x, y, z)$ places the origin of the component at the appropriate location in the lab frame. $\underline{\mathbf{n}}_{x'}(x, y, z)$ and $\underline{\mathbf{n}}_{z'}(x, y, z)$ are derived from rotating the lab frame $\underline{\mathbf{x}}$ and $\underline{\mathbf{z}}$ vectors by the specified (α, β, γ) angles. The definition of each angle (α, β, γ) matches the convention used in sample manipulators, which is described in the next section.

Within MoBSTer, the information about a components location and orientation is stored in a 3x3 matrix called the parameters matrix. Each parameters matrix is passed to the instrument when the instrument is called.

$$parameters = \begin{pmatrix} P(x, y, z) \\ \underline{\mathbf{n}}_{z'}(x, y, z) \\ \underline{\mathbf{n}}_{x'}(x, y, z) \end{pmatrix}$$

2.1.1 Description of (α, β, γ) angles

What each angle represents is described below- each rotation is clockwise when viewed with the corresponding vector pointing toward you.
α: A rotation clockwise α° about the component frame z' axis.

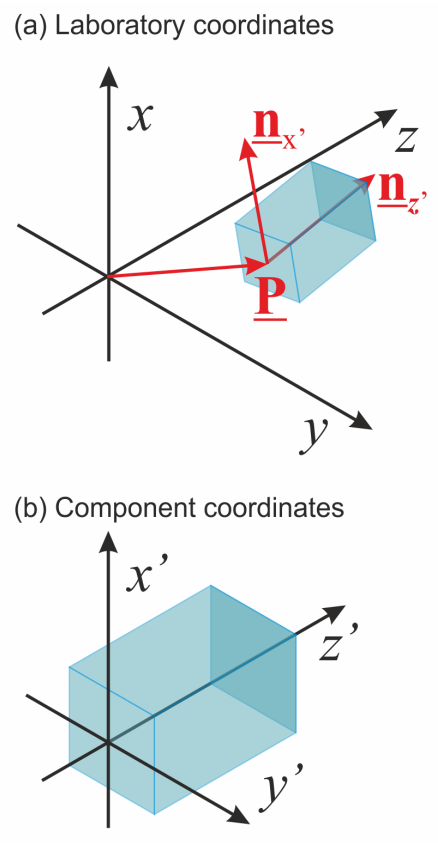


Fig. 2.1 Coordinates used in the trajectory simulations. (a) shows the laboratory coordinates, x, y, z, together with a component whose coordinate origin is at the point $\mathbf{P}(x, y, z)$. The alignment of the component is defined by the orthogonal unit vectors $\mathbf{n}_{x'}$ and $\mathbf{n}_{z'}$ corresponding to the x' and z' axes of the component. (b) illustrates the local coordinates, x', y', z' of a component.

β: A rotation clockwise β° about the lab frame y axis.
γ: A rotation clockwise γ° about the lab frame x axis.
Before any rotation, each instrument starts with $\mathbf{n}_{x'}$ and $\mathbf{n}_{z'}$ aligned with the lab frame x and z vectors. Note - the exception to this is the flatsample instrument, which by default has the $\mathbf{n}_{z'}$ vector aligned with the -z direction.

2.1.2 Placing a component in MoBSTer (example)

The following details an example of how to place and rotate components within MoBSTer, both when initialising or when running a simulation.

Figure 2.2 highlights the relevant lines the user must change to define the position and orientation of an instrument during initialisation. Note that if the instrument being placed has a non-zero length, the origin represents the start of the instrument, not the middle.

```
%HEXAPOLE 1
component(2).name = 'hexapole1'
component(2).angles = [0 0 0]
component(2).origin = [1 0 220e-3]
component(2).radius = 1e-3
component(2).length = 300e-3
component(2).fieldstrength = 1.1
[nz,nx] = angletonormalvector(component(2).angles);
parameters(:,:,2) = [component(2).origin; nz; nx]
```

Fig. 2.2 An example instrument initialisation. The lines underlined in red represent those which require user input. The first is the entry for the $[\alpha, \beta, \gamma]$ angles and the second is the entry for the lab frame $[x, y, z]$ of the component origin. The first line underlined in blue represents the function which converts these angles to $\mathbf{n}_{x'}$ and $\mathbf{n}_{z'}$ unit vectors. The second line underlined in blue represents the creation of the 3x3 parameters matrix as specified in section 2.1

At the end of this initialisation section, the user must define the parameters matrix for the instrument (as specified in section 2.1). Figure 2.2 has the two relevant lines of code to complete this underlined in blue, which must be present for each instrument. Once the user has placed all the components in the correct places using the initialise function, they can check the entire machine set-up is correct in 3 dimensions using the EquipmentVisualisation plot in the MoBSTer investigation UI. Figure 2.3 is an example of this for the Cambridge machine set up.

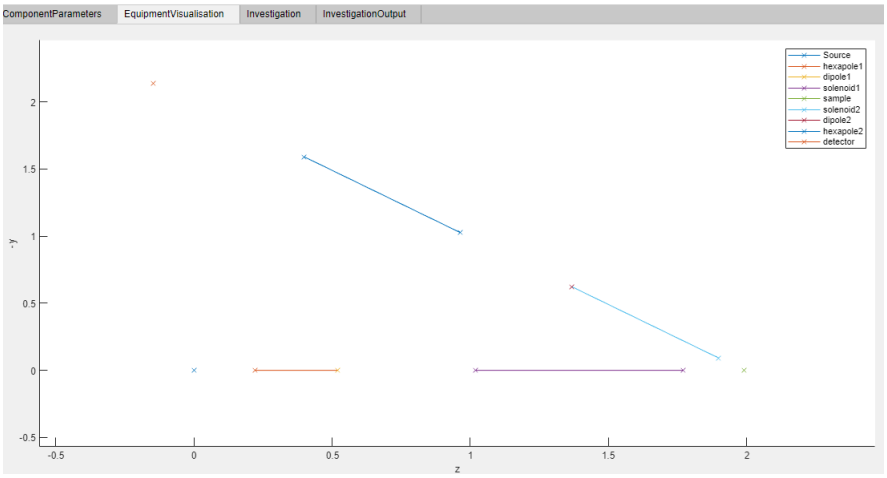


Fig. 2.3 An example machine layout mapped out in the MoBSTerUI. This is an easy way to check if everything is where it should be and oriented correctly.

When running a simulation, the user may change a components parameters however they wish. For example, if a user wishes to perform a specular scan of the sample through a small angular range they simply need to change the angle of the sample each iteration. As the user is changing the angular orientation of the instrument, they must make sure to re-define the parameters matrix. Figure 2.4 displays the syntax required in order to achieve this. Any component parameters can be altered during a simulation in the same way.

```
for run = 1:41
    component(6).angles = [0 0 22.4+0.005*run];
    [nz,nx] = angletonormalvector(component(6).angles);
    parameters(:,:,6) = [component(6).origin;nz;nx];
```

Fig. 2.4 An example of changing the parameters of an instrument during a simulation. Note that if one is changing the origin or angle of an instrument the parameters matrix must be updated with the corresponding lines of code.

2.2 Transforming velocity and position between coordinate frames

The following sections detail the methods used on each particle in MoBSTer to transform each position and velocity between reference frames as particles propagate through the simulation.

2.2.1 Transforming velocity

The relationship between the velocity vector in the lab frame $\mathbf{V}(\mathbf{x}, \mathbf{y}, \mathbf{z}) = (Vx, Vy, Vz)$ and the component frame $\mathbf{V}'(\mathbf{x}', \mathbf{y}', \mathbf{z}') = (Vx', Vy', Vz')$ is

$$\mathbf{V}(x, y, z) = (Vx'\underline{\mathbf{x}}' + Vy'\underline{\mathbf{y}}' + Vz'\underline{\mathbf{z}}'), \tag{2.1}$$

where $\underline{\mathbf{x}}'$ and $\underline{\mathbf{z}}'$ are the predefined $\underline{\mathbf{n}}'_{\mathbf{x}}$ and $\underline{\mathbf{n}}'_{\mathbf{z}}$ vectors respectively, and the unit vector, $\underline{\mathbf{y}}'$ is given by

$$\underline{\mathbf{y}}' = \underline{\mathbf{z}}' \times \underline{\mathbf{x}}'. \tag{2.2}$$

Expression 2.1 is equivalent to multiplying the component frame velocity $\mathbf{V}'(\mathbf{x}', \mathbf{y}', \mathbf{z}')$ with a rotation matrix R comprised of $\underline{\mathbf{x}}'$, $\underline{\mathbf{y}}'$ and $\underline{\mathbf{z}}'$ expressed in the lab frame:

$$R = \begin{pmatrix} x'(1) & y'(1) & z'(1) \\ x'(2) & y'(2) & z'(2) \\ x'(3) & y'(3) & z'(3) \end{pmatrix}$$

Hence, the transform 2.1 can be expressed as:

$$\mathbf{V}(x, y, z) = \underline{\underline{R}} * \mathbf{V}'(x', y', z') \tag{2.3}$$

R is orthogonal, meaning the inverse transform is simply:

$$\mathbf{V}'(x', y', z') = \underline{\underline{R^T}} * \mathbf{V}(x, y, z) \tag{2.4}$$

.

2.2.2 Transforming position between coordinate frames

This process is fundamentally the same as transforming velocity, apart from the extra step of adding or subtracting the component origin. If the component origin in the lab frame is $\mathbf{Q}(x, y, z) = (p, q, r)$, the position vector of the particle in the lab frame is $\mathbf{P}(x, y, z) = (a, b, c)$ and the position vector of the particle in the component frame is $\mathbf{P}'(x', y', z') = (a', b', c')$, then the transformation from the component frame to the lab frame is given by:

$$\mathbf{P}(x, y, z) = \mathbf{Q}(x, y, z) + \underline{\underline{R}} * \mathbf{P}'(x', y', z') \tag{2.5}$$

And the inverse transform from the component frame to the lab frame is given by

$$\mathbf{P}'(x', y', z') = \underline{\underline{R^T}} * (\mathbf{P}(x, y, z) - \mathbf{Q}(x, y, z)) \tag{2.6}$$

Chapter 3
Particles and Trajectories

3.1 How information about each particle is stored in MoBSTer

Figures 3.1 and 3.2 display the data structures MoBSTer uses to store simulation data.

- The particles structure: This stores information about each particle instantaneously as it's travelling through all the instruments in the machine. By default, this stores information on particle position, velocity, time, spin, weight and the B field the particle spin is defined with respect to. This structure is transformed between reference frames frequently as it propagates.
- The trajectories structure: This stores information about the history of each successful particle that has propagated through the machine. A data point is only stored in the trajectories vector when velocity changes. In this way, an overall trajectory can be constructed by joining together these points with straight lines, and there is not an excess of data. By default, this only stores position data. Each entry also has a Numberofentries value, which is an integer that can be read to find out how many non-zero entries there are in each storage vector for each particle. This is very useful for data analysis of the storage vectors as well as plotting out trajectories. This structure remains in the lab frame the entire time.

3.1.1 Accessing the data within these structures - an example

Figure 3.3 shows the very simple syntax needed to access data within these Matlab structures. The process is very simple - indeed this was one of the main reasons why structures were chosen to store data in MoBSTer.

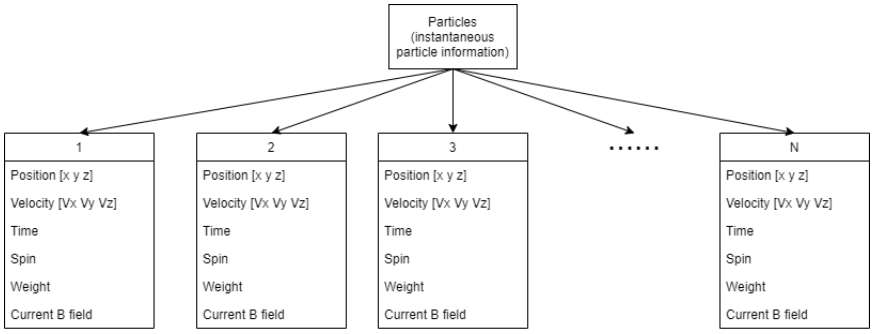


Fig. 3.1 A visualisation of the particles structure. The structure has N entries, (where N is the number of particles), with each entry corresponding to each particle. The subheadings within each category above represent the information stored by default about each particle. That being, Position, Velocity, Time, Spin, Weight and the current B field that the particle spin is defined with respect to. Each of these are described in the relevant manual section.

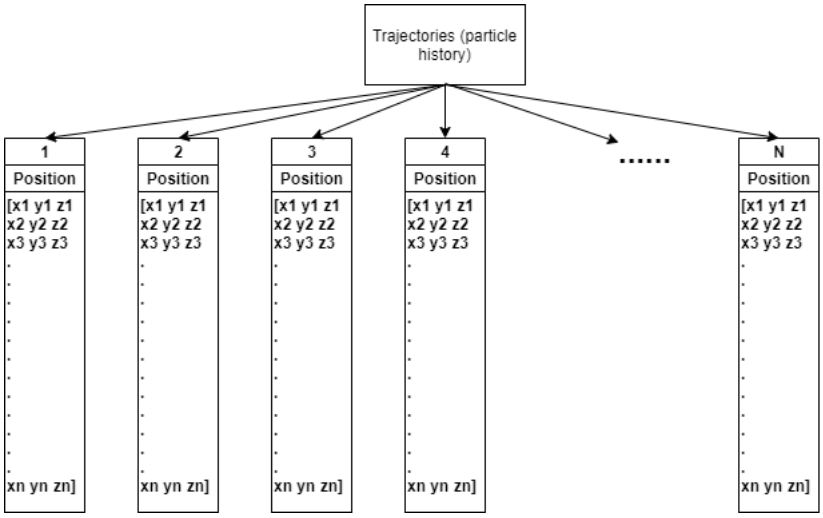


Fig. 3.2 A visualisation of the trajectories structure. The structure also has N entries, with one corresponding to each particle. Within each entry, information on the history of each particle is stored. By default, this is the history of the location each particle in the lab frame. There is also another entry for each particle (not shown) which stores the number of entries the storage vectors contain.

```
>> particles(10).spin

ans =

   0.9793 + 0.0488i   0.0000 + 0.0000i

>> particles(10).weight

ans =

    0.9615

>> particles(10).position

ans =

    1.0000   -2.1688   -0.2250
```

Fig. 3.3 The syntax required to access data within Matlab structures. This is some of the data of the 10th successful particle in a simulation.

3.2 The creation, destruction, and multiplication of data within structures

In MoBSTer, the number of particles and hence data entries within the structures are changing constantly as propagation occurs through instruments. Outlined below are the three cases in which the structures can change size in MoBSTer, with focus on the methods which MoBSTer uses to decrease simulation run-time when handling these huge structures.

3.2.1 Creation

Within the source function, the empty data structures are pre-initialised and the data of each particle is imported or created and assigned to the correct locations in the data structures. This includes initial position, velocity and spin vectors. Pre-initialising the data structures means that there is no dynamic memory allocation and results in the software running far faster. To see more about writing a source function and the methods used by the default source instrument, see the 'source' section within Part II (section 6.1) of this manual.

3.2.2 Destruction

For some instruments, for example a simple circular aperture, there is a spatial condition particles must satisfy in order to propagate through. For instruments like these, once all particles have been propagated into the $z = 0$ plane in the instrument frame, a check must be made for each particle to see if the condition to pass through is satisfied. For all those particles which fail this check, the data entries corresponding to this particle in both the particles and trajectories structure are removed. To reduce computation time, this is done by creating new smaller structures and copying across all the data of the particles that successfully pass through. Using this method, the memory is cleared of all trajectories that are unsuccessful and no computation is done further down the line on particles and trajectories which could not exist in the machine as they should have already been filtered out.

3.2.3 Multiplication

For instruments with strong inhomogenous magnetic field gradients, the overall beam wavefunction splits into highly localised wavepackets, each taking a classical trajectory. In MoBSTer this is approximated by splitting each particle into two as it enters the instrument. For more information on this, see section 4.2.1. This is handled by pre-initialising new structures which are twice as big, and then duplicating every particle in the old structures such that the new structures contain duplicate pairs of particles and trajectories. The new particles structure is then iterated over and the correct spin is assigned to each new particle, with each particle then being correctly propagated through the instrument. To see an example of this, see section 6.3.1 in Part II of this manual on the Hexapole component.

3.3 Stored data on each particle

Here we describe in suitable detail the form, type and use of the data stored in the structures for each particle (apart from the data defining spin- which is covered in section 4.)

3.3.1 Positon

The position of each particle is given by a 1-by-3 vector defining the $[x, y, z]$ of that particles position within the relevant frame. The vector is transformed between frames using the relationship specified in section 2.2.

3.3.2 Velocity

The velocity of each particle is also given by a 1-by-3 vector and defining the $[Vx, Vy, Vz]$ of that particle within the relevant frame. The vector is transformed between frames using the relationship specified in section 2.2.

3.3.3 Time

The time of each particle is stored as a float, representing the duration that particle has taken to propagate from the source to that point. Hence, it is defined to be 0 at the source, and as the particles propagate through each instrument the time is updated independently for each particle.

3.3.4 Weight

The weight of a particle is a float between 0 and 1 which represents the probability of that particle being measured in that position. There are two circumstances in which the weight of a particle can be changed:

- When using an instrument which is designed to remove a fraction of the beam. Instead of simulating the interaction of the beam with this component, the total weight of each particle can just be lowered by that fraction. This saves time having to simulate the instrument. For example, if one wanted to create a filter which cut out 9/10 particles at random, all particles could be allowed to pass through just with a weight multiplied by a factor of 0.1
- When the wavefunction of a particle splits in a strong inhomogeneous magnetic field. The weights of the new particles must be updated to reflect the probability of finding the particle in each location. This is described in detail in section 4.2.1.

3.4 How a general instrument will treat the particles and trajectories structures

Figure 3.3 is a flowchart that describes the way a general instrument will interact with the different components of the particles and trajectories structure.

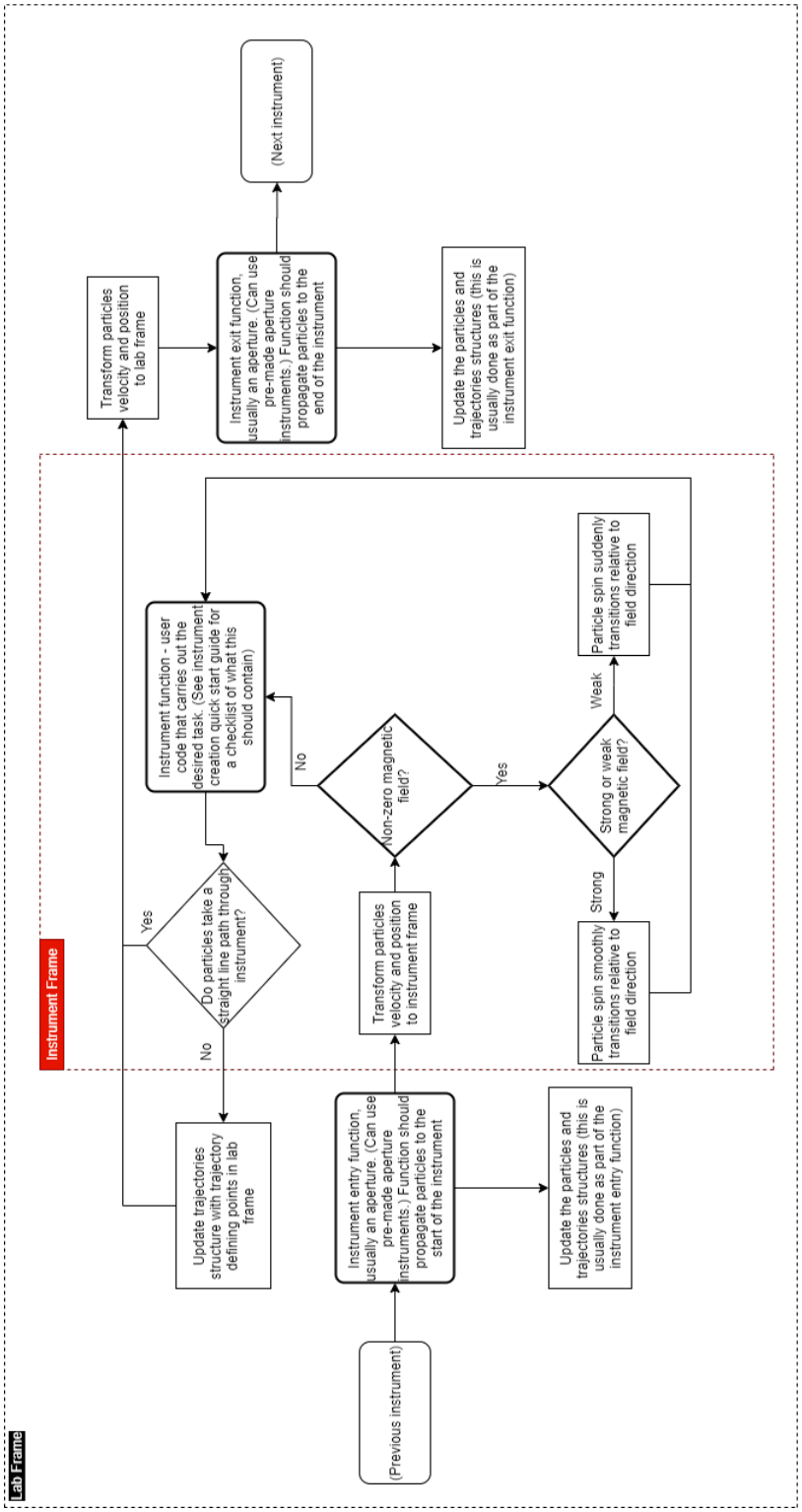


Fig. 3.4 Flowchart describing the form of a general instrument. One can see all the locations where the particles and trajectories structures need to be updated and transformed, and in which frames. The entry and exit functions will likely be simple pre-made aperture functions, which already propagate and store particle information, and remove those which do not satisfy the spatial condition as described in section 3.2.2. Part II of this manual has more information about the use and creation of the specific parts of each instrument.

Chapter 4
Spin

4.1 Introduction to spin in MoBSTer

Within MoBSTer, the spin component of each particle's wavefunction is described quantum mechanically using a vector of the complex coefficients of the eigenstates:

$$|\psi\rangle = C_\alpha |\alpha\rangle + C_\beta |\beta\rangle + C_\gamma |\gamma\rangle + ...$$

$$spinvector = \begin{pmatrix} C_\alpha \\ C_\beta \\ C_\gamma \\ .. \\ .. \end{pmatrix}$$

In MoBSTer v1.0, all the implementation of spin within the default instruments is limited to spin-1/2 - that is, only particles with two m_j states are supported within those instruments that interact with spin (the hexapole, dipole and solenoid).

However, by the expandable and versatile nature of MoBSTer, new versions of such instruments can easily be created which correctly interact with an arbitrary number of spin states. See section 8.3 of this manual for more information and guidance on which instruments and functions will need to be altered to allow for propagation of a greater number of spin states.

4.1.1 How the spin state is dependent on the external field direction

The eigenstates above are those defined by the basis of the S_z operator, where the z direction is the direction of the external magnetic field[3]. In MoBSTer, different particles could have spins defined relative to different local field directions. So, to preserve the generality of MoBSTer, it is important to store the information about

the lab frame direction of the field that a particle spin is defined by at any point. This is especially important when transforming spin between instruments (see sec 4.3), and is very useful when considering the implementation of numerical fields (see sec 8.2).

4.2 Spin propagation within instruments

Here, an approximation is made in order to combine the classical nature of the spatial information and the quantum mechanical nature of the spin information of each particle. If one considers the beam wave-function as a whole propagating through a strong inhomogenous magnetic field, it can be shown that the wave-function splits into highly localised propagating maxima [4]. Each maxima corresponds to the path taken by a classical beam of particles where each particle is in the same S_z eigenstate. If one considers the wavefunction of each particle splitting into these highly localised maxima, one can make an accurate semi-classical approximation that each particle splits at the start of the field into many sub-particles with an altered weight and spin which propagate following the classical paths of spin polarised particles.

4.2.1 Propagating spin in a strong in-homogeneous magnetic field

In this section, the spin-1/2 case for strong inhomogenous spin propagation is detailed - this can be easily generalised to higher spin states by following the same logic. As discussed above, when a particle enters a component containing a magnetic field within which it would take a different spatial path depending on it's spin state its wavefunction splits into two propagating regions of high probability.

Within MoBSTer v1.0, this is handled by splitting each incoming particle at the start of the instrument into distinct particles representing these propagating maxima. If the incoming particle has spin:

$$S_i = \begin{pmatrix} C_\alpha \\ C_\beta \end{pmatrix}$$

Then the outgoing particles have spins:

$$S_1, S_2 = \begin{pmatrix} C_\alpha \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ C_\beta \end{pmatrix}$$

The weights of these outgoing particles are adjusted accordingly; if the incoming particle has a weight W_i then the outgoing particles will have weights

$$W_1 = |C_\alpha|^2 * \frac{W_i}{|C_\alpha|^2 + |C_\beta|^2}$$

and

$$W_2 = |C_\beta|^2 * \frac{W_i}{|C_\alpha|^2 + |C_\beta|^2}$$

4.2.2 Propagating spin in a field perpendicular to the spin direction

In some components (like a solenoid) the field is designed to be in a separate direction to the classical spin vector. In fields such as these, the spin state evolves according to the solution of the time dependent Schrodinger equation.

By default in MoBSTer v1.0 this is implemented for spin-1/2. In this simple case, the solution to the time dependent Schrodinger equation is a procession of the spin vector around the field direction - which is usually called Larmor procession. To express this quantum mechanically, the time taken δt to propagate through a small uniform field element B is calculated, and then this small time is used to calculate the angle of rotation using the gyro magnetic ratio γ. The spin is then rotated using the rotation operator R_z given by the solution to the spin-1/2 time dependent Schrodinger equation.[3]

$$\phi_z = -\gamma |B| \delta t$$

$$R_z = \begin{pmatrix} exp(-i\frac{\phi_z}{2}) & 0 \\ 0 & exp(i\frac{\phi_z}{2}) \end{pmatrix}$$

4.3 Transforming spin at the entrance of components

In MoBSTer there are two types of spin transitions as particles transfer between component magnetic fields - smooth transitions which occur when entering strong fields and sudden transitions which occur when entering weak. The difference emerges from the fact that when entering stronger fields the rate of Larmor procession (the Larmor frequency) is higher - leading to the spin vector being able to move freely with the field in the lab frame. By design, these transitions are good approximations to true spin transitions between instruments in the machine.

4.3.1 Smooth transitions

A smooth transition takes place when a particle enters a strong field where the larmor frequency is high- this allows the spin state to rotate with the field- meaning

in the reference frame where the field is fixed (the frame within which C_α and C_β are defined) the spin is also fixed. Hence, in a smooth transition C_α and C_β remain constant. Within MoBSTer, a smooth transition means that only the lab frame direction of the field that each particle spin is defined by must be updated.

4.3.2 Sudden transitions

A sudden transition takes place when a particle enters a weak field, such that the larmor frequency is too low to allow for the spin state to rotate with the field. Hence, in the lab frame, the spin vector remains fixed whilst the field direction that the spin is defined by rotates. Arbitrary sudden spin transitions are described below, as well as their default implementation within MoBSTer v1.0.

4.3.2.1 Describing an arbitrary sudden spin transition

Consider a particle suddenly transitioning between an instrument where the lab frame field direction is B_{old} to an instrument where the field direction is B_{new}.

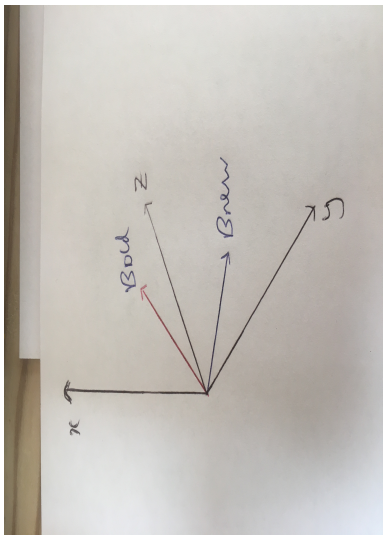


Fig. 4.1 A representation of the lab frame axes x, y, z. B_{old} and B_{new} represent the new and old field directions respectively.

It is easiest to describe a sudden spin transition in the frame where the spinor vector $\begin{pmatrix} C_\alpha \\ C_\beta \end{pmatrix}$ can be expressed classically. Figure 4.2 depicts this; the frame in which

the initial field direction B_{old} is along the z direction. We will call the frame where the z direction is aligned with the B field direction the 'spin frame'.

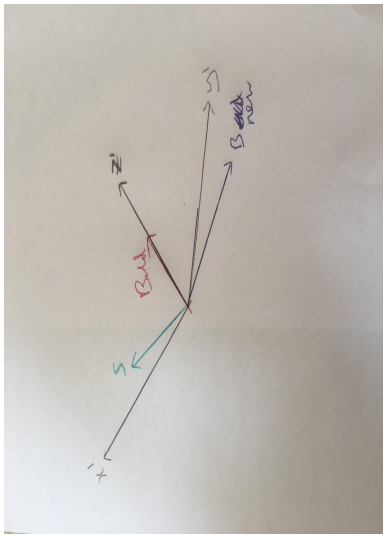


Fig. 4.2 A representation of the spin frame before the transition. z is in the direction of B_{old}. A classical depiction of the spin vector S is shown.

Now, consider the spin frame after the sudden transition, as shown in figure 4.3. In this new frame, the spin vector remains fixed but the z direction is now aligned with the new field direction.

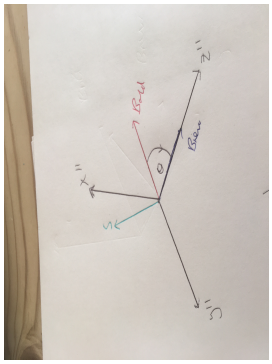


Fig. 4.3 A representation of the spin frame after the transition. The spin vector is in the same place but the z direction is now aligned with B_{new} field direction. The rotation angle is θ

Now, consider viewing the transition in the spin frame where the axis are fixed. Figure 4.4 depicts the motion of the spin vector in this reference frame: what is observed is the rotation of the spin by the negative of the angle θ defined as the angle between the new and old fields. This rotation occurs about the vector perpendicular to both the new and old field vectors.

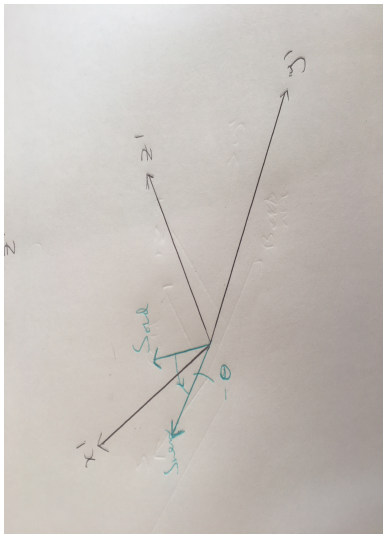


Fig. 4.4 A representation of the motion of the spin vector relative to the spin frame. The angle of rotation is $-\theta$.

4.3.2.2 Implementing this transition in MoBSTer

The following is a description of the implementation of the arbitrary sudden transition described above in MoBSTer v1.0, for spin-1/2. This method can easily be generalised to a greater number of spin states by replacing the rotation operators.

The first step after normalising the B vectors is to generate the transformation matrix which can operate on B_{new} to transform it into the spin frame (the frame which B_{old} is along z). First, the directions of the spin frame x and y axes must be derived from the direction of the magnetic field and the lab frame axes.

These $spinx$ and $spiny$ directions are defined to be equal to the lab frame directions when the spin vectors are generated, and hence can be derived from the lab frame directions at any point in the machine. This can be done using the idea that these axes are fixed in the 'spin frame' which is the frame where the field is always along z. One can imagine these x and y directions moving through the machine as if

on a rail that follows the magnetic field. The rail can pivot up and down (rotate about y) and left and right (rotate about x), but not twist (rotate about z). At any point in the machine the spinx and spiny directions can be found by rotating the lab frame axis about x and y such that the z direction matches the spin frame z direction. In this way, the spin frame x and y axes can always be found. In vector terms:

$$spiny = \frac{spinz \times x}{|spinz \times x|}$$

$$spinx = spiny \times spinz$$

This relationship breaks down when the spin frame z direction is equal to the lab frame x direction- in this case they can be found from:

$$spinx = \frac{y \times spinz}{|y \times spinz|}$$

$$spiny = spinz \times spinx$$

(Where x and y represent the lab frame axes).

Now, once the spin frame axes directions have been defined, the transformation matrix to act on B_{new} can be generated. Figure 4.5 shows both the theta angle and the normal vector direction.

$$Rspinframe' = \begin{pmatrix} spinx(1) & spinx(2) & spinx(3) \\ spiny(1) & spiny(2) & spiny(3) \\ spinz(1) & spinz(2) & spinz(3) \end{pmatrix}$$

B_{new} is then transformed from the lab frame into the spin frame.

$$B_{new(SF)} = Rspinframe' * B_{new}$$

Now, in the spin frame, the angle of theta between the two B vectors and the x-y plane normal vector which represents the axis of rotation are calculated.

$$\theta = acos(B_{new} \cdot B_{old})$$

$$\mathbf{n} = (B_{old} \times B_{new})$$

Using the components of this normal vector projected onto the x and y axes the phi values used in the spin rotation matrices can be generated (making sure to invert the sign from the angle defined between the magnetic field directions).

$$\phi_x = -\theta * n(1)$$

$$\phi_y = -\theta * n(2)$$

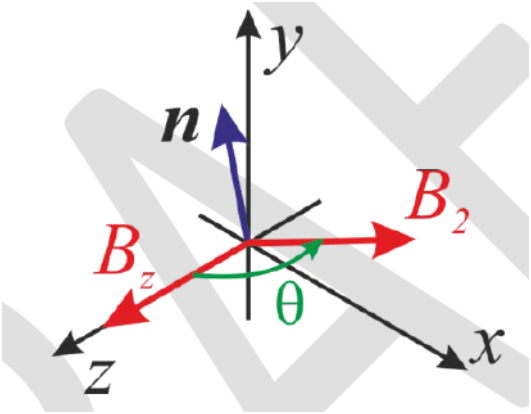


Fig. 4.5 The angle theta between the two B field vectors in the spin frame, as well as the $x-y$ plane rotation axis vector n

Finally, the spin-1/2 x and y rotation operators are generated which act on the spinor vector $\begin{pmatrix} C_\alpha \\ C_\beta \end{pmatrix}$.

$$R_x = \begin{pmatrix} cos(\frac{\phi_x}{2}) & -isin(\frac{\phi_x}{2}) \\ -isin(\frac{\phi_x}{2}) & cos(\frac{\phi_x}{2}) \end{pmatrix}$$

$$R_y = \begin{pmatrix} cos(\frac{\phi_y}{2}) & -sin(\frac{\phi_y}{2}) \\ sin(\frac{\phi_y}{2}) & cos(\frac{\phi_y}{2}) \end{pmatrix}$$

The resultant rotation matrix is then formed $R = R_x * R_y$ and the spinor vector is transformed.

$$\begin{pmatrix} C_{\alpha new} \\ C_{\beta new} \end{pmatrix} = R * \begin{pmatrix} C_\alpha \\ C_\beta \end{pmatrix}$$

4.4 Detecting Polarisation - for spin 1/2

Beam polarisation is a measure of the number of spin up particles (those focused by the analyser) against the number of spin down particles (those de-focused by the analyser). MoBSTer v1.0 can analyse the beam to determine this in two ways.

4.4.1 Determining true polarisation

After the analyser, the beam can be inspected and the total weight of all the spin up and spin down wave-packets can be added up. This can be used to find the true

polarisation of the beam exactly, which is only limited by how good the polariser is. Here, n_α represents the total weight of all spin up particles that have passed through the analyser and n_β all those which are spin down.

$$P = \frac{n_\alpha - n_\beta}{n_\alpha + n\beta}$$

4.4.2 Determining polarisation experimentally

In the lab, all that can be detected is the absolute number of particles that passes through the detector- and nothing about the spin of these particles. This can also be found in MoBSTer- the total number can be found by adding up all the weights of particles that remain after the detector aperture. In order to find the beam polarisation from this information, a calibration solenoid must be added after the polariser. A range of simulations should be run with the field in this solenoid being varied such that the polarisation of the beam is rotated by one cycle. The total weight from each run should be plotted against the field in the calibration solenoid.

The resulting graph should be a cosine curve - where the maximum of the curve represents the point where the beam polarisation is aligned with the dipole field direction, and the minimum where it is misaligned. One can use the curve fitting toolbox in Matlab to determine its parameters. From this, the polarisation that would be measured in the lab can be determined using the equation:

$$|P| = \frac{n_{max} - n_{min}}{n_{max} + n_{min}} \tag{4.1}$$

Where n_{max} and n_{min} are the minima and maxima of the cosine curve respectively.

If the analyser was perfect (it filtered out all spin down wavepackets) this would match the true polarisation. This is the method used within the provided 'CambridgeMachineTransmissionAnalysisExample.m'.

Note- in MoBSTer due to the fact that there are orders of magnitude less particles than there would be in a true experiment, the amount of shot noise is much greater- this means for readings where the polarisation is small it may require large numbers of particles and calibration solenoid measurements to fit a cosine curve to the data.

Part II
Components

Chapter 5
Creating instruments in MoBSTer

Abstract This chapter provides an overview of instrument creation with MoBSTer including a tutorial with a detailed example.

5.1 Creating an instrument: In steps

Support exists in MoBSTer to help a user both create and test new instrument ideas.

Figure 5.1 displays the Instrument Creation UI, which is a tool that one can use for developing and testing instruments. It can be found in the UsefulTools folder. An example of using this UI is shown in the next section.

The versatility of MoBSTer allows a user to have complete free reign when creating an instrument- provided they stick to the following basic rules:

- The instrument being created must take the particles and trajectories structures (see chapter 3) and the parameters matrix (see section 2.1) as input arguments.
- The output arguments must include the updated particles and trajectories structures in the lab frame.
- If the particles change direction, the positions of these changes in the lab frame should be stored in the trajectories structure - the addtotrajectories.m function already exists to make this easier.
- Within the instrument, the particles structure should be in the instrument frame.

The following bullet points provide some further advice on instrument creation:

- An instrument template is provided which contains sections of useful code commented out which one may need for a general instrument - one can access this both in the templates folder and through the instrument creation UI.
- Where possible, construct the new instrument out of simple instrument and non-instrument functions that have already been created - these are well-commented for ease of use and detailed in the following chapters.

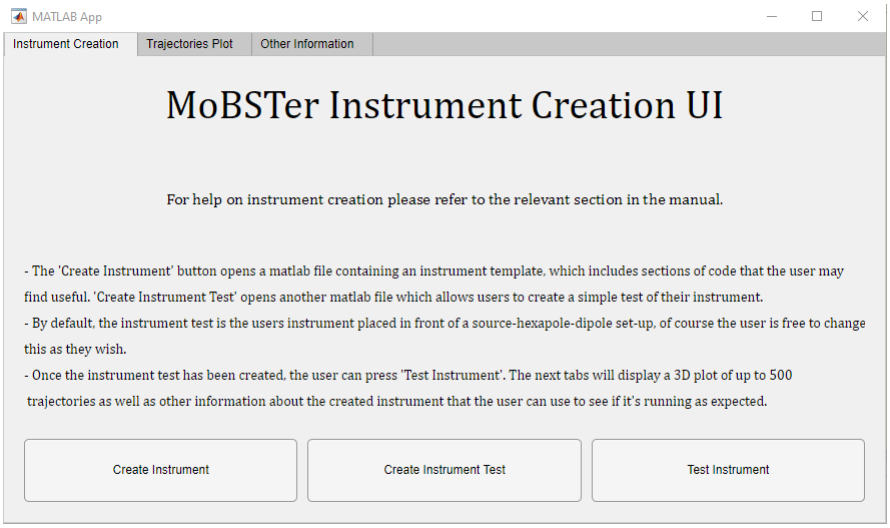


Fig. 5.1 The instrument creation UI, which can be found in the UsefulTools folder can be used for creating and testing new instruments.

- At the start of the instrument it is important to make sure you propagate all the particles forward from their previous position to the instrument origin plane. The propagate.m function allows this to be done easily. Much of the time, instruments will begin with an aperture. If this is the case, the already constructed aperture instrument can be used, which already contains a propagation step.
- If the particles undergo a sudden or smooth spin transition upon instrument entry (see section 4.3) make sure this is correctly implemented. For a smooth transition, one only needs to update the field direction each particle is defined relative to at the end of the instrument. For a sudden transition one needs to call the suddenspintransition.m function for each particle with the correct input arguments specified. Examples of these transitions can be seen in the default solenoid and dipole functions.
- If the number of particles and trajectories changes during propagation through the instrument through one of the ways detailed in section 3.2 make sure to pre-initialise new empty structures before copying the information across for speed. See the well commented removetrajectories.m and hexapole.m functions for examples of this.

A general instrument structure has already been shown in flowchart form in figure 3.3. This figure is also available in the templates folder.

5.2 Creating an instrument: An example

This section is a walkthrough of the creation and testing of a simple rectangular aperture instrument.

5.2.1 Creating the instrument function

First, the instrument template was opened, and all code which was not necessary was discarded - this instrument will not be interacting with the spin of the particles, just the spatial information. Next, the input arguments were determined. These should be particles, trajectories, the instrument parameters and the width and height of the rectangle. Figure 5.2 shows the result of this. The lines of code remaining are those related to the instrument transitions.

```
function [particles,trajectories] = rectangularaperture(particles,trajectories,param,width,height)

for i = 1:numel(particles) %convert to component frame
    [particles(i).velocity,particles(i).position] = frametransform(particles(i).velocity,particles(i).position,param);
end

%% Instrument function

% This region is your own code which defines what the instrument does.
% See the manual for guidance on how to update the data structures.


for i = 1:numel(particles)
    [particles(i).velocity,particles(i).position] = labtransform(particles(i).velocity,particles(i).position,param);
end


end
```

Fig. 5.2 The initial instrument template, containing just the frame transitions at the start and end.

The first step after the particles are converted into the component frame is to propagate them to the instrument origin frame. The propagate() function already exists to do this - so it can just be called. Figure 5.3 shows the relevant line of code.

```
particles = propagate(particles);
```

Fig. 5.3 Calling the propagate function in order to propagate all the particles into the components origin frame.

Now, a new function (removetrajecrectangle.m) must be created which is able to take the particles and either let through or block those which lie within a rectangle

around the instrument origin in the origin plane. This can then be re-purposed for both the normal aperture and a block aperture. Figure 5.4 shows the function well commented. The 'block' parameter can be set to 0 or 1 where 1 is a rectangular block and 0 is a rectangular hole.

```
function [newparticles, newtrajectories] = removetrajecrectangle(particles,trajectories,width,height,block)
%% FUNCTION DEFINITION
%this function finds the particles and corresponding trajectories for
%particles that are either within(seethrough) or not within(blocking)
%the specified width and height from the instrument axis,it
%then creates new particles and trajectories structures, adds them and then
%returns these back
%% code
rm= zeros(1,numel(particles)); %define an empty list of 0s which is the same length as the particles structure
for i = 1:(numel(particles))
    %Check if the particle lies in the correct region for each aperture,
    %and if it does then set the corresponding value in the rm matrix to
    %the number particle that matches the condition
    if abs(particles(i).position(1))<(width/2) && abs(particles(i).position(2))<(height/2)
        if block == 0
            rm(i) = i;
        end
    else
        if block == 1
            rm(i) = i;
        end
    end
end
%Remove all of the 0s from the rm vector, leaving a list of the positions
%of all the particles that pass through in the particles structure.
rm(rm==0)=[];
N = numel(rm);
%Initialise new structures for particles and trajectories
newparticles  = repmat(struct('position',zeros(1,3),'velocity',zeros(1,3),'spin',zeros(1,2),'weight',1,'time',0,'Bfield',zeros(1,3)), N, 1 );
newtrajectories = repmat(struct('position',zeros(100,3),'Numberofentries',1),N,1);
for i = 1:N
    %copy across the data corresponding to the particles which passed the
    %check
    newparticles(i) = particles(rm(i));
    newtrajectories(i) = trajectories(rm(i));
end
end
```

Fig. 5.4 The removetrajecrectangle.m function. This is the core body of the rectangular aperture functions. The first section of this is an if statement that finds all the particles within the allowed region based on the block parameter, and the second part copies those tagged to a new structure.

This function is broken up into 2 main sections, the first part being a for loop that iterates over all the particles in the structure and tags those which lie either inside or outside the correct area based on the block parameter. The second section then takes these tagged particles and adds them to a pre-initialised empty structure (following the method for particle destruction outlined in section 3.2.2). The function then returns the particles and trajectories structures.

This 'removetrajecrectangle' function is called after the propagation step with a block value of 0.

Finally, the current location of the particles must be added to the trajectory structure. This can be accomplished easily by calling the addtotrajectories.m function. Figure

5.5 displays the finished instrument function, which should now interact perfectly with MoBSTer.

```
function [particles,trajectories] = rectangularaperture(particles,trajectories,param,width,height)
%% function definition
%This is a rectangular aperture that operates the exact same as the
%circular aperture but instead of removing particles within a radius it
%removes particles outside of the x,y range defined by the rectangular
%aperture dimensions and origin.

%It operaterates generally the exact same, but calls the
%removetrajecrectangle.m function instead of the removetrajec.m function to
%remove the trajectories.

%it calls removetrajecrectangle with the block argument set to 0, as it is
%not blocking
for i = 1:numel(particles)
    [particles(i).velocity,particles(i).position] = frametransform(particles(i).velocity,particles(i).position,param);
end
particles = propagate(particles);
[particles, trajectories] = removetrajecrectangle(particles,trajectories,width,height,0);
trajectories = addtotrajectories(particles,trajectories,param);
for i = 1:numel(particles)
    [particles(i).velocity,particles(i).position] = labtransform(particles(i).velocity,particles(i).position,param);
end
end
```

Fig. 5.5 The finished code of the rectangular aperture instrument. Note that it is very simple and relies heavily on general functions which can be re-purposed for other instruments.

5.2.2 Testing the instrument function

The rectangular instrument function once created can be easily tested using the Instrument Creation UI. Figure 5.6 shows the default instrument test Matlab file which can be opened by pressing the 'Create Instrument Test' button - it is not complex, the aperture is just placed in front of a source-hexapole-dipole set up. Once the test has been written in this file to the users satisfaction, the user can press 'Test Instrument'.

The outputs of this test are shown in figures 5.7 and 5.8. Figure 5.9 shows the output of a test with the alpha input angle = 45. Data about the runtime and the number of particles passing through the instrument are also provided for the user. If the user wishes to test more information about the instrument, they can create their own custom simulation to do so.

```
%rectangular aperture
component(4).name = 'rectangular aperture'
component(4).angles = [0 0 0]
component(4).origin = [0 0 700e-3]
[nz,nx] = angletonormalvector(component(4).angles);
parameters(:,:,4) = [component(4).origin;nz;nx];
component(4).width = 1e-3
component(4).height = 2e-3

%% Run simulation
[particles,trajectories] = simplesourcev2(N,parameters(:,:,1),component(1).radius,100e-3,component(1).beamenergy);

[particles,trajectories] = hexapole(particles,trajectories,parameters(:,:,2),component(2).radius,component(2).length,component(2).fieldstrength

[particles,trajectories] = dipole(particles,trajectories,parameters(:,:,3),component(3).entrytype);

Nin = numel(particles);
tic
%YOUR INTSTRUMENT HERE
[particles,trajectories] = rectangularaperture(particles,trajectories,parameters(:,:,4),component(4).width,component(4).height)

runtime = toc
Nout = numel(particles);
```

Fig. 5.6 The test of the instrument. Note, the other instruments in the test are initialised above and are not shown in this figure.

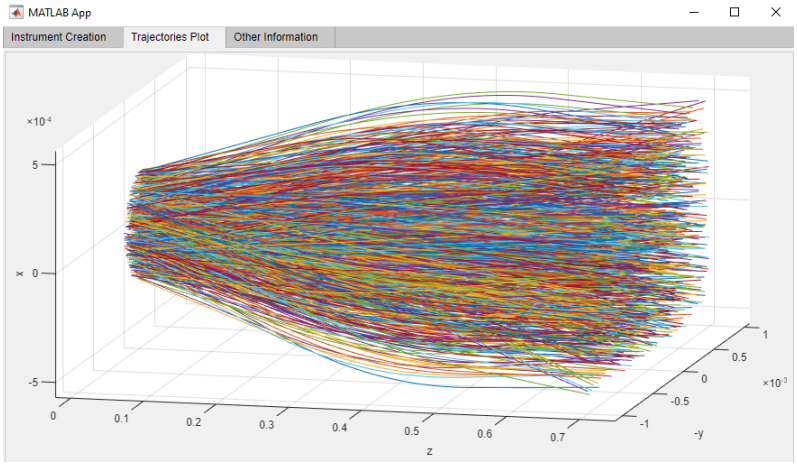


Fig. 5.7 The trajectory output diagram of the first test.

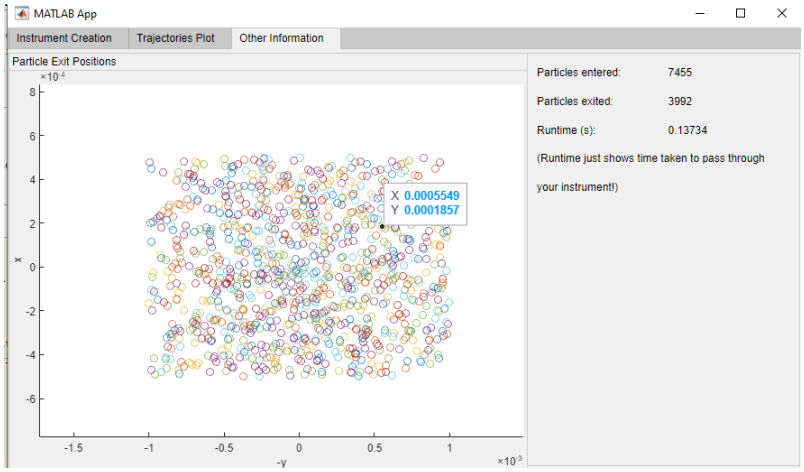


Fig. 5.8 The scatter output of the first test, showing the particle exit points. As well as this, the runtime and number of particles passing through are shown.

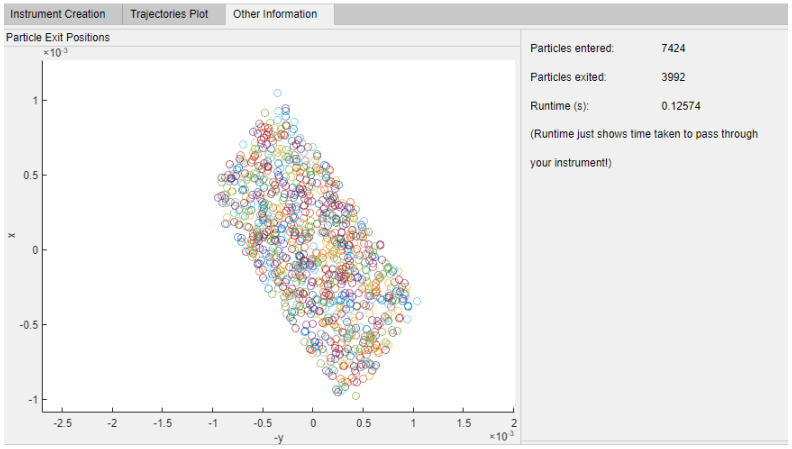


Fig. 5.9 The scatter output of the second test, with $\alpha = 45^o$, showing the particle exit points. As well as this, the runtime and number of particles passing through are shown.

Chapter 6
List of components in MoBSTer v1.0

Abstract We provide a library of default components for use with the MoBSTer v1.0 package. In the present chapter, each section is devoted to a particular component-class. Within each subsection, the function of an individual component is described in detail. The code of each component is not presented, but can be found easily by viewing the instrument files in the MoBSTer v1.0 install.

6.1 Beam sources

The beam-source is the first component in any simulation. Beam sources are the only components which do not take the particle and trajectory structures as inputs. Beam sources must contain a section that initialises and outputs the trajectory and particle structures of the correct form, with the particle structure including for each generated particle an initial spatial position, velocity, spin, weight and time. Each particle weight is initially set to 1 as it exits the source.

6.1.1 simplesourceV2(N,param, radius,zspacing,energy)

Input parameters: The number of particles generated(N), the 3x3 parameter matrix containing information about the position and orientation of the front of the source in the lab frame, the radius of both the internal sources apertures, the distance between these virtual apertures(zpacing) and the beam energy.
Output parameters: The particles structure (in the lab frame) and the trajectories structure.

SimplesourceV2 is a basic and fast source designed to be used when testing other components - it is an obvious approximation to a better source. It generates a simple beam using two circular apertures with defined radius separated by a distance

zspacing along the component frame z axis. Molecular paths are generated using two sets of connected randomly generated points on each aperture using the function RandPoints specified in sec 7.2.6. The position generated on the second aperture is the initial position of each particle and the generated trajectories represent the direction of the velocity vectors of each particle. Each path is then assigned a velocity magnitude from a Gaussian distribution with mean velocity specified by the beam energy and FWHM equal to 1/10 of this mean value. If the user wants a wider or narrower gaussian they need only to change the value of 10 specified in the code.

By default, the zspacing between the virtual apertures is 100mm, which provides an ample beam spread for the cambridge machine set-up. The user can change this to a greater number if they would like a narrower beam. A further key assumption made when initialising the trajectories structure is that the size of the position storage is only pre-initialised up to 100 points for each particle. This is to save memory, as any larger would not be necessary for most machines. If the user knows that the machine they are constructing will contain particle trajectories with more than 100 points (for example, if the machine contains numerical field propagation), they can increase this value.

The initial C_α and C_β spin states required to define the spin-1/2 spin states are then generated. To do this, two sets of random complex numbers of magnitude 1 are generated, and then scaled randomly but keeping the property that $|C_\alpha|^2 + |C_\beta|^2 = 1$. These spin states are defined in the lab frame, specifically defined with respect to a field in the [0 0 1] direction. Finally, the particle and trajectories structures are initialised, and the initial lab frame data added.

6.2 Beam blockers and filters

This section contains the components which act to modify only spatial shape of the overall beam.

6.2.1 aperture(particles,trajectories,param,radius)

Input parameters: The particles and trajectories structures, the 3x3 parameters matrix containing information about the position and orientation of the component, and the radius of the open space in the centre of the aperture.
Output parameters: The particles structure (in the lab frame) and the updated trajectories structure.
This component is the most simple circular aperture. It converts the particles into the component frame, and propagates them to the origin plane such that $z = 0$ for each particle using the propagate function defined in section 7.2.5. It then removes

any particles and trajectories that lie a distance greater than the radius away from the component frame z axis by calling the removetrajec.m function as described in section 7.2.7 with the block argument set to 0. The trajectories structure is then updated with the new particle positions using the addtotrajectories.m function described in section 7.2.1. The structures are then transformed back to the lab frame and returned.

6.2.2 blockaperture(particles,trajectories,param,radius)

Input parameters: The particles and trajectories structures, the 3x3 parameters matrix containing information about the position and orientation of the component, and the radius of the aperture.
Output parameters: The particles structure (in the lab frame) and the updated trajectories structure.
This component is a circular blocking aperture. It converts the particles into the component frame, and propagates them to the origin plane such that $z = 0$ for each particle using the propagate function defined in section 7.2.5. It then removes any particles and trajectories that lie a distance smaller than the radius away from the component frame z axis by calling the removetrajec.m function as described in section 7.2.7 with the block argument set to 1. The trajectories structure is then updated with the new particle positions using the addtotrajectories.m function described in section 7.2.1. The structures are then transformed back to the lab frame and returned.

6.2.3 rectangualaraperture(particles,trajectories,param,width,height)

Input parameters: The particles and trajectories structures, the 3x3 parameters matrix containing information about the position and orientation of the component, and the total width and height of the rectangular hole in the centre of the aperture.
Output parameters: The particles structure (in the lab frame) and the updated trajectories structure.
This component is a rectangular aperture. It converts the particles into the component frame, and propagates them to the origin plane such that $z = 0$ for each particle using the propagate function defined in section 7.2.5. It then removes any particles and trajectories that do not lie within both half the specified width and half the specified height of the aperture from the component frame z axis by calling the removetrajecrectangle.m function as described in section 7.2.8 with the block argument set to 0. The trajectories structure is then updated with the new particle positions using the addtotrajectories.m function described in section 7.2.1. The structures are then transformed back to the lab frame and returned.

6.2.4 rectangualarblockaperture(particles,trajectories,param,width,height)

Input parameters: The particles and trajectories structures, the 3x3 parameters matrix containing information about the position and orientation of the component, and the total width and height of the rectangular block.
Output parameters: The particles structure (in the lab frame) and the updated trajectories structure.
This component is a rectangular blocking aperture. It converts the particles into the component frame, and propagates them to the origin plane such that $z = 0$ for each particle using the propagate function defined in section 7.2.5. It then removes any particles and trajectories that lie within both half the specified width and half the specified height of the aperture from the component frame z axis by calling the removetrajecrectangle.m function as described in section 7.2.8 with the block argument set to 1. The trajectories structure is then updated with the new particle positions using the addtotrajectories.m function described in section 7.2.1. The structures are then transformed back to the lab frame and returned.

6.2.5 realblockaperture(particles,trajectories,param,radius,width,height)

Input parameters: The particles and trajectories structures, the 3x3 parameters matrix containing information about the position and orientation of the component, the radius of the circular block part of the aperture, and the total width and height of the rectangular block part of the aperture.
Output parameters: The particles structure (in the lab frame) and the updated trajectories structure.
This component is designed to simulate a realistic implementation of a circular blocking aperture, which would be suspended on the path of the beamline with a thin wire. It is a simple combination of the circular blocking aperture instrument specified in section 6.2.2 and the rectangular blocking aperture specified in section 6.2.4.

6.3 Magnetic field containing components

This section contains any components which alter the beam polarisation due to the presence of a magnetic field.

6.3.1 hexapole(particles,trajectories,param,radius,length,fieldstrength)

Input parameters: The particles and trajectories structures, the 3x3 parameters matrix containing information about the position and orientation of the component, the radius of the open space in the centre of the hexapole, the length of the hexapole and the maximum field strength in the hexapole (at the pole pieces).
Output parameters: The particles structure (in the lab frame) and the updated trajectories structure.
This component is used to propagate spin-1/2 particles through a perfect hexapole. The function of the hexapole is to polarise the beam of particles by manipulating the z-spin eigenstates as separate wave-packets as described in sections 3.2.3 and 4.2.1. The component contains two simple apertures at the start and end, separated by a distance equal to the length of the hexapole. Initially, each particle is propagated to the entrance aperture using the aperture function. Then, each particle is split into 2 separate trajectories which represent the propagating spatial maxima of the wavefunction (with each propagating maxima having a separate well defined z-spin eigenstate). The two split wave-packets are entirely in $|\alpha\rangle$ or $|\beta\rangle$ spin states and are assigned weights as described in section (4.2.1). New particles and trajectories structures which are twice the size are initialised to hold all the new particles.

6.3.1.1 Solving for the spatial motion of the wavepackets

The path of each wavepacket down the hexapole is then determined by solving the numerical differential equations of motion for a classical particle of spin up or spin down in a perfect hexapole field, with the force on each particle given by: [2]

$$\mathbf{F} = \pm\mu\nabla|B_{hex}| \tag{6.1}$$

With + corresponding to those in the $|\alpha\rangle$ state and − corresponding to those in the $|\beta\rangle$ state. In this instrument, the hexapole field is assumed to be perfect:[2]

$$\mathbf{B_{hex}} = 3C_3 r(cos(3\phi)\hat{\mathbf{r}} - sin(3\phi)\hat{\phi}) \tag{6.2}$$

Where the location of the particle in polar coordinates relative to the hexapole axis is (r, ϕ),and C_3 is a constant which is determined from the maximum field strength when r = radius.

The motion of each particle is then solved numerically, with an error tolerance of 1% and 20 time steps. These can be changed to make the hexapole more accurate, but a greater accuracy leads to much larger computing time and a minimal improvement in results. Figure 6.1 shows the divergence of the same trajectory after a hexapole of length 800mm with step sizes specified in the legend: a step size of $10e^{-3}$m is 80 steps and $100e^{-3}$m is only 8 steps. The improvement in accuracy is only on the order of 10nm.

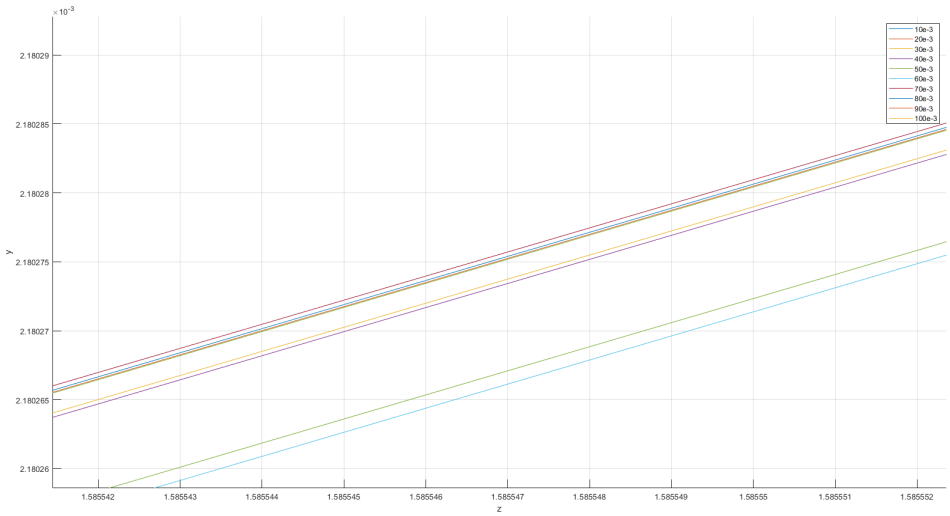


Fig. 6.1 The difference between a single trajectory with a different number of timesteps after a 800mm hexapole, with the number of timesteps being altered based on the length of each step. $10e^{-3}$m is 80 steps and $100e^{-3}$m is only 8 steps. The difference between the trajectories is only on the order of 10nm- which is negligible.

Once the component frame x,y and z positions over time for each propagating wavepacket are generated, they are converted into the lab frame and saved to the new trajectories structure. The new particles structure is then also updated with the final exit positions, velocities and spin of the wavepackets and converted into the lab frame. Finally, the exit aperture function is called which acts to only allow through those wavepackets in the centre of the hexapole.

6.3.2 dipole(particles,trajectories,param,entrytransition)

Input parameters: The particles and trajectories structures, the 3x3 parameters matrix containing information about the position and orientation of the component, and the type of entry spin transition.
Output parameters: The particles structure (in the lab frame) and the trajectories structure.
The dipole function is very simple and essentially just acts as a transition such that spin states are defined relative to a new field direction. The direction of the field in the dipole functions is along the component $\mathbf{n_x}$. The transition from the previous component can either be smooth (in the case of the first dipole) or sudden (in the case of the second dipole). These transitions are outlined in section 4.3.

For both entry transitions, the B field direction that the spin state of each parti-

cle is defined relative to is changed to $\mathbf{n_x}$. If the entry transition is sudden, the suddenspintransition function is called for each particle - defined in section 7.2.9.

6.3.3 solenoid(particles,trajectories,param,radius,length,gmr,Bstrength,entrytransition)

Input parameters: The particles and trajectories structures, the 3x3 parameters matrix containing information about the position and orientation of the component, the length of the instrument, the gyro-magnetic ratio of particles inside the instrument, the strength of the uniform field within the instrument and the type of the entry transition.
Output parameters: The particles structure (in the lab frame) and the trajectories structure.

This function defines a basic solenoid instrument capable of propagating spin-1/2 particles through a simple uniform field defined along the instrument z axis. The spin-1/2 propagation through this instrument is described in section 4.2.2.

Initially, the particles are propagated to the entrance aperture function and then transformed into the instrument frame. If necessary, a sudden spin transition is then performed by calling the suddenspintransition function for each particle as defined in section 7.2.9. As the field is assumed to be perfectly uniform through the entire solenoid, the spin propagation can be carried out in one time step. Once the time has been found for each particle to propagate through the length of the instrument, the propagatespinhalf function is called to carry out the spin integration. This function is described in section 7.2.10. The new spin state of each particle is then assigned to the particle structure and it is transformed back into the lab frame. The exit aperture function is then called to spatially propagate the particles through the instrument.

6.3.4 hexapoledipoletransition(particles,trajectories,param,gaussparam)

Input parameters: The particles and trajectories structures, the 3x3 parameters matrix containing information about the position and orientation of the component, and the parameters relating to the Gaussian distribution that controls the random spin rotation.
Output parameters: The particles structure (in the lab frame) and the trajectories structure.
This instrument is a simple approximation to the hexapole-dipole transition region for spin-1/2 particles; it rotates the spin-state of each particle randomly according to a velocity dependent random variable.

First, a vector of velocity magnitudes is generated from the particles structure, then

a vector of θ values is generated from a Gaussian random distribution that uses a mean of $(a*v+b)$ and a standard deviation of c. a,b and c are parameters defined by the user which are passed to the instrument as components of the gaussparam vector. Next, a vector of random values between 0 and 2π is generated, which are used to define the rotation axes (nx, ny) of each rotation in the (x, y) plane. These (nx, ny) values are then dotted with the θ values to generate vectors of rotation angles about the x and y axes. The combined spin rotation operator is then generated for each particle:

$$R = R_x * R_y = \begin{pmatrix} cos(\frac{\phi_x}{2}) & -isin(\frac{\phi_x}{2}) \\ -isin(\frac{\phi_x}{2}) & cos(\frac{\phi_x}{2}) \end{pmatrix} * \begin{pmatrix} cos(\frac{\phi_y}{2}) & -sin(\frac{\phi_y}{2}) \\ sin(\frac{\phi_y}{2}) & cos(\frac{\phi_y}{2}) \end{pmatrix}$$

The spin of each particle is then operated on by this matrix and updated.

6.4 Samples

This section contains the different samples that can be implemented in mobster.

6.4.1 flatsample(particles, trajectories, param)

Input parameters: The particles and trajectories structures and the 3x3 parameters matrix containing information about the position and orientation of the component
Output parameters: The particles structure (in the lab frame) and the updated trajectories structure.
This function simulates reflection of a perfectly flat sample. The initial sample orientation is such that if the sample is not rotated the surface normal points in the -z direction. In order to simulate reflection, each particle is transformed into the component frame, propagated to the plane of z = 0 and then the z component of the velocity is inverted for each particle. The trajectory structure is then updated and the particles are transformed back into the lab frame.

6.5 Detectors

6.5.1 simpledetector(particles,trajectories,param,radius)

Input parameters: The particles and trajectories structures, the 3x3 parameters matrix containing information about the position and orientation of the component, and the radius of the open space in the centre of the detector aperture.
Output parameters: The particles structure (in the lab frame) and the updated

trajectories structure.
This simple detector function is just a black hole aperture. It is assumed that all particles which pass through this aperture are detected.

Chapter 7
Non-instrument functions

7.1 Introduction

This chapter provides detail on how each function which is not associated directly with an instrument operates.

7.2 Functions

7.2.1 addtotrajectories(particles, trajectories, param)

Input parameters: The particles and trajectories structures and the 3x3 parameters matrix containing information about the position and orientation of the component.
Output parameters: The new trajectories structure
The add to trajectories function accepts the particles structure in the component frame and transforms it to the lab frame. It then takes the lab frame position and adds it as an entry to the trajectories structure, making sure to update the total number of entries in each trajectory by one.

7.2.2 angletonormalvector([alpha,beta,gamma])

Input parameters: The angles vector in the form [alpha, beta, gamma], which define the orientation of a component
Output parameters: nz and **nx**, the instrument normal and vertical direction expressed as lab frame unit vectors
This function takes the angles defined in section and generates the correct rotation matrices using the following formulae for each rotation:
The alpha rotation matrix R1:

$$\begin{pmatrix} cos(\alpha) & sin(\alpha) & 0 \\ -sin(\alpha) & cos(\alpha) & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

The beta rotation matrix R2:

$$\begin{pmatrix} cos(\beta) & 0 & -sin(\beta) \\ 0 & 1 & 0 \\ sin(\beta) & 0 & cos(\beta) \end{pmatrix}$$

The gamma rotation matrix R3:

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & cos(\gamma) & sin(\gamma) \\ 0 & -sin(\gamma) & cos(\gamma) \end{pmatrix}$$

The combined matrix RT is formed (=R3xR2xR1), which then operates on the unrotated nx and nz vectors in the lab frame. These being $\mathbf{n_z} = \hat{\mathbf{z}} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$ and $\mathbf{n_x} = \hat{\mathbf{x}} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$. The rotated nx and nz unit vectors are returned.

7.2.3 frametransform(v, r, param)

Input parameters: Instantaneous particle velocity and location, and the 3x3 matrix defining the instrument frame parameters(origin, **nz**, **nx** unit vectors).
Output parameters: Particle velocity and location in the instrument frame.
This function transforms velocity and position vectors from the lab frame to the instrument frame, using the method layed out in section 2.2. It defines the x',y' and z' axes in the lab frame using the **nx, nz** and **ny** = cross product(**nz,nx**). The rotation matrix is created from transpose of the the unit vectors in the x',y' and z' directions:

$$R' = \begin{pmatrix} nx(1) nx(2) nx(3) \\ ny(1) ny(2) ny(3) \\ nz(1) nz(2) nz(3) \end{pmatrix}$$

This is used to find the component frame velocity by directly transforming the lab frame velocity. Similarly, the component frame position vector is found by subtracting the lab frame origin of the component from the lab frame particle position vector before transforming it with R'. The component frame velocity and position are returned.

7.2.4 labtransform(v, r, param)

Input parameters: Instantaneous particle velocity and location, and the 3x3 matrix defining the instrument frame parameters(origin, **nz**, **nx** unit vectors).
Output parameters: Particle velocity and location in the lab frame.
This function takes the instrument frame position and velocity vectors of a particle and converts them to the lab frame, using the method layed out in section 2.2. It defines the x', y' and z' axes using the **nx, nz** and **ny** = cross product(**nz,nx**). The rotation matrix is created from the lab frame unit vectors in the x',y' and z' directions:

$$R = \begin{pmatrix} nx(1)ny(1)nz(1) \\ nx(2)ny(2)nz(2) \\ nx(3)ny(3)nz(3) \end{pmatrix}$$

This is used to find the lab frame velocity by directly transforming the component frame velocity. Similarly, the new position vector is found by transforming the lab frame particle position vector with R before adding the lab frame origin. The lab frame velocity and position are returned.

7.2.5 propagate(particles)

Input parameters: The particles structure in the instrument frame.
Output parameters: The updated particles structure.
This function propagates all particles forward in a straight line until the z component of the instrument frame position vector equals 0. In the lab frame this corresponds to all particles existing within a plane oriented with the instrument containing the instrument origin.

To accomplish this, the z component of velocity and position is used to find the time required to propagate each particle such that z=0. The new position is then found by adding the initial position vector to this calculated time multiplied by the velocity vector. The position and time entries in the particle structure are updated for each particle, and the particle structure is returned.

7.2.6 RandPoints(N,param,radius)

Input parameters: The number of particles being generated, the 3x3 parameters matrix which defines the instrument frame parameters and the radius of the circle.
Output parameters: An N by 3 matrix where each row corresponds to the x, y and z lab frame coordinates of points generated on a circle defined by the radius and parameters matrix.

The rand function is used to generate phi values between 0 and 2pi and rho values between 0 and r. To make every point on the circle equally likely, the rho values are generated by taking the square root of random numbers between 0 and 1. These values are then converted into position frame (x,y,z) cartesian coordinates and then lab frame cartesian coordinates before being returned as a list of random points on a circle.

7.2.7 removetrajec(particles, trajectories, rlimit,block)

Input parameters: The particles structure (in the component frame), the trajectories structure, the limiting off axis r value and the block parameter, defining whether or not the aperture is blocking.
Output parameters: The new trajectories structure and the component frame particles structure.
This function indexes the particles and corresponding trajectories that are less than a distance rlimit from the instrument axis if block = 0, and more than a distance rlimit if block = 1. It then initialises new empty particles and trajectories structures which have a size equal to the total number of indices in the list. The entries in these new particles and trajectories structures are then set to be equal to the indexed (allowed) particle and trajectory entries in the original structures. In this way, the particles which do not make it through each component are removed from the simulation and are not propagated further. Doing it this way is far faster than removing each entry from the original lists individually.

7.2.8 removetrajecrectangle(particles, trajectories, width, height, block)

Input parameters: The particles structure (in the component frame), the trajectories structure, the limiting off width and height values and the block parameter, defining whether or not the aperture is blocking.
Output parameters: The new trajectories structure and the component frame particles structure.
This function indexes the particles and corresponding trajectories that are either both less than a y distance width/2 and x distance height/2 from the instrument axis (for the non-blocking case) or both greater than a y distance width/2 and x distance height/2 from the instrument axis (for the blocking case). It then initialises new empty particles and trajectories structures which have a size equal to the total number of indices in the list. The entries in these new particles and trajectories structures are then set to be equal to the indexed (allowed) particle and trajectory entries in the original structures. In this way, the particles which do not make it through each component

are removed from the simulation and are not propagated further. Doing it this way is far faster than removing each entry from the original lists individually.

7.2.9 suddenspintransition(oldB,newB,spin)

Input parameters: The old defined B field direction in the lab frame, the new defined B field direction in the lab frame and the initial particle spin vector.
Output parameters: The new spin vector after the transition.
This particle takes two magnetic field vectors in the lab frame as input arguments and a spin vector. It carries out the sudden spin transition on the particle using the spin rotation method defined in detail in section 4.3.2. It returns the spin vector after the sudden transition defined relative to the new field direction.

7.2.10 propagatespinhalf(oldspin,gmr,B,t)

Input parameters: The old spin vector before the short field increment, the gyromagnetic ratio, the magnitude of the B field, and the time spent in the uniform field element t.
Output parameters: The new spin state after the propagation in the uniform field element.
This is a simple propagation function for spin-1/2 particles in a uniform field element, where the spin vector undergoes Larmor procession as defined in section 4.2.2. The angle of rotation is calculated using the provided parameters, and then the spin rotation operator R_z is generated.

$$\phi_z = -\gamma |B| \delta t$$

$$R_z = \begin{pmatrix} exp(-i\frac{\phi_z}{2}) & 0 \\ 0 & exp(i\frac{\phi_z}{2}) \end{pmatrix}$$

Chapter 8
Adapting MoBSTer

Abstract This chapter provides some guidance for advanced users on common cases where adapting some of the core mechanics of MoBSTer may be useful. These include: altering the data stored by the trajectory structure; implementing numerical fields; and implementing a larger number of spin states.

8.1 Storing different information in the trajectories structure

It is a likely scenario that a user may want alter the function of the trajectories structure. This could take the form of either stopping data storage in the trajectory vector in order to greatly decrease memory use to run simulations with very high numbers of particles, or increasing the amount of data stored if the user wishes to track different particle parameters during propagation around the machine. The following sections describe how both of these can be done in detail.

8.1.1 Disabling the trajectory vector

The alterations which need to be made to the default functions to disable all data storage in the trajectory vector in MoBSTer v1.0 are detailed below:

- In the simplesourceV2 function, comment out the lines that initialise the trajectory vector and add the initial location data:

```
trajectories = repmat(struct('position',zeros(100,3),'Numberofentries',1),N,1);
.
.
trajectories(int).position(1,:) = points2(int,:);
```

  Add the following line:

```
trajectories = [];
```

This means that a trajectories variable will still be passed around, even if it is empty, preventing many errors.

- In the addtotrajectories.m function, comment out every line.
- In the removetrajec.m and removetrajecrectangle.m, comment out the lines:

```
newtrajectories = repmat(struct('position',zeros(100,3),'Numberofentries',1),N,1);
.
.
newtrajectories(i) = trajectories(rm(i));
```

And add the line

```
newtrajectories = trajectories;
```

- In the hexapole function, comment out the lines:

```
newtrajectories = repmat(struct('position',zeros(100,3),'Numberofentries',1)
,numel(particles)*2,1);
.
newtrajectories((2*i-1):(2*i)) = [trajectories(i),trajectories(i)];
.
.
for i = 1:numel(newparticles)
    N = newtrajectories(i).Numberofentries;
    newtrajectories(i).Numberofentries = N+stepno+1;
    newtrajectories(i).position(N+1:N+stepno+1,:) = cell2mat(labpos(:,i));
end
```

and add the line

```
newtrajectories = trajectories;
```

8.1.2 Storing different information in the trajectory vector

The alterations which need to be made to the default functions to increase the data storage in the trajectory vector in MoBSTer v1.0 are detailed below:

- In simplesourceV2, add the extra data you are storing to the initialisation step, for example to store spin data, the line:

```
trajectories = repmat(struct('position',zeros(100,3),'Numberofentries',1),N,1);
```

should be changed to

```
trajectories = repmat(struct('position',zeros(100,3),'spin',zeros(100,2),
'Numberofentries',1),N,1);
```

Additionally, add the line to the end of the for loop to store the initial data point. For spin:

```
trajectories(int).spin(1,:) = particles(int).spin;
```

- In the addtotrajectories.m function, add the line which stores your new datatype. For spin:

```
trajectories(i).spin(trajectories(i).Numberofentries,:) = particles(i).spin;
```

For velocity, change the tilde in first the transform line to a variable name (e.g, vel) and then save this to the structure in the same way as position.

- In removetrajec and removetrajectrectangle, add the extra data you are storing to the initialisation of the newtrajectories structure. For spin:

```
newtrajectories = repmat(struct('position',zeros(100,3),'spin',zeros(100,2),
'Numberofentries',1),N,1);
```

- In the hexapole function, add the extra data you are storing to the initialisation of the newtrajectories structure. For spin:

```
newtrajectories = repmat(struct('position',zeros(100,3),'spin',zeros(100,2)
'Numberofentries',1),numel(particles)*2,1);
```

Additionally, the for loop at line 137 needs to be updated in order add the data the user wants stored for every hexapole step. If the user wishes to store spin, or another variable which is constant throughout the hexapole, this is simple and the following line should be added:

```
newtrajectories(i).spin(N+1:N+stepno+1,:) = newparticles(i).spin.*ones(stepno+1,2);
```

If the user wishes to store the particle velocity, the following line must be added which performs a lab transform on the velocity cell matrix:

```
labvel = cellfun(@(M)((R*M')'),vel,'UniformOutput',false);
```

and then the velocity can be stored in the structure by adding a line equivalent to the storage of the position within the for loop, as so:

```
newtrajectories(i).velocity(N+1:N+stepno+1,:) = cell2mat(labvel(:,i));
```

8.2 Implementing numerical fields

The following section details how the already written MoBSTer framework can be re-purposed for a numerical field instrument. Whilst the user will have to do the majority of coding themselves, MoBSTer was developed with eventual numerical field integration in mind.

The major functionality already written into MoBSTer which is important in numerical fields is that the spin vector of each individual particle can be defined with respect to a different field direction (using the Bfield component of the particles structure). This means that no generality is lost as particles propagate through numerical field instruments.

Additionally, the sudden spin transition function is implementable for numerical fields to transform the spins of the entire particles structure. Calling this function in this case would still be very simple, and would be of the form:

```
if strcmp('sudden', entrytransition) %if the entry transition is sudden
    for i = 1:numel(particles)
        NewB = Numericalfield(particles(i).position)
        particles(i).spin = suddenspintransition(particles(i).Bfield,NewB,particles(i).spin))
    end
end
```

Where NewB is the B field vector of the numerical field at the position of the particle.

The propagate spin-1/2 function has also been designed with incremental units of a uniform field in mind, and may prove useful in propagation through numerical fields with short time-steps.

8.3 Implementing a larger amount of spin states

Finally, by the versatile nature of MoBSTer the propagation of non-spin 1/2 molecules is possible. The following provides a list of the functions in MoBSTer v1.0 that will need to be changed or re-made:

Those that need to be changed (only changing an element of the function to work with more spin states):

- The source function (change the initialisation of the spin vectors).
- Sudden spin transition function (Use rotation operators that work on higher numbers of spin states).
- Hexapole dipole transition function (Use rotation operators that work on higher numbers of spin states.)

Those that need to be completely re-made:

- The hexapole instrument. (Though much of the current instrument code could be re-purposed)
- The propagate spin function (for the solenoid - the schrodinger equation will need to be re-solved for this higher number of spin states).

The user is not restricted to the pre-made instrument functions, and could easily just create an entire simulation within the MoBSTer syntax using new instruments.

References

[1] P. Fouquet et al. “Thermal energy He3 spin-echo spectrometer for ultrahigh resolution surface dynamics measurements”. In: *Review of Scientific Instruments* 76.5 (2005), p. 053109. DOI: 10.1063/1.1896945. eprint: https://doi.org/10.1063/1.1896945. URL: https://doi.org/10.1063/1.1896945.

[2] A. P. Jardine et al. “Hexapole magnet system for thermal energy 3He atom manipulation”. In: *Review of Scientific Instruments* 72.10 (2001), pp. 3834–3841. DOI: 10.1063/1.1405794. eprint: https://doi.org/10.1063/1.1405794. URL: https://doi.org/10.1063/1.1405794.

[3] Malcolm Harris Levitt. *Spin Dynamics Basics of Nuclear Magnetic Resonance Second Edition*. John Wiley and Sons Ltd, 2015. ISBN: 9780470511183.

[4] Marcel Utz et al. “Visualisation of quantum evolution in the Stern–Gerlach and Rabi experiments”. In: *Phys. Chem. Chem. Phys.* 17 (5 2015), pp. 3867–3872. DOI: 10.1039/C4CP05606J. URL: http://dx.doi.org/10.1039/C4CP05606J.