Discrete-event simulation of a single-neutron interferometry experiment

Jurriaan Pruis

July 6, 2015

Supervisors:
H. De Raedt
R. G. E. Timmermans

Abstract

In this thesis it is shown that by using a discrete-event based approach it is possible to do simulations that match both the theory and the experimental data without using any probability-based theory / equations, e.g. without use of the Schrödinger equation. From the resulting simulations it is clear that an event-based approach is a good way to describe the working of the single-neutron interferometer on a particle-by-particle basis.

Contents

1 An introduction to event-based simulation

In this thesis we will consider first the simple case of a single-photon beam splitter to familiarize ourselves with event-based simulation. Subsequently we will take a look at a neutron interferometry experiment and compare the results of both the beam splitter and the interferometry experiment with theoretical and experimental results.

When you think about simulating a quantum mechanical experiment, it seems that the simplest way to simulate is to just plug some values into the equations you get from the quantum mechanical theory. But, when you think about it, merely entering values does not really produce a simulation, since in that case you only compute the probabilities.

As said in [4]: "The mathematical framework of quantum theory allows for a straightforward calculation of numbers which can be compared with experimental data as long as these numbers refer to statistical averages of measured quantities."

Enter the event-based approach - a radically different way of working which makes it possible to simulate the experiments on a event-by-event/particle-by-particle basis without ever touching the Schrödinger equations.

In an event-based model a neutron is described as being a messenger that carries a message. This message is transformed by the system as it travels through its components (a beam splitter / phase shifter etc.). As these messages travel along a single path, there is no possibility of interference, which is very interesting because most interpretations of quantum theory say that interference patterns arising in for example the double-slit experiment are due to the particle interfering with itself. A process which is not allowed in the event-based approach.

The 'heart' of this discrete-event approach is the deterministic learning machine, a system that learns about the incoming events and produces output according to its current state.

The C++ source code used to do the simulations is available at https://github.com/jurriaan/event-based [6] or in Appendix A.

2 Single-photon beam splitter

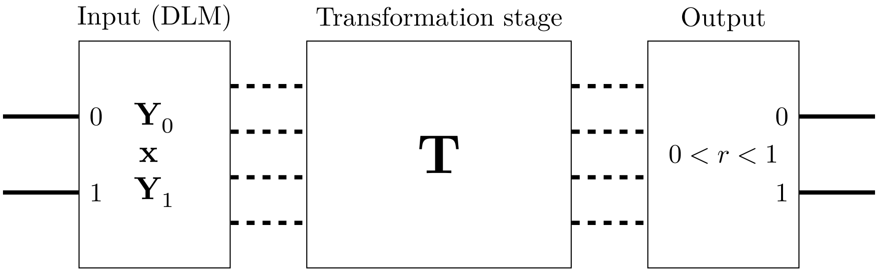

Figure 1: Basic structure of an event-based beam splitter which has two inputs and two outputs. Messages enter at one of the inputs, are processed by the input stage and transformed by the matrix **T**, to eventually leave the beam splitter via the output stage by one of the two outputs.

To get a feeling of how event-based simulation works, we will start at the basics, with a simple photonic beam splitter. We will consider a 50-50 beam splitter, which implies that $T = R = 1/2$, e.g. the transmissivity is equal to the reflectivity.

2.1 Theory

For a 50-50 beam splitter we have, according to quantum theory[1]:

$$\begin{pmatrix} b_0 \\ b_1 \end{pmatrix} = \frac{1}{\sqrt{2}} \begin{pmatrix} a_0 + ia_1 \\ a_1 + ia_0 \end{pmatrix} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & i \\ i & 1 \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \end{pmatrix}. \tag{1}$$

In the simulation of the beam splitter we will use the message $(cos\,\psi_0, sin\,\psi_0)$ for input channel 0 and $(cos\,\psi_1, sin\,\psi_1)$ for input channel 1. In quantum theory this corresponds to: [1]

$$a_0 = \sqrt{p_0}e^{i\psi_0}, \tag{2a}$$
$$a_1 = \sqrt{1-p_0}e^{i\psi_1}. \tag{2b}$$

From this you can compute the output probabilities.

The probability of detecting a particle at outputs 0 or 1 is given by the following equations:

$$P_{b_0} = b_0^* b_0 = \frac{1 + 2\sqrt{1-p_0}\,sin\,(\psi_0 - \psi_1)}{2}, \tag{3a}$$
$$P_{b_1} = b_1^* b_1 = \frac{1 - 2\sqrt{1-p_0}\,sin\,(\psi_0 - \psi_1)}{2}. \tag{3b}$$

with the input probability p_0 (the chance of a photon entering at input 0).

[1] Using the same definition as in [1]: "We use the term quantum theory when we refer to the mathematical formalism, i.e., the postulates of quantum mechan- ics (with or without the wave function collapse postulate) and the rules (algorithms) to compute the wave function. The term quantum physics is used for microscopic, experimentally observable phenomena that do not find an explanation within the mathematical framework of classical mechanics."

2.2 Event-based approach

2.2.1 Messages

In the event-based approach a particle (in this case a photon) is seen as a messenger. The messenger carries a message that is received and modified by the various components with which it interacts (for example the beam splitter).

The most common way to represent a message is by a two-dimentional complex-valued unit vector. For example: [5]

$$\mathbf{y} = \begin{pmatrix} e^{i\psi^{(1)}} \, sin\,\xi \\ e^{i\psi^{(2)}} \, cos\,\xi \end{pmatrix}, \tag{4}$$

which matches Maxwell's theory and quantum theory.

In this simple single-photon case we use a non-polarizing beam splitter and we do not need to encode phases into our messages. This means that a simplified message representation is sufficient:

$$\mathbf{y} = \begin{pmatrix} sin\,\psi \\ cos\,\psi \end{pmatrix}. \tag{5}$$

2.2.2 Deterministic Learning Machine / Input stage

As seen in figure 1 the beam splitter, and thus the DLM, has two inputs. If we represent the messages that arrive at port 0 and 1 by the vectors $\mathbf{v} = (1, 0)$ and $\mathbf{v} = (0, 1)$ respectivly, we can use the DLM described in section A.3 of [5].

For this we need a two-dimensional internal vector $\mathbf{x}$ where $x_0 + x_1 = 1$ and $x_0, x_1 \geq 0$. In addition we need two registers $\mathbf{Y}_k = \left(\mathbf{Y}_{k,0}, \mathbf{Y}_{k,1}\right)$ to store the last message $\mathbf{y}$ that arrived at port k.

After receiving a message at port k the message is processed as following:

First the message is copied to the corresponding register $\mathbf{Y}_k$. The internal vector is then updated according to the rule:

$$\mathbf{x} \leftarrow \gamma\mathbf{x} + (1-\gamma)\mathbf{v}, \tag{6}$$

where $0 \leq \gamma < 1$. This parameter γ determines the rate at which the DLM 'learns' its input. As we will see later on, this will affect the visibility of the interference fringes.

2.2.3 Transformation stage

The transformation stage uses the registers $\mathbf{Y}_k$ and internal vector $\mathbf{x}$ and transforms them in order to generate input that enables the output stage to decide the output message and port.

In this case we use the following rule for this simple photonic beam splitter:

$$\frac{1}{\sqrt{2}} \begin{pmatrix} \mathbf{Y}'_{0,0} - \mathbf{Y}'_{1,1} \\ \mathbf{Y}'_{1,0} + \mathbf{Y}'_{0,1} \\ \mathbf{Y}'_{1,0} - \mathbf{Y}'_{0,1} \\ \mathbf{Y}'_{0,0} + \mathbf{Y}'_{1,1} \end{pmatrix} \leftarrow \begin{pmatrix} \mathbf{Y}'_{0,0} \\ \mathbf{Y}'_{0,1} \\ \mathbf{Y}'_{1,0} \\ \mathbf{Y}'_{1,1} \end{pmatrix}, \tag{7}$$

where $\mathbf{Y}'_{i,j} = \sqrt{x_i}\,\mathbf{Y}_{i,j}$.

Which using a matrix notation we can rewrite as:

$$\begin{pmatrix} \mathbf{Z}_{0,0} \\ \mathbf{Z}_{1,0} \\ \mathbf{Z}_{0,1} \\ \mathbf{Z}_{1,1} \end{pmatrix} = \mathbf{T} \begin{pmatrix} \mathbf{Y}'_{0,0} \\ \mathbf{Y}'_{1,0} \\ \mathbf{Y}'_{0,1} \\ \mathbf{Y}'_{1,1} \end{pmatrix} = \frac{1}{\sqrt{2}} \begin{pmatrix} \mathbf{Y}'_{0,0} - \mathbf{Y}'_{1,1} \\ \mathbf{Y}'_{1,0} - \mathbf{Y}'_{0,1} \\ \mathbf{Y}'_{1,0} + \mathbf{Y}'_{0,1} \\ \mathbf{Y}'_{0,0} + \mathbf{Y}'_{1,1} \end{pmatrix}, \tag{8}$$

with

$$\mathbf{T} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 0 & 0 & -1 \\ 0 & 1 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{pmatrix}. \tag{9}$$

The resulting registers $\mathbf{Z}_k$ are sent to the output stage.

2.2.4 Output stage

The output stage uses the data from the transformation stage to send the modified message to the output k'. First we compute $z = \left|\mathbf{Z}_{1,0}\right|^2 + \left|\mathbf{Z}_{1,1}\right|^2$ and use this value to select the output port by the rule:

$$k' = \Theta(z - r), \tag{10}$$

where $\Theta(x)$ is the unit step function and $0 \leq r < 1$ is a uniform pseudo-random number.

The message (which is normalized for internal consistency)

$$\mathbf{z} = \frac{1}{\sqrt{\left|\mathbf{Z}_{k',0}\right|^2 + \left|\mathbf{Z}_{k',1}\right|^2}} \begin{pmatrix} \mathbf{Z}_{k',0} \\ \mathbf{Z}_{k',1} \end{pmatrix}, \tag{11}$$

is then sent to output k'.

2.3 Results

We now have all the ingredients needed to do an event-based simulation of the beam splitter.

The process of simulating the beam splitter is as follows. We first generate two random messages:

$$\mathbf{y}_0 = \begin{pmatrix} sin\,\psi_0 \\ cos\,\psi_0 \end{pmatrix}, \mathbf{y}_1 = \begin{pmatrix} sin\,\psi_1 \\ cos\,\psi_1 \end{pmatrix}. \tag{12}$$

The phase difference between those two messages ($\chi = \psi_0 - \psi_1$) is used for the x-axis of the plot.

After generating these two messages, we determine input i to which a message is to be sent with probability p_0. The message y_i is then sent to this input. We repeat this proces N times, after which two new messages are generated and the process starts all over again.

Figure 2 shows the probability of finding a particle at output 0.

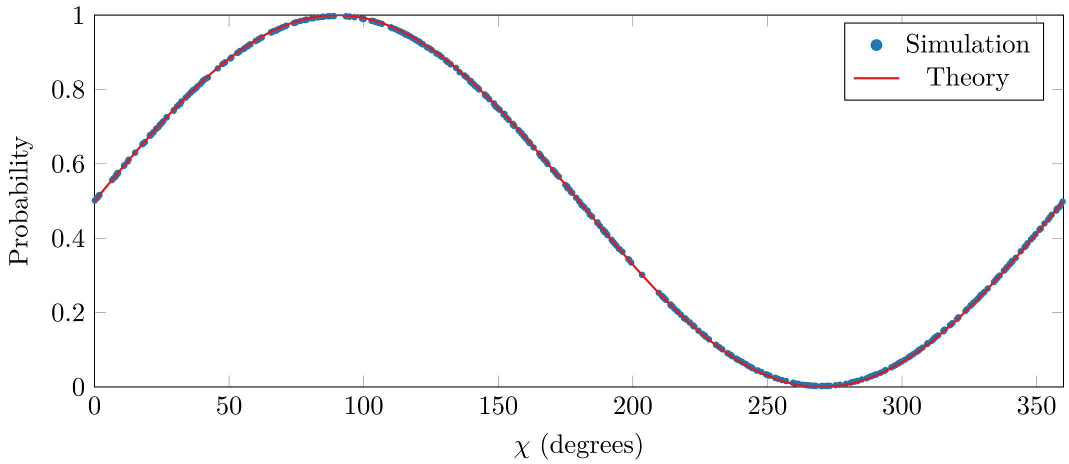

Figure 2: Beam splitter simulation results (parameters: $\gamma = 0.99$, $p_0 = 1/2$, $R = T = 1/2$ with a root mean square error of 0.0019). This graph shows the normalized count of the particles leaving at output 0 against the phase difference χ; it clearly shows that the simulation (blue dots) matches the theory (red line) very closely.

In order to convert the counts into a probability, we divided the number of counts at the output by the total number of counts:

$$P_{b_i} = \frac{N_i}{N_0 + N_1}, \tag{13}$$

where $i = 0, 1$.

As seen in Figure 2 the simulation matches the theory given by Equation 3 very well (the root mean square error is 0.0019).

3 Single-neutron interferometry

Having established that the results of our photon-based simulations match the theoretical predictions, we can use the acquired knowledge and techniques for the more complex Mach-Zehnder type of neutron interferometer experiment.

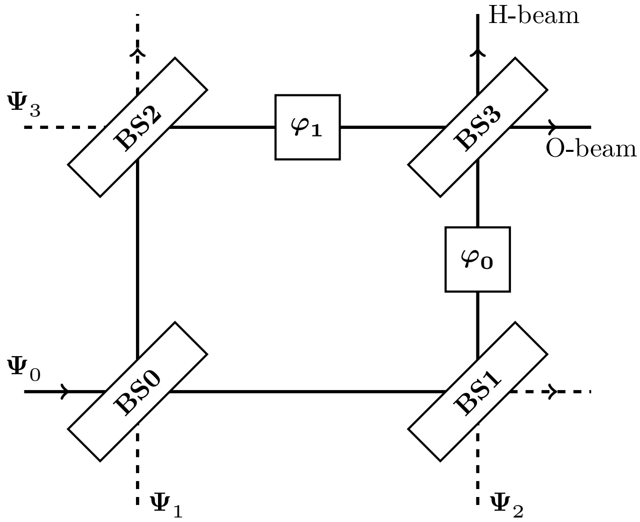

Figure 3: Diagram of the single-neutron interferometer, **BS0**...**BS3** are beam splitters, φ_0 and φ_1 phase shifters. The 'beam' enters left of **BS0** at $\mathbf{\Psi}_0$ and travels through the interferometer. Messengers that exit at the H- and O-Beam outputs of **BS1** are counted. The additional (dashed) inputs of the other beam splitters are not used in the experiment, but are needed for the quantum mechanical theory.

The diagram in Figure 3 shows an event-based model of the silicon-perfect-crystal neutron interferometer. It is a simplified view of an experiment done with a silicon-perfect-crystal neutron interferometer as shown in Figure 4 to which we will compare the simulation in the next chapter.

In this experiment a beam of neutrons enters the interferometer at $\mathbf{\Psi}_0$, travels through the interferometer and, with a certain probability, passes through **BS3**, after which the O-beam and H-beam components are counted individually, using detectors.

The beam splitters **BS0**, **BS1** and **BS2** have inputs that are not used in the simulation and experiment, but are needed for the quantum mechanical description.

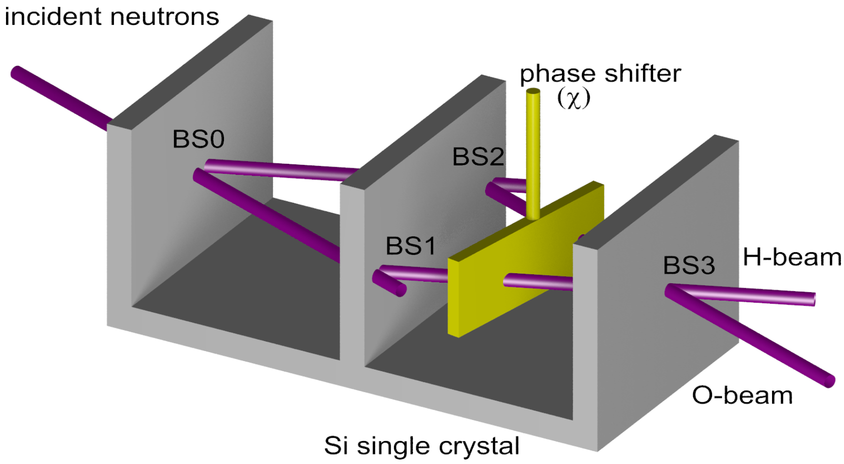

Figure 4: A schematic picture of a silicon-perfect-crystal MZI neutron interferometer. The picture features a phase shifter that blocks both beams instead of two individual phase shifters. The χ in this picture is equal to $\varphi_1 - \varphi_0$ from the diagram in Figure 3

3.1 Theory

The state vector $|\Psi\rangle$ is given by:

$$|\Psi\rangle = (\mathbf{\Psi}_0, \mathbf{\Psi}_1, \mathbf{\Psi}_2, \mathbf{\Psi}_3)^T . \tag{14}$$

As the message propagates through the interferometer, it changes according to:

$$|\mathbf{\Psi'}\rangle = \begin{pmatrix} t^* & r \\ -r^* & t \end{pmatrix}_{2,3} \begin{pmatrix} e^{i\varphi_0} & 0 \\ 0 & e^{i\varphi_1} \end{pmatrix}_{2,3} \begin{pmatrix} t^* & r \\ -r^* & t \end{pmatrix}_{1,3} \begin{pmatrix} t & -r^* \\ r & t^* \end{pmatrix}_{0,2} \begin{pmatrix} t & -r^* \\ r & t^* \end{pmatrix}_{0,1} |\mathbf{\Psi}\rangle . \tag{15}$$

The subscripts in this equation refer to the pair of elements of the state vector on which the matrix acts.

Computing this for the input $|\mathbf{\Psi}\rangle = (1, 0, 0, 0)$, allows us to compute the probability of a particle leaving through the H- or O-beam output:

$$P_H = \left|\mathbf{\Psi}'_2\right|^2 = R\left(T^2 + R^2 - 2RT\, cos\, \chi\right), \tag{16a}$$

$$P_O = \left|\mathbf{\Psi}'_3\right|^2 = 2R^2 T\left(1 + cos\, \chi\right). \tag{16b}$$

The χ in these equations matches the χ in Figure 4, which equals $\varphi_1 - \varphi_0$.

3.2 Messages

In the case of the single-neutron interferometry experiment we need to represent the message by a two-dimensional complex-valued unit vector:

$$\mathbf{y} = \left(e^{i\psi^{(1)}}\, cos\,(\theta/2),\, e^{i\psi^{(2)}}\, sin\,(\theta/2)\right). \tag{17}$$

The message also has a time-dependency:

$$\mathbf{y} \leftarrow e^{ivT}\mathbf{y}, \tag{18}$$

but, since we are considering a monochromatic beam of neutrons, we can consider this phase a constant.

The messages are created and sent one-by-one (which is important to show that we can produce interference patterns without neutrons interacting with each other).

We can create messages with specific properties. In order to create a fully coherent spin-polarized beam, the generated messages will all have the same fixed $\psi^{(1)}$, $\psi^{(2)}$ and θ.

In this specific example we used a beam consisting solely of the following messages:

$$\mathbf{y} = \begin{pmatrix} 0.5 + 0.5i \\ 0.5 + 0.5i \end{pmatrix}. \tag{19}$$

3.3 Components

The single-neutron interferometry experiment consists of multiple phase shifters, beam splitters and detectors. This allows for a modular approach at the side of the simulation, as the components of the same type can be reused. We only have to build those components once and then link them together.

3.3.1 Phase shifter

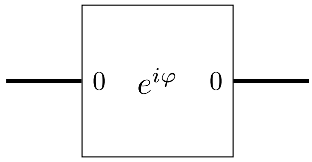

Figure 5: Basic structure of a phase shifter, which has one input and one output. This component shifts the phase of the incoming message with φ radians by multiplying the message with $e^{i\varphi}$.

The phase shifter is a very simple component with a single input and output. It shifts the phase of the input message and emits the phase-shifted message at its output. In neutron experiments, phase shifters usually consist of metal foil that changes the time of flight (thus shifting the phase). In this model we take the absorption as negligible.

This leaves us with the following equation:

$$f(\varphi) : \mathbf{u} \rightarrow e^{i\varphi}\mathbf{u}, \tag{20}$$

where φ is the phase shift.

3.3.2 Beam splitter

We can use the same definition of a beam splitter as before, except for the fact that this beam splitter introduces a phase shift. This means, in combination with the slightly different interpretation of the messages (which now contain the polarisation of the messenger), that the transformation rules and thus transformation matrix $\mathbf{T}$ are a bit different. [2, 3]

The following rule is used for the neutron beam splitter:

$$\begin{pmatrix} \sqrt{T}\mathbf{Y}'_{0,0} + i\sqrt{R}\mathbf{Y}'_{0,1} \\ i\sqrt{R}\mathbf{Y}'_{0,0} + \sqrt{T}\mathbf{Y}'_{0,1} \\ \sqrt{T}\mathbf{Y}'_{1,0} + i\sqrt{R}\mathbf{Y}'_{1,1} \\ i\sqrt{R}\mathbf{Y}'_{1,0} + \sqrt{T}\mathbf{Y}'_{1,1} \end{pmatrix} \leftarrow \begin{pmatrix} \mathbf{Y}'_{0,0} \\ \mathbf{Y}'_{0,1} \\ \mathbf{Y}'_{1,0} \\ \mathbf{Y}'_{1,1} \end{pmatrix}. \tag{21}$$

In this transformation rule, R and T $(= 1 - R)$ are the reflection and transmission coefficients.

This rule can be rewritten as the following transformation matrix:

$$\mathbf{T} = \begin{pmatrix} \sqrt{T} & i\sqrt{R} & 0 & 0 \\ i\sqrt{R} & \sqrt{T} & 0 & 0 \\ 0 & 0 & \sqrt{T} & i\sqrt{R} \\ 0 & 0 & i\sqrt{R} & \sqrt{T} \end{pmatrix}. \tag{22}$$

3.3.3 Detector

The detector is a component with one input that only counts the incoming messages. In reality a detector has an efficiency < 100%, but in this simulation we just count all incoming messages.

3.4 Results

We now have all the information needed to build the individual components of this interferometer and combine them to be able to simulate the experiment.

The simulation is done as follows. First we generate two random numbers that will be used as the phase of the phase shifters. Then we compute the relative phase difference $\chi = \varphi_1 - \varphi_0$, which we use for the x-axis of the following figures. After this we can repeatedly ($N = 10^7$ in this case) send a fixed message (as the beam is monochromatic) to the first beam splitter. We then save the number of counts that are detected by both the O- and H-beam detectors and select two new phases randomly. This process is repeated 400 times to generate Figure 6.

Figure 6 shows the results of a simulation in which beam splitters with a reflectivity of $R = 0.2$ are used.

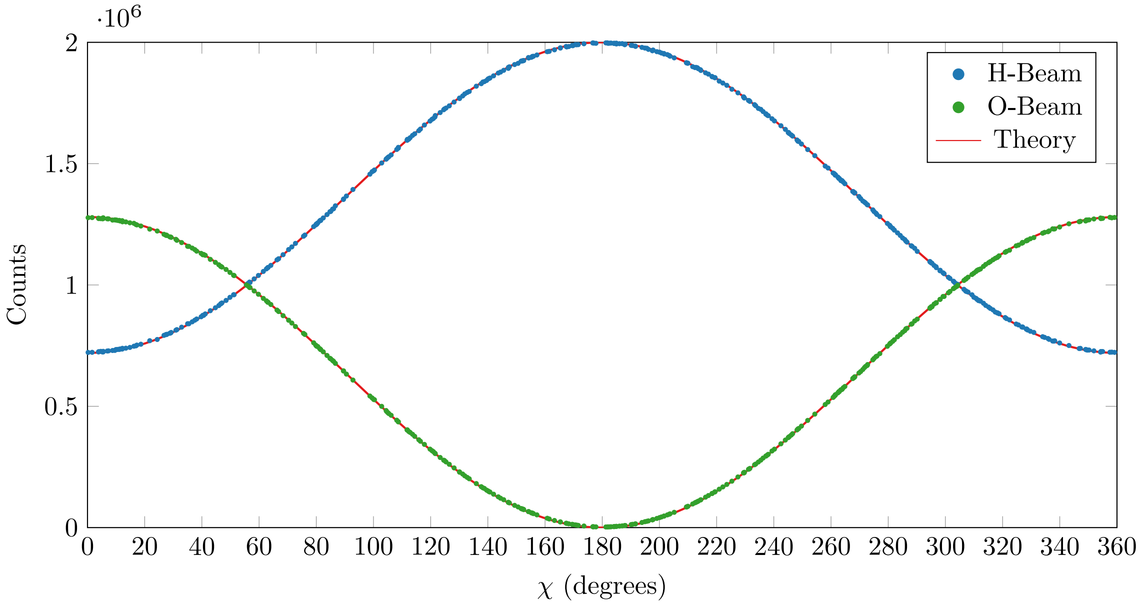

Figure 6: Neutron interferometry simulation results, 400 points per line, 10^7 messages per point. Model parameters: $R = 0.2, \gamma = 0.99$, with a RMSE of 2072 for the O-beam and a RMSE of 2222 for the H-beam. Both the H- and O-Beam counts are plotted against the phase difference χ. This graph clearly shows that the H- and O-Beam counts (the blue and green dots) closely match the theory (the red line).

As shown by Figure 6, the simulation matches the theory (given by Equation 16) very well.

4 Comparison with experimental results

We start with two sets of data from a single-crystal neutron interferometry experiment. As the direct relation between the data and the phase shift is not known, we first have to fit the data to get a scaling factor and phase shift.

For this we use the following formula:

$$f(x) = a + b\,cos\,(d \cdot \varphi) + c\,sin\,(d \cdot \varphi). \tag{23}$$

This gives us the following fit parameters:

Table 1: Parameters obtained by fitting the data sets to Equation 23. The full data set was fitted in one go, to obtain a single d-value for it. The quality of the 2nd data set was a bit lower which resulted in a lower R^2 value. Later on we will see that this data set has a much lower visibility.

	1st dataset O-count	**1st dataset H-count**	**2nd dataset O-count**	**2nd dataset H-count**
a	5336	1.037×10^4	2138	5942
b	-4547	4670	-334.4	238.9
c	1467	-1192	77.57	-55.15
d	4.793		4.877	
$\mathbf{R^2}$	0.998	0.997	0.943	0.805

Using the following equation, we can determine the phase shift θ that corresponds to the position of the encoder:

$$f(\varphi) = a + b\,cos\,(d \cdot \varphi) + c\,sin\,(d \cdot \varphi) = a + \sqrt{b^2 + c^2}\,cos\,(d \cdot \varphi + \theta), \tag{24}$$

where $tan\,\theta = -c/b$.

This gives the following phase shifts:

Table 2: The phase shifts θ. As the H- and O-counts are of the same measurements, the phase shifts should be the same, which is approximately the case.

	1st dataset	**2nd dataset**
O-count	0.2657	0.2280
H-count	0.2499	0.2269

Now that we have determined the phase shifts and scaling factors, we can use them to map the O- and H-beam counts to the phase shift χ used by the theoretical neutron interferometry experiment.

Using the theoretical P_H and P_O (Equation 16), we can determine the reflectivity/transmitivity:

$$\frac{P_H}{P_O}\left(\chi = \frac{\pi}{2}\right) = \frac{R\,(T^2 + R^2)}{2R^2T}. \tag{25}$$

Rewriting this equation for a normalized P_O and P_H ($P_O + P_H = 1$) gives us:

$$R = \frac{1}{2} \pm \sqrt{\frac{1}{4} - \frac{1}{2} \cdot P_O\left(\chi = \frac{\pi}{2}\right)}, \tag{26}$$

which we will use to determine the reflectivity making use of the data fits we did above.

From this equation we get:

Table 3: Reflectivity and transmitivity values computed for both data sets, using Equation 26.

	1st dataset	**2nd dataset**
R	0.2169	0.1570
T	0.7831	0.8458

The next step is figuring out the visibility. Both data sets are clear examples of a visibility $\neq 1$ as both the H- and O-count fits never touch $y = 0$.

To find the visibility for both data sets we first fit the data to the following formulae:

$$P_H = R\left(T^2 + R^2\right)\left(1 - v_H \cos\chi\right) \tag{27a}$$
$$P_O = 2R^2 T\left(1 + v_O \cos\chi\right) \tag{27b}$$

where v_O and v_H are the visibility parameters.

After fitting the data sets, we get the following visibilities:

Table 4: The visibilities for the O- and H-beams of both data sets computed using eqs. (27a) and (27b).

	1st dataset	**2nd dataset**
v_O	0.8829	0.1605
v_H	0.4648	0.04126

Using the values of R extracted from the experimental data, we performed two simulations:

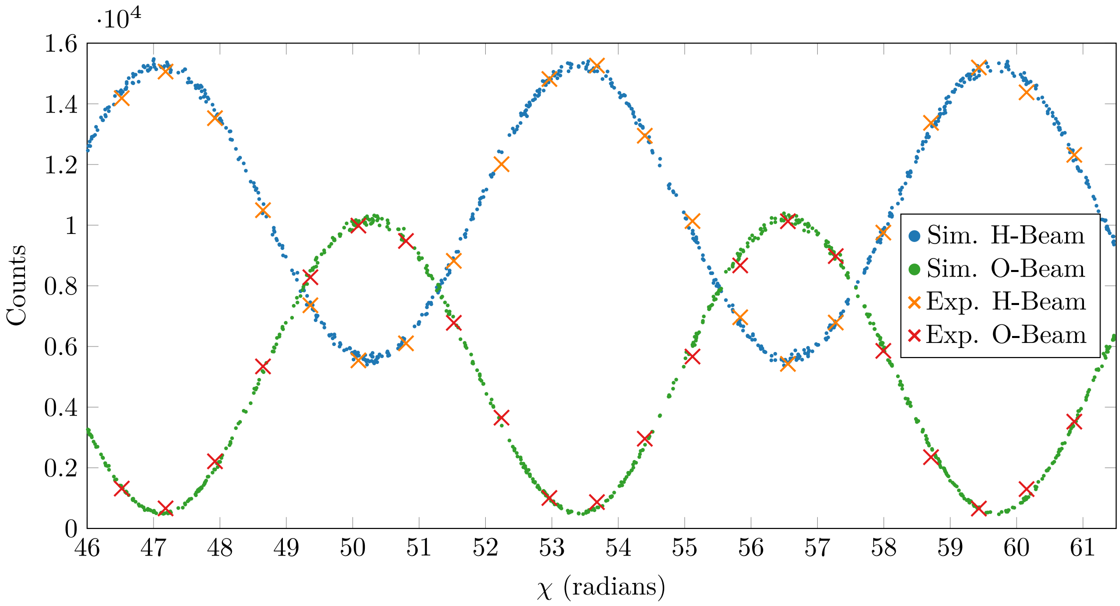

Figure 7: Neutron interferometry simulation results, 72 727 messages per point. Model parameters: $R = 0.2169, \gamma = 0.80$. In this plot the experimental results are compared with the simulation. The plot closely matches the experimental data after some manual adjustments of the learning parameter γ.

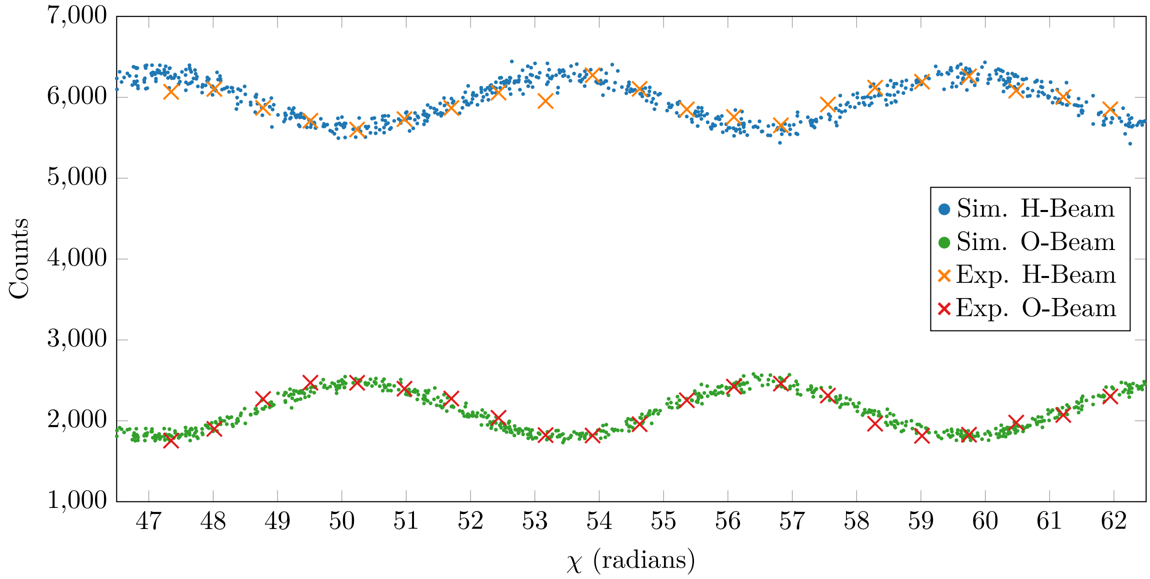

Figure 8: Neutron interferometry simulation results, 51 515 messages per point. Model parameters: $R = 0.1570, \gamma = 0.038$. In this figure the lower visibility of the data is clearly visible (the lines do not overlap anymore). This plot shows once again how the simulation data match the experimentally acquired data.

These simulations were done by using the same simulation setup as used in the theoretical neutron interferometry simulation, with R / T values matching the experiment. The learning parameter γ and the amount of messages required some manual tweaking to get right.

There clearly is a relation between the learning parameter and the visibility. The exact relationship between these parameters is not clear from the two sets of data, leaving it an interesting subject for further research.

5 Conclusion and discussion

This thesis demonstrates that the results of the experiments covered can be reproduced using event-based simulation methods. Event-based simulation turns out to be an effective method to describe the working of the mentioned experiments.

Especially in the neutron interferometry experiments, it doesn't really make sense to describe the process quantum mechanically in terms of interference of the neutrons, since we can emit the neutrons one-by-one and measure that they only take one of either routes. The simulations are on a one-by-one basis and the messengers can only take a single path while still producing the same results as the quantum theory and the experimental data. This makes the event-based approach a quite promising method.

The above certainly does not imply that Quantum Theory is invalid. As seen in this thesis, it still works perfectly to describe the experiments probabilistically. But, as the event-based approach is able to replicate both theory and experiment with great precision, the results cannot be ignored.

The event-based approach is an interesting way to look at the physics going on, because, as said in [3], it requires a different way of thinking, where the results of experiments are not based on the mathematical rules of quantum physics, but instead on discrete events and the relations between them: "This is a departure from the prevailing mode of thinking in theoretical physics, which assumes that the definite results which we observe are signatures of an underlying objective reality that is mathematical in nature." [3]

A lot of entirely different experiments remain that can be checked against this event-based approach. This being said, the event-based approach has in recent years been validated by numerous experiments and it will be interesting to see how this approach catches on.

All in all, the event-based approach appears to have a rosy future.

6 Acknowledgement

We are grateful to Dr. H. Lemmel for making the experimental data available to us. I would also like to thank H. De Raedt for introducing me to the interesting world of event-based simulations and showing the possibilities of this approach.

7 Bibliography

[1] H De Raedt, K De Raedt, and K Michielsen. "Event-based simulation of single-photon beam splitters and Mach-Zehnder interferometers". In: Europhysics Letters 69.6 (2005), pp. 861–867. ISSN: 0295-5075. DOI: 10.1209/epl/i2004-10443-7.

[2] H. De Raedt, F. Jin, and K. Michielsen. "Discrete-event simulation of neutron interferometry experiments". In: vol. 1508. 1. 2012, pp. 172–186. DOI: 10.1063/1.4773129.

[3] H De Raedt, F Jin, and K Michielsen. "Event-based simulation of neutron interferometry experiments". In: Quantum Matter 1.1 (2012), pp. 20–40.

[4] K. Michielsen and H. de Raedt. "Event-based simulation of quantum physics experiments". In: International Journal of Modern Physics C 25, 1430003 (July 2014). DOI: 10.1142/S0129183114300036.

[5] K. Michielsen, F. Jin, and H. De Raedt. "Event-Based Corpuscular Model for Quantum Optics Experiments". In: Journal of Computational and Theoretical Nanoscience 8.6 (2011), pp. 1052–1080. ISSN: 1546-1955. DOI: 10.1166/jctn.2011.1783.

[6] Jurriaan Pruis. Event-Based simulation source code. 2015. URL: https://github.com/jurriaan/event-based.

Appendix A: Simulation code

The C++ source code used to do the simulations is also available at https://github.com/jurriaan/event-based [6].

Listing 1: ../C++/src/event.cpp

```cpp
#include "types.h"
#include "event_component.h"
#include "observer.h"
#include "count_observer.h"
#include "beam_splitter.h"
#include <boost/progress.hpp>

const size_t runs    = 500;
const size_t repeats = 200000;
const double p0      = 0.5;
const double PI      = 3.14159265358979323846 3;
using namespace event_based;
using namespace std;

/**
 * Simulation code for the beam splitter experiment
 **/
int main ()
{
  event_based::BeamSplitter split(0.98);
  event_based::CountObserver obs(split, 0);
  std::random_device rd;
  std::mt19937_64 generator(rd());
  std::uniform_real_distribution<double> distribution;
  event_based::ComplexVector messages[2];
  boost::progress_display progress(runs, std::cerr);

  for (size_t i = 0; i < runs; ++i)
  {
    auto angle = distribution(generator) * 2 * PI;
    messages[0] = ComplexVector();
    messages[0] << cos(angle) << sin(angle);
    auto angle2 = distribution(generator) * 2 * PI;
    messages[1] = ComplexVector();
    messages[1] << cos(angle2) << sin(angle2);

    angle = (fmod(angle - angle2, 2 * PI));
    if (angle < 0)
      angle += 2*PI;
    angle *= 180.0/PI;

    for (size_t j = 0; j < repeats; ++j)
    {
      auto port = p0 > distribution(generator) ? 0 : 1;
      ComplexVector message = ComplexVector(messages[port]);
      split.process(port, message);
    }

    std::cout << angle << ", " << static_cast<double>(obs.count())/repeats <<
        std::endl;
    obs.reset();
    ++progress;
  }
  cerr << "done\n";
}
```

Listing 2: ../C++/src/event2.cpp

```
#include "event_based/types.h"
#include "event_based/event_component.h"
#include "event_based/observer.h"
#include "event_based/count_observer.h"
#include "event_based/connect_observer.h"
#include "event_based/phase_shifter.h"
#include "event_based/neutron_beam_splitter.h"
#include <boost/progress.hpp>

using namespace event_based;
using namespace std;
const size_t runs = 100;
const size_t repeats = 10000000;
const double PI = 3.14159265358979323846 3;
const double alpha = 0.99;
const Message input = {{0.5,0.5},{0.5,0.5}};

/**
 * Simulation code for the neutron interferometry experiment
 **/
int main ()
{
  std::random_device rd;
  std::mt19937_64 generator(rd());
  std::uniform_real_distribution<double> distribution(0.0, 2* PI);
  NeutronBeamSplitter bs0(alpha);
  NeutronBeamSplitter bs1(alpha);
  NeutronBeamSplitter bs2(alpha);
  NeutronBeamSplitter bs3(alpha);
  PhaseShifter xi0;
  PhaseShifter xi1;
  CountObserver obs1(bs3, 0);
  CountObserver obs2(bs3, 1);

  bs0(0) + bs1(0);
  bs1(1) + xi0(0);
  xi0(0) + bs3(1);

  bs0(1) + bs2(1);
  bs2(0) + xi1(0);
  xi1(0) + bs3(0);

  boost::progress_display progress(runs, std::cerr);

  for (size_t i = 0; i < runs; ++i)
  {
    auto angle = distribution(generator);
    xi0.set_phase(angle);
    auto angle2 = distribution(generator);
    xi1.set_phase(angle2);

    angle = (fmod(angle - angle2, 2 * PI));
    if(angle < 0)
      angle += 2*PI;
    angle *= 180.0/PI;

    Message message;
    for (size_t j = 0; j < repeats; ++j)
    {
      message = input;
      bs0.process(0, message);
    }
```

```

    std::cout << angle << ", " << obs1.count() << ", " << obs2.count() << std::endl;
    obs1.reset();
    obs2.reset();
    ++progress;
  }

  cerr << "done\n";
}
```

Listing 3: ../C++/src/event3.cpp

```
#include "event_based/types.h"
#include "event_based/event_component.h"
#include "event_based/observer.h"
#include "event_based/count_observer.h"
#include "event_based/connect_observer.h"
#include "event_based/phase_shifter.h"
#include "event_based/neutron_beam_splitter.h"
#include <boost/progress.hpp>

using namespace event_based;
using namespace std;
const size_t runs = 777;
const size_t repeats = 72727;
const double PI = 3.141592653589793238463;
const double alpha = 0.80;
const double reflection = 0.2169333831276301;
const Message input = {{0.5,0.5},{0.5,0.5}};

/**
 * Simulation code for the neutron interferometry experimental data comparison
     (first dataset)
 **/
int main ()
{
  std::random_device rd;
  std::mt19937_64 generator(rd());
  std::uniform_real_distribution<double> distribution(45.5, 62.5);
  NeutronBeamSplitter bs0(alpha, reflection);
  NeutronBeamSplitter bs1(alpha, reflection);
  NeutronBeamSplitter bs2(alpha, reflection);
  NeutronBeamSplitter bs3(alpha, reflection);
  PhaseShifter xi0;
  PhaseShifter xi1;
  CountObserver obs1(bs3, 0);
  CountObserver obs2(bs3, 1);

  bs0(0) + bs1(0);
  bs1(1) + xi0(0);
  xi0(0) + bs3(1);

  bs0(1) + bs2(1);
  bs2(0) + xi1(0);
  xi1(0) + bs3(0);

  boost::progress_display progress(runs, std::cerr);

  for (size_t i = 0; i < runs; ++i)
  {
    auto angle = distribution(generator);
    xi0.set_phase(angle);
    xi1.set_phase(0);

```

```
    Message message;
    for (size_t j = 0; j < repeats; ++j)
    {
      message = input;
      bs0.process(0, message);
    }

    std::cout << angle << ", " << obs1.count() << ", " << obs2.count() << std::endl;
    obs1.reset();
    obs2.reset();
    ++progress;
  }

  cerr << "done\n";
}
```

Listing 4: ../C++/src/event4.cpp

```
#include "event_based/types.h"
#include "event_based/event_component.h"
#include "event_based/observer.h"
#include "event_based/count_observer.h"
#include "event_based/connect_observer.h"
#include "event_based/phase_shifter.h"
#include "event_based/neutron_beam_splitter.h"
#include <boost/progress.hpp>

using namespace event_based;
using namespace std;
const size_t runs = 777;
const size_t repeats = 51515;
const double PI = 3.14159265358979323846З;
const double alpha = 0.038;
const double reflection = 0.15695631627438367;
const Message input = {{0.5,0.5},{0.5,0.5}};

/**
 * Simulation code for the neutron interferometry experimental data comparison
     (second dataset)
 **/
int main ()
{
  std::random_device rd;
  std::mt19937_64 generator(rd());
  std::uniform_real_distribution<double> distribution(46, 63);
  NeutronBeamSplitter bs0(alpha, reflection);
  NeutronBeamSplitter bs1(alpha, reflection);
  NeutronBeamSplitter bs2(alpha, reflection);
  NeutronBeamSplitter bs3(alpha, reflection);
  PhaseShifter xi0;
  PhaseShifter xi1;
  CountObserver obs1(bs3, 0);
  CountObserver obs2(bs3, 1);

  bs0(0) + bs1(0);
  bs1(1) + xi0(0);
  xi0(0) + bs3(1);

  bs0(1) + bs2(1);
  bs2(0) + xi1(0);
  xi1(0) + bs3(0);

  boost::progress_display progress(runs, std::cerr);

```

```
  for (size_t i = 0; i < runs; ++i)
  {
    auto angle = distribution(generator);
    xi0.set_phase(angle);
    xi1.set_phase(0);

    Message message;
    for (size_t j = 0; j < repeats; ++j)
    {
      message = input;
      bs0.process(0, message);
    }

    std::cout << angle << ", " << obs1.count() << ", " << obs2.count() << std::endl;
    obs1.reset();
    obs2.reset();
    ++progress;
  }

  cerr << "done\n";
}
```

Listing 5: ../C++/src/event_based/phase_shifter.cpp

```
#include "phase_shifter.h"
using namespace event_based;

PhaseShifter::PhaseShifter()
  : EventComponent()
{
  set_phase(0);
}

PhaseShifter::PhaseShifter(double phase)
  : EventComponent()
{
  set_phase(phase);
}

void PhaseShifter::set_phase(double phase)
{
  this->phase = phase;
  phase_shifter = std::polar(1.0, phase);
}

size_t PhaseShifter::handle_input(size_t port, event_based::Message &message)
{
  message *= phase_shifter;
  return port;
}
```

Listing 6: ../C++/src/event_based/phase_shifter.h

```
#include "event_component.h"
#ifndef EVENTBASEDPHASESHIFTER
#define EVENTBASEDPHASESHIFTER
namespace event_based
{
  class PhaseShifter : public EventComponent<1, 1>
  {
      double phase;
      complex_type phase_shifter;

    public:
```

```
    PhaseShifter();
    PhaseShifter(double phase);
    void set_phase(double phase);
    size_t handle_input(size_t port, event_based::Message &message) override;
  };
}
#endif
```

Listing 7: ../C++/src/event_based/beam_splitter.cpp

```
#include "beam_splitter.h"
#include <random>
using namespace event_based;

BeamSplitter::BeamSplitter(double alpha)
  : EventComponent(), alpha(alpha), tr_matrix(), rd(), generator(rd()),
      distribution(0.0, 1.0)
{
  state = random_state();
  setup_transformation_matrix();
}

BeamSplitter::BeamSplitter(double alpha, InternalState initial_state)
  : EventComponent(), alpha(alpha), state(initial_state), tr_matrix(), rd(),
      generator(rd()), distribution(0.0, 1.0)
{
  setup_transformation_matrix();
}

inline size_t BeamSplitter::handle_input(size_t port, event_based::Message &message)
{
  state = state * alpha;
  state[port] += (1 - alpha);
  auto vec = input_vector();
  vec = tr_matrix * vec;
  auto output = random_step(vec);

  message = arma::normalise(ComplexVector({vec[0 + output], vec[2 + output]}));

  return output;
}

inline size_t BeamSplitter::random_step(ComplexVector vec)
{
  auto first = std::abs(vec[1]);
  auto sec = std::abs(vec[3]);
  auto z = first * first + sec * sec;
  auto r = distribution(generator);
  return z > r ? 1 : 0;
}

inline InternalState BeamSplitter::random_state()
{
  double total = distribution(generator);
  double part = distribution(generator);

  return { total * part, total *(1 - part) };
}

inline InputVector BeamSplitter::input_vector() const
{
  auto s0 = sqrt(state[0]);
  auto s1 = sqrt(state[1]);

```

```
  return { last_message[0][0] * s0,
           last_message[1][0] * s1,
           last_message[0][1] * s0,
           last_message[1][1] * s1 };
}

void BeamSplitter::setup_transformation_matrix()
{
  tr_matrix = ComplexMatrix();
  tr_matrix << 1.0 <<  0.0 <<  0.0 << -1.0 << arma::endr
            << 0.0 <<  1.0 << -1.0 <<  0.0 << arma::endr
            << 0.0 <<  1.0 <<  1.0 <<  0.0 << arma::endr
            << 1.0 <<  0.0 <<  0.0 <<  1.0;
  tr_matrix *= pow(0.5, 0.5);
}
```

Listing 8: ../C++/src/event_based/beam_splitter.h

```
#include "event_component.h"

#ifndef EVENTBASEDBEAMSPLITTER
#define EVENTBASEDBEAMSPLITTER

namespace event_based
{
  typedef Vector InternalState;
  typedef ComplexVector InputVector;
  class BeamSplitter : public EventComponent<2, 2>
  {
      double alpha;
      InternalState state;
      std::random_device rd;
      std::mt19937_64 generator;
      std::uniform_real_distribution<double> distribution;

    public:
      explicit BeamSplitter(double alpha = 0.98);
      explicit BeamSplitter(double alpha, InternalState initial_state);
      size_t handle_input(size_t port, event_based::Message &message) override;

    protected:
      ComplexMatrix tr_matrix;
      virtual void setup_transformation_matrix();

    private:
      InternalState random_state();
      InputVector input_vector() const;
      size_t random_step(ComplexVector vector);
  };
}

#endif
```

Listing 9: ../C++/src/event_based/neutron_beam_splitter.cpp

```
#include "neutron_beam_splitter.h"
using namespace event_based;
NeutronBeamSplitter::NeutronBeamSplitter(double alpha, double reflection)
  : BeamSplitter(alpha), reflection(reflection)
{
  setup_transformation_matrix();
}

```

```
NeutronBeamSplitter::NeutronBeamSplitter(double alpha, double reflection,
    InternalState initial_state)
  : BeamSplitter(alpha, initial_state), reflection(reflection)
{
  setup_transformation_matrix();
}

void NeutronBeamSplitter::setup_transformation_matrix()
{
  complex_type t(sqrt(1 - reflection), 0);
  complex_type r(0, sqrt(reflection));
  tr_matrix = ComplexMatrix();
  tr_matrix <<   t <<   r <<  0.0 <<  0.0 << arma::endr
            <<   r <<   t <<  0.0 <<  0.0 << arma::endr
            << 0.0 << 0.0 <<    t <<    r << arma::endr
            << 0.0 << 0.0 <<    r <<    t;
}
```

Listing 10: ../C++/src/event_based/neutron_beam_splitter.h

```
#include "beam_splitter.h"

#ifndef EVENTBASEDNEUTRONBEAMSPLITTER
#define EVENTBASEDNEUTRONBEAMSPLITTER

namespace event_based
{
  class NeutronBeamSplitter : public BeamSplitter
  {
      double reflection;

    public:
      explicit NeutronBeamSplitter(double alpha = 0.98, double reflection = 0.2);
      explicit NeutronBeamSplitter(double alpha, double reflection, InternalState
          initial_state);

    protected:
      void setup_transformation_matrix() override;
  };
}

#endif
```

Listing 11: ../C++/src/event_based/count_observer.h

```
#include "observer.h"

#ifndef EVENTBASEDCOUNTOBSERVER
#define EVENTBASEDCOUNTOBSERVER
namespace event_based
{
  class CountObserver : public Observer
  {
      size_t d_count = 0;
    public:
      using Observer::Observer;
      void notify(event_based::Message &message) override;
      size_t count() const;
      void reset();
  };

  inline void CountObserver::notify(event_based::Message &message)
  {
    ++d_count;
```

```
  }

  inline size_t CountObserver::count() const
  {
    return d_count;
  }

  inline void CountObserver::reset()
  {
    d_count = 0;
  }
}

#endif
```

Listing 12: ../C++/src/event_based/connect_observer.cpp

```
#include "connect_observer.h"
#include "event_component.h"

namespace event_based
{
  ConnectObserver::ConnectObserver(Observable &component, size_t port,
      AbstractEventComponent &observer, size_t observer_port)
    : Observer(component, port), observer(observer), observer_port(observer_port)
  {
  }

  inline void ConnectObserver::notify(event_based::Message &message)
  {
    observer.process(observer_port, message);
  }

  ConnectObserver *operator+(ConnectableEventComponent &&a,
      ConnectableEventComponent &&b)
  {
    return new ConnectObserver(a.component, a.port, b.component, b.port);
  }
}
```

Listing 13: ../C++/src/event_based/connect_observer.h

```
#include "observer.h"
#include "abstract_event_component.h"

#ifndef EVENTBASEDCONNECTOBSERVER
#define EVENTBASEDCONNECTOBSERVER
namespace event_based
{
  class AbstractEventComponent;
  class ConnectObserver : public Observer
  {
      AbstractEventComponent &observer;
      size_t observer_port;
    public:
      ConnectObserver(Observable &component, size_t port, AbstractEventComponent
          &observer, size_t observer_port);
      void notify(event_based::Message &message) override;
  };

}

#endif
```

Listing 14: ../C++/src/event_based/event_component.cpp

```
#include "event_component.h"

using namespace event_based;
ConnectableEventComponent AbstractEventComponent::operator()(size_t port)
{
  return ConnectableEventComponent(*this, port);
}


ConnectableEventComponent::ConnectableEventComponent(AbstractEventComponent &component, size_t port)
  : component(component), port(port)
{
}
```

Listing 15: ../C++/src/event_based/event_component.h

```
#include "observer.h"
#include "abstract_event_component.h"
#include "connect_observer.h"

#ifndef EVENTBASEDEVENTCOMPONENT
#define EVENTBASEDEVENTCOMPONENT
namespace event_based
{
  class ConnectableEventComponent
  {
      AbstractEventComponent &component;
      size_t port;

    public:
      explicit ConnectableEventComponent(AbstractEventComponent &component, size_t port);
      friend ConnectObserver *operator+(ConnectableEventComponent &&a, ConnectableEventComponent &&b);
  };

  template <size_t inputs, size_t outputs>
  class EventComponent : public AbstractEventComponent
  {
      event_based::Observer *observer[outputs];

    public:
      EventComponent();

      void attach(event_based::Observer *observer, size_t port) override;
      void detach(size_t port) override;
      void process(size_t port, event_based::Message &message) override;
      virtual ~EventComponent();

    protected:
      event_based::Message last_message[inputs];
      virtual size_t handle_input(size_t port, event_based::Message &message) = 0;
  };

  template <size_t inputs, size_t outputs>
  void EventComponent<inputs, outputs>::process(size_t port, Message &message)
  {
    last_message[port] = message;

    auto output = handle_input(port, message);
    if(observer[output])
```

```
      observer[output]->notify(message);
  }

  template <size_t inputs, size_t outputs>
  void EventComponent<inputs, outputs>::detach(size_t port)
  {
    this->observer[port] = 0;
  }

  template <size_t inputs, size_t outputs>
  void EventComponent<inputs, outputs>::attach(event_based::Observer *observer,
      size_t port)
  {
    this->observer[port] = observer;
  }

  template <size_t inputs, size_t outputs>
  EventComponent<inputs, outputs>::EventComponent()
  {
    for (size_t i = 0; i < inputs; ++i)
      last_message[i] = Message(2, arma::fill::eye);
    for (size_t i = 0; i < outputs; ++i)
      observer[i] = 0;
  }

  template <size_t inputs, size_t outputs>
  EventComponent<inputs, outputs>::~EventComponent()
  {
    for (size_t i = 0; i < outputs; ++i)
      delete observer[i];
  }
}
#endif
```

Listing 16: ../C++/src/event_based/abstract_event_component.h

```
#include "observable.h"
#ifndef EVENTBASEDABSTRACTEVENTCOMPONENT
#define EVENTBASEDABSTRACTEVENTCOMPONENT
namespace event_based
{
  class ConnectableEventComponent;
  class AbstractEventComponent : public Observable
  {
    public:
      virtual void process(size_t port, event_based::Message &message) = 0;
      ConnectableEventComponent operator()(size_t port);
  };

}
#endif
```

Listing 17: ../C++/src/event_based/observable.h

```
#ifndef EVENTBASEDOBSERVABLE
#define EVENTBASEDOBSERVABLE
namespace event_based
{
  class Observer;
  class Observable
  {
    public:
      virtual void attach(event_based::Observer *observer, size_t port) = 0;
      virtual void detach(size_t port) = 0;
```

```
  };
}
#endif
```

Listing 18: ../C++/src/event_based/observer.cpp

```
#include <cstddef>
#include "observer.h"

namespace event_based
{
  Observer::Observer(Observable &component, size_t port)
    : observable(component)
  {
    observable_port = port;
    observable.attach(this, port);
  }

  Observer::~Observer()
  {
    observable.detach(observable_port);
  }
}
```

Listing 19: ../C++/src/event_based/observer.h

```
#include "types.h"
#include "observable.h"

#ifndef EVENTBASEDOBSERVER
#define EVENTBASEDOBSERVER
namespace event_based
{
  class Observer
  {
    public:
      explicit Observer(Observable &component, size_t port);
      virtual void notify(event_based::Message &message) = 0;
      virtual ~Observer();
    protected:
      size_t observable_port;
      Observable &observable;
  };
}
#endif
```

Listing 20: ../C++/src/event_based/types.h

```
#include <armadillo>

#ifndef EVENTBASEDTYPES
#define EVENTBASEDTYPES

namespace event_based
{
  typedef std::complex<double> complex_type;
  typedef arma::cx_vec Message;
  typedef arma::vec Vector;
  typedef arma::cx_vec ComplexVector;
  typedef arma::cx_mat ComplexMatrix;
}

#endif
```