Hochschule Fulda
University of Applied Sciences

Software Development Project

# Acceleration of Quantum Field Theory Calculations using Machine Learning Packages

*Author:*
Rohan Deo (650271)
Mahbubur Rahman (250154)
Swetaketu Majumder (450237)

*Submitted To:*
Prof. Dr. Alexander Gepperth

October 8, 2018

# Contents

**Abstract**

Quantum field theory calculations are complicated and time consuming if they are done using procedural programming and it is obviously impossible to perform such complex calculations using pen and paper. Today, with the advent of machine learning packages, complex calculations can be performed with ease by harnessing the power of a graphics processing unit. This paper investigates the possibility of using such a machine learning package to compute action for a large Euclidean space time history in a short amount of time. And further, the paper also minimizes the said histories to find histories that are the most probable.

**Keywords:** Science, Physics, Quantum Mechanics, Quantum Field Theory, Acceleration, Action, Lattice, Free Scalar Field, Convolution, Fast Fourier Transform FFT, Gradient Descent Optimizer, Machine Learning, TensorFlow, Complicated, Python, numpy, scipy

# 1 Mission Statement

This project was aimed at proving that machine learning packages can be used to accelerate complex calculations of quantum field theory. In particular, it tried to compute the action of a randomly generated 4-Dimensional Euclidean space-time history and then find histories that are the most probable by minimizing said action.

Action ($S$) is an attribute of the dynamics of a physical system from which the equations of motion of the system can be derived. For the purposes of this project, the following equation was used:

$$\partial^2\phi(x) \to \sum_{\mu}(\phi(x+e_\mu)+\phi(x-e_\mu)-2\phi(x)) = \sum_{\substack{+\\-}\mu}(\phi(x+e_\mu)+\phi(x-e_\mu))-2d\phi(x) \quad (1)$$

where $\phi$ is now dimensionless and $d$ is the dimensionality of space-time [10].

Interestingly, this complex formula can be solved by basic matrix manipulation in a computer program. And in the present era of super-fast computational power, it is possible to do such matrix manipulations on large matrices very efficiently and quickly.

# 2 Planning and Proposed Solutions

For action $S$ calculation, we have used Python as programming language, machine learning packages of TensorFlow for faster, error-free and parallel calculation of complex mathematical equations and Matplotlib to plot results for comparison and also view the final outcome in 2D plane. In addition we have used **Circular Shift** of axis using **Roll** operation or **Circular Convolution** using **Fast Fourier Transform (FFT)** for action $S$ calculation.

## 2.1 Circular Shift using Roll

In combinatorial mathematics, a circular shift is the operation of rearranging the entries in a tuple, either by moving the final entry to the first position, while shifting all other entries to the next position, or by performing the inverse operation. A circular shift is a special kind of cyclic permutation, which in turn is a special kind of permutation. Formally, a circular shift is a permutation $\sigma$ of the $n$ entries in the tuple such that either [3]

$$\sigma(i) \equiv (i+1) \tag{2}$$

or

$$\sigma(i) \equiv (i-1) \tag{3}$$

modulo $n$, for all entries $i = 1, ..., n$

The result of repeatedly applying circular shifts to a given tuple are also called the circular shifts of the tuple. For example, repeatedly applying circular shifts to the four-tuple $(a, b, c, d)$ successively gives [3]

- $(d, a, b, c)$,
- $(c, d, a, b)$,
- $(b, c, d, a)$,
- $(a, b, c, d)$ (the original four-tuple),

and then the sequence repeats; this four-tuple therefore has four distinct circular shifts. However, not all $n$-tuples have $n$ distinct circular shifts. For instance, the 4-tuple $(a, b, a, b)$ only has 2 distinct circular shifts. In general the number of circular shifts of an $n$-tuple could be any divisor of $n$, depending on the entries of the tuple [3].

In computer programming, a circular shift (or bitwise rotation) is a shift operator that shifts all bits of its operand. Unlike an arithmetic shift, a circular shift

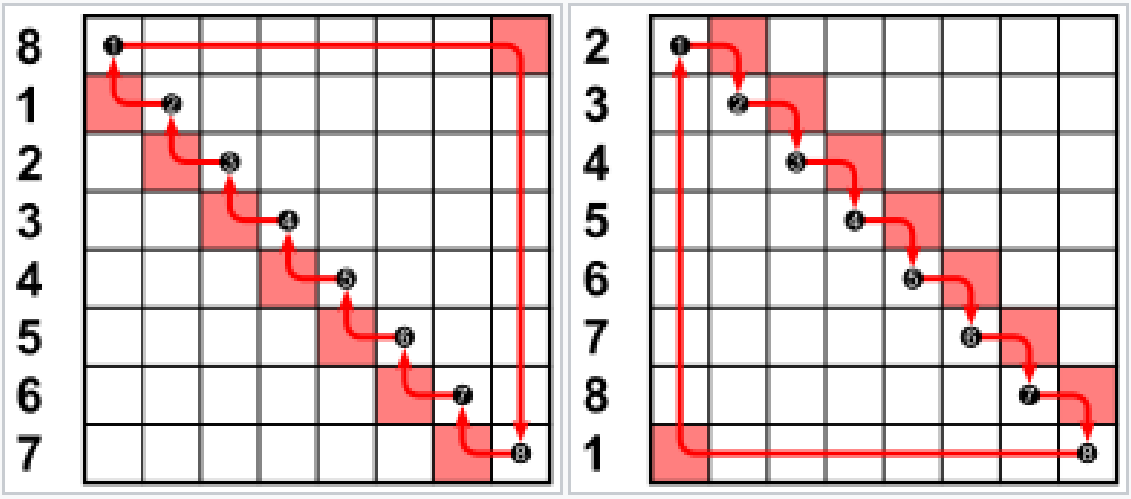


Figure 1: Matrices of 8-element circular shifts to the left and right

does not preserve a number's sign bit or distinguish a number's exponent from its significand (sometimes referred to as the mantissa). Unlike a logical shift, the vacant bit positions are not filled in with zeros but are filled in with the bits that are shifted out of the sequence [3].

## 2.2 Circular Convolution

The circular convolution, also known as cyclic convolution, of two aperiodic functions (i.e. Schwartz functions) occurs when one of them is convolved in the normal way with a periodic summation of the other function. That situation arises in the context of the circular convolution theorem. The identical operation can also be expressed in terms of the periodic summations of both functions, if the infinite integration interval is reduced to just one period. That situation arises in the context of the discrete-time Fourier transform (DTFT) and is also called periodic convolution. In particular, the DTFT of the product of two discrete sequences is the periodic convolution of the DTFTs of the individual sequences [2].

Let $x$ be a function with a well-defined periodic summation, $x_T$, where [2]:

$$x_T(t) = \sum_{k=-\infty}^{\infty} x(t - kT) = \sum_{k=-\infty}^{\infty} x(t + kT) \tag{4}$$

If $h$ is any other function for which the convolution $x_T * h$ exists, then the convolution $x_T * h$ is periodic and identical to [2]:

$$(x_T * h)(t) = \int_{-\infty}^{\infty} h(\tau).x_T(t-\tau)d\tau \tag{5}$$

$$= \int_{t_0}^{t_0+T} h_T(\tau).x_T(t-\tau)d\tau \tag{6}$$

where $t_o$ is an arbitrary parameter and $h_T$ is a periodic summation of $h$. The second integral is called the periodic convolution of functions $x_T$ and $h_T$ and is sometimes normalized by $1/T$. When $x_T$ is expressed as the periodic summation of another function, $x$, the same operation may also be referred to as a circular convolution of functions $h$ and $x$ [2].

### 2.2.1 Discrete Sequences

Similarly, for discrete sequences and period $N$, we can write the circular convolution of functions $h$ and $x$ as [2]:

$$(x_N * h)[n] = \sum_{m=-\infty}^{\infty} h[m].x_N[n-m] \tag{7}$$

$$= \sum_{m=-\infty}^{\infty} (h[m]. \sum_{K=-\infty}^{\infty} x[n-m-kN]) \tag{8}$$

For the special case that the non-zero extent of both $x$ and $h$ are $\leq N$, this is reducible to matrix multiplication where the kernel of the integral transform is a circulant matrix [2].

### 2.2.2 Example

A case of great practical interest is illustrated in the figure. The duration of the $x$ sequence is $N$ (or less), and the duration of the $h$ sequence is significantly less. Then many of the values of the circular convolution are identical to values of $x * h$, which is actually the desired result when the $h$ sequence is a finite impulse response (FIR) filter. Furthermore, the circular convolution is very efficient to compute, using a fast Fourier transform (FFT) algorithm and the circular convolution theorem [2].

There are also methods for dealing with an $x$ sequence that is longer than a practical

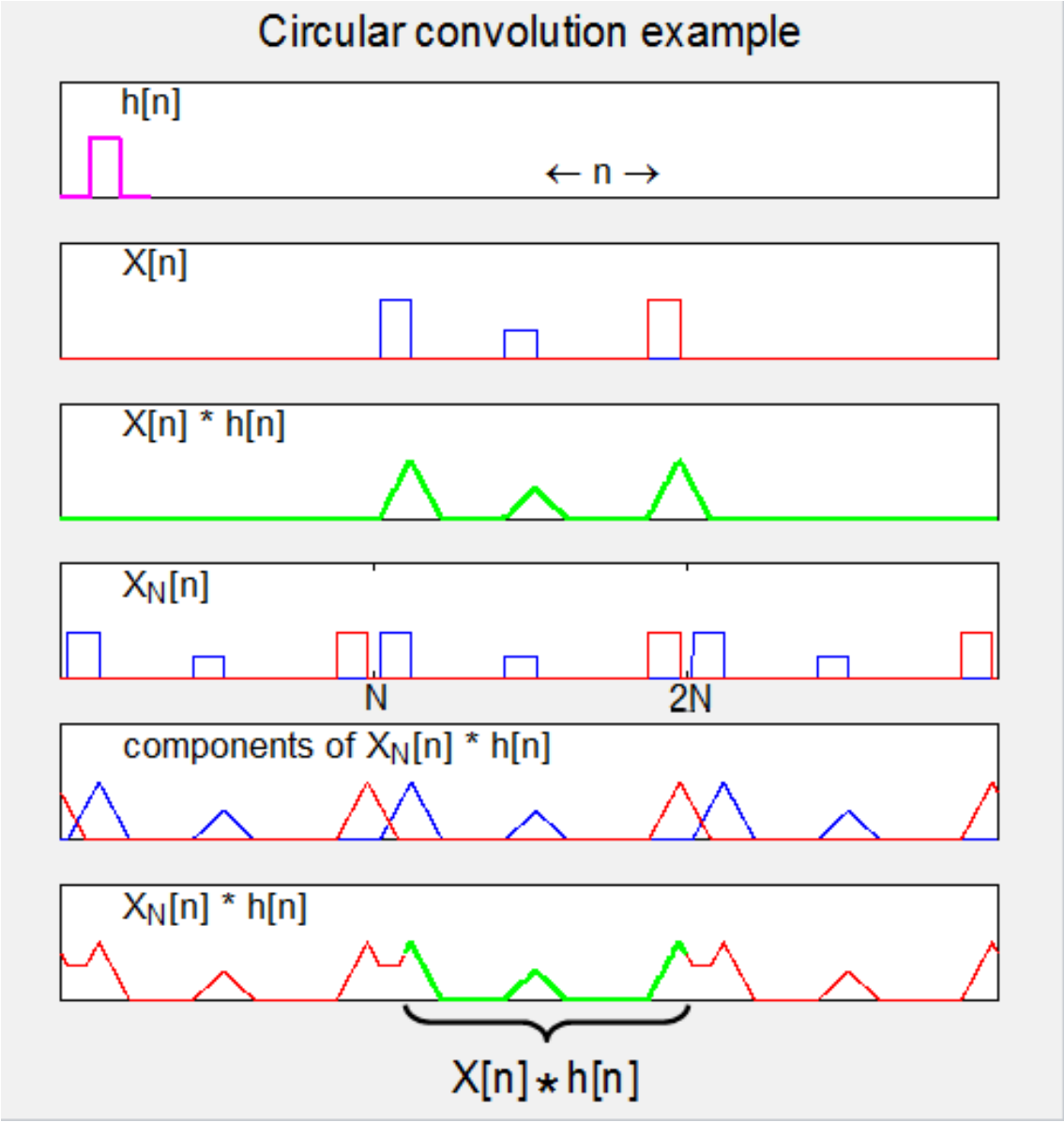


Figure 2: Circular Convolution

value for $N$. The sequence is divided into segments (blocks) and processed piecewise. Then the filtered segments are carefully pieced back together. Edge effects are eliminated by overlapping either the input blocks or the output blocks. To help explain and compare the methods, we discuss them both in the context of an $h$ sequence of length 201 and an FFT size of $N = 1024$ [2].

## 2.3 Circular Convolution using FFT

Suppose we want to convolve a signal of $N$ with a filter of size $M$. As we saw in the introduction, a direction implementation of the convolution product, using nested for loops requires an order of $N.\ M$ operations. It is possible to speed up this computation by using the Fourier Transform. This relies on the the convolution theorem : the circular convolution product of two signals equals the inverse Fourier transform of the product of the fourier transforms of the signals [1]:

### 2.3.1 Theorem 1

Let $f$ and $g$ be two discrete sequences of respectively length $N$ and $M$ and $P \epsilon N$. The inverse fourier transform of the product of the fourier transform of $fP$ and $gP$ equals the circular convolution of $f$ and $g$ modulo $P$ [1]:

$$\forall k \in [0, P-1], (f\ {}_P g)[k] = IFT[FT[f_P].FT[g_P]][k] \tag{9}$$

To show this, let's compute the fourier transform of the circular convolution modulo $P$ of $f$ and $g$ [1]:

$$\forall k \in [0, P-1], FT[f\ {}_P g][k] = \sum_{i=0}^{P-1}((f\ {}_P g)[i] w_P^{ki}) \tag{10}$$

$$= \sum_{i=0}^{P-1}\sum_{j=0}^{P-1} f_P[j] g_p[i-j] w_P^{ki} \tag{11}$$

$$= \sum_{j=0}^{P-1} f_P[j] w_P^{kj} \sum_{i=0}^{P-1} g_p[i-j] w_P^{k(i-j)} \tag{12}$$

Since the signal $g_P$ is periodic of period $P$, then [1]

$$\forall j \in [0, P-1], \sum_{i=0}^{P-1} g_p[i-j] w_P^{k(i-j)} = \sum_{i=0}^{P-1} g_P[i] w_P^{ki} \tag{13}$$

Which leads to [1]:

$$\forall k \in [0, P-1], FT[f \ _P g][k] = \sum_{j=0}^{P-1} f_P[j] w_P^{kj} \sum_{i=0}^{P-1} g_p[i-j] w_P^{k(i-j)} \tag{14}$$

$$= \sum_{j=0}^{P-1} f_P[j] w_P^{kj} \sum_{i=0}^{P-1} g_P[i] w_P^{ki} \tag{15}$$

$$= FT[f_P][k].FT[g_P][k] \tag{16}$$

By taking the inverse fourier transform of both sides, we get the result that we wanted to demonstrate. The circular convolutions are faster with the FFT based implementation for kernel's size at least $8 \times 8$ independently of the lattice size [1].

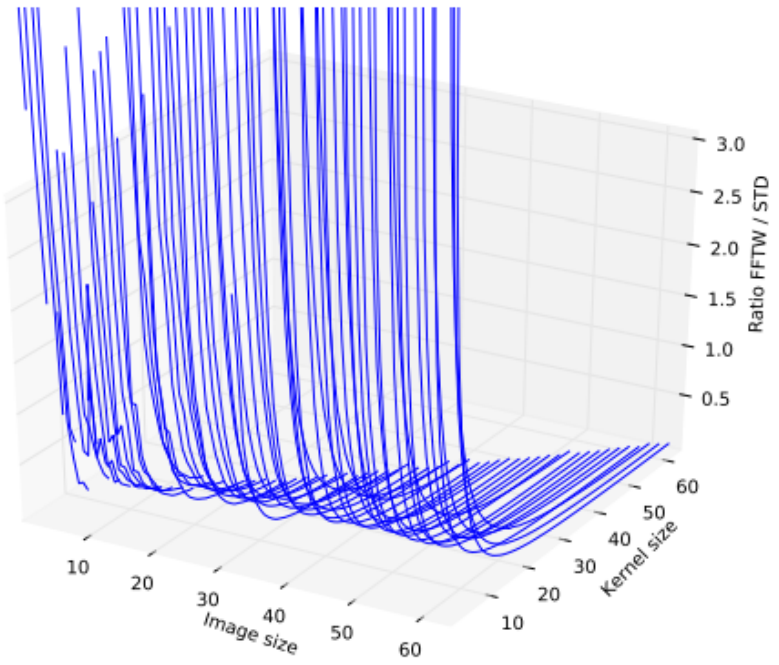


Figure 3: Circular convolutions : circular convolutions are faster with the FFT based implementation if the kernel's size is at least $8 \times 8$. [1]

In the following, the description of proposed technologies are given.

## 2.4 Python

Python is an interpreted high-level programming language for general-purpose programming. Python features a dynamic type system and automatic memory management. It supports multiple programming paradigms, including object-oriented, imperative, functional and procedural, and has a large and comprehensive standard library. Python interpreters are available for many operating systems [5].

Python is used in many scientific researches and projects. The NumPy and SciPy are two well defined scientific libraries for scientific research works. In this project, we use NumPy and SciPy for calculation purposes.

NumPy is the fundamental package for scientific computing with Python. It contains among other things [9]:

- a powerful N-dimensional array object
- sophisticated (broadcasting) functions
- tools for integrating C/C++ and Fortran code
- useful linear algebra, Fourier transform, and random number capabilities

Besides its obvious scientific uses, NumPy can also be used as an efficient multi-dimensional container of generic data. Arbitrary data-types can be defined. This allows NumPy to seamlessly and speedily integrate with a wide variety of databases [9].

The SciPy library is one of the core packages that make up the SciPy stack. It provides many user-friendly and efficient numerical routines such as routines for numerical integration and optimization [7].

## 2.5 TensorFlow

TensorFlow is an open source software library for high performance numerical computation. Its flexible architecture allows easy deployment of computation across a variety of platforms (CPUs, GPUs, TPUs), and from desktops to clusters of servers to mobile and edge devices. Originally developed by researchers and engineers that comes with strong support for machine learning and deep learning and the flexible numerical computation core is used across many other scientific domains [8].

## 2.6 Matplotlib

Matplotlib is a Python 2D plotting library which produces publication quality figures in a variety of hard copy formats and interactive environments across platforms. Matplotlib tries to make easy things easy and hard things possible. We can generate plots, histograms, power spectra, bar charts, errorcharts, scatterplots, etc., with just a few lines of code. For simple plotting the pyplot module provides a MATLAB-like interface, particularly when combined with IPython. For the power user, we have full control of line styles, font properties, axes properties, etc, via an object oriented interface or via a set of functions familiar to MATLAB users [6].

## 2.7 Proposed Solutions

The proposed solutions for $S$ calculation are mentioned bellow:

- Brute Force Approach: the action will be calculated using normal brute force approach and the we measure the time taken for calculating the action.
- Roll: using this way, the 4D matrix is rolled from left to right and vise verse. Elements that roll beyond the last position are re-introduced at the first.
- Convolution: it is the process of adding each element of the 4D matrix to its local neighbors, weighted by the kernel. Here we will use wrapping or circular convolution to calculate the action $S$.
- Gradient Descent Optimizer: it is a first-order iterative optimization algorithm for finding the minimum of a function. To find a local minimum of a function using gradient descent, one takes steps proportional to the negative of the gradient (or approximate gradient) of the function at the current point. If instead one takes steps proportional to the positive of the gradient, one approaches a local maximum of that function; the procedure is then known as gradient ascent [4].

# 3 Implementation

As a first step, we computed action using a brute-force approach and used the value of action and the time taken as a references for comparison with the results of the upcoming methods. Once we had our control values, we moved forward with the implementation of roll.

## 3.1 Roll

We used the Roll operation in numpy and tensorflow to get the desired matrices that simply needed to be added together to get the action that we needed. According to the formula previously introduced, all we needed to do circular shift of elements in each dimension by one index right and then one index left and add all of such matrices with the mass term.

```
tLeft = np.roll(arr[n,:,:,:], -1, axis = 1)
tRight = np.roll(arr[n,:,:,:], 1, axis = 1)

xLeft = np.roll(arr[n,:,:,:], -1, axis = 0)
xRight = np.roll(arr[n,:,:,:], 1, axis = 0)

yLeft = np.roll(arr[n,:,:,:], -1, axis = 2)
yRight = np.roll(arr[n,:,:,:], 1, axis = 2)

zLeft = np.roll(arr[n,:,:,:], -1, axis = 3)
zRight = np.roll(arr[n,:,:,:], 1, axis = 3)

common =  arr[n,:,:,:] * (-8 + (CONST_m**2/2) * arr[n,:,:,:])

total = tLeft + tRight + xLeft + xRight + yLeft + yRight + zLeft
+ zRight + common
```

## 3.2 Convolution

For convolution, we chose $[1, -2, 1]$ as the kernel. And then, we firstly used the convolve function in the scipy.ndimage package. This function allows for a circular convolution by using "wrap" mode and this was particularly helpful because it meant that we could include the filed values on the boundaries, just as in roll. We convolved along each axis and then added the resultant matrices and the common mass term to get the action.

```
S = 0
convarrt = ndimg.filters.convolve(arr[n,:,:,:],
    self.kernels.CONST_ConvKernelt, mode = 'wrap')
convarrx = ndimg.filters.convolve(arr[n,:,:,:],
    self.kernels.CONST_ConvKernelx, mode = 'wrap')
convarry = ndimg.filters.convolve(arr[n,:,:,:],
```

```
        self.kernels.CONST_ConvKernely, mode = 'wrap')
    convarrz = ndimg.filters.convolve(arr[n,:,:,:],
        self.kernels.CONST_ConvKernelz, mode = 'wrap')
    common =  arr[n,:,:,:] * ((CONST_m**2/2) * arr[n,:,:,:])
    S = (convarrt + convarrx + convarry + convarrz + common).sum()
```

But since our aim was to use a machine learning package for the calculations and there were no appropriate convolve function readily available for tensorflow, we had to use fast fourier transform (FFT) and inverse FFT (IFFT) to compute the convolved matrices. The IFFT of the product of the FFT of the whole matrix and the FFT of the convolution kernel gave the convolved matrix, which was added with the mass term to get the action.

```
arr_fft = tf.fft(tf.complex(self.arrVar, imag_zeros), name="arr_fft")
conv_axis = tf.real(tf.ifft(tf.multiply(arr_fft, kernel_fft)),
    name="op_convolve_axis")
common = tf.multiply(tf.multiply(self.arrVar, self.arrVar),
    tf.divide(tf.multiply(self.mass_tf, self.mass_tf), 2), name="op_common")
conv_action = tf.divide(tf.reduce_sum(tf.add(conv_axis, common)),
    self.arrSize, name="graph_conv_action")
```

# 4 Comparison among Solutions

The comparison among different process for calculation the action $S$, is plotted using Matplotlib bar graph as shown below. This figure was plotted on a machine with no GPU.

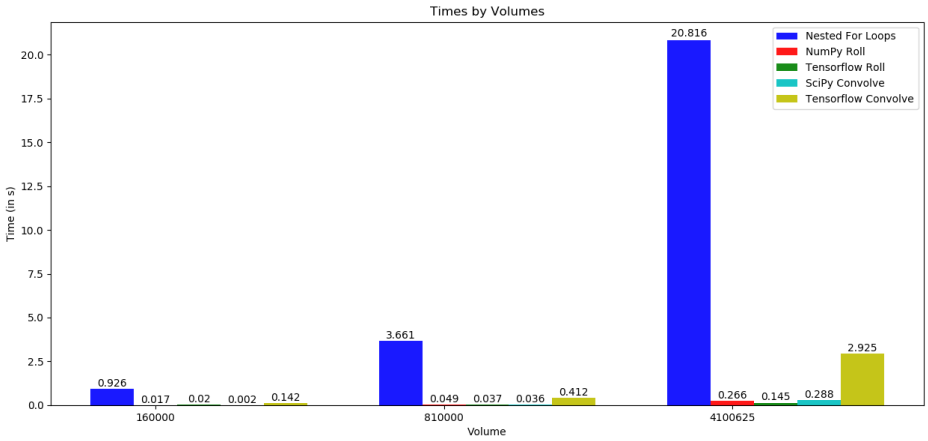


Figure 4: Time taken by different methods on CPU

Since tensorflow benefits largely from the power of a GPU, we ran the program on a machine that has GPU cores and the results are indicated in the graph below.

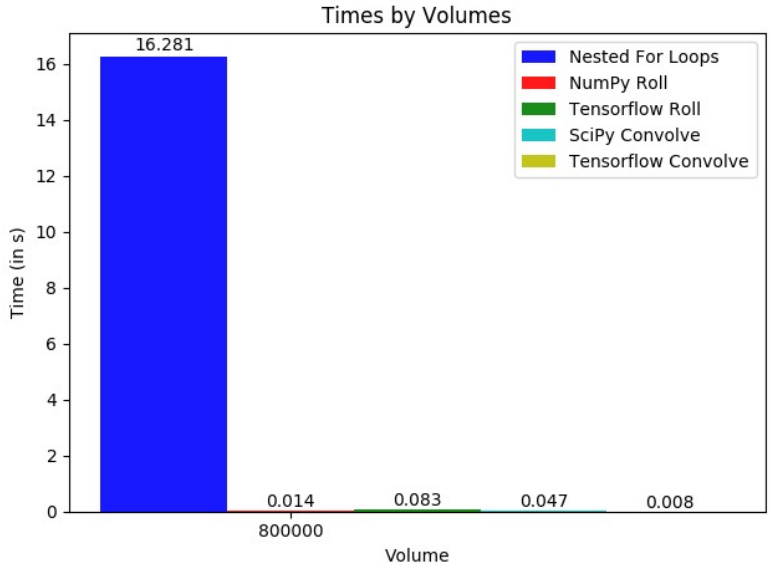


Figure 5: Time taken by different methods on GPU

From these figures, it is obvious that machine learning packages such as tensorflow can be used on a computer with GPUs to accelerate complex calculations of quantum field theory by a huge degree.

# 5 Final Outcome

Gradient descent optimizer was run until a minima was reached to compute the most probable space-time histories. Once it finished running, the field values were plotted by slicing the lattices. The values of the initial random histories and the resultant minimized histories were plotted side-by-side for better comparison.

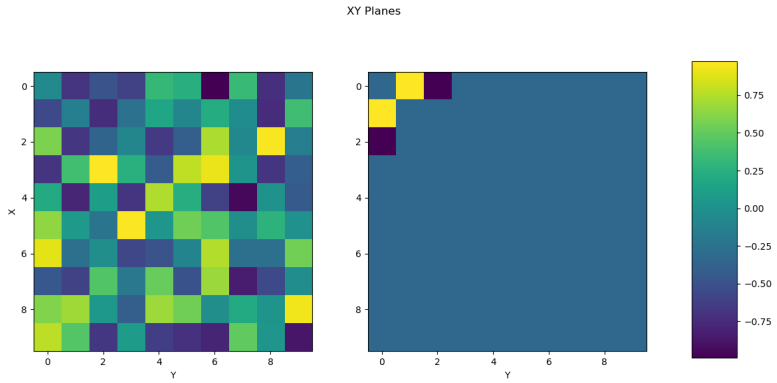


Figure 6: Final Outcome - XY Planes

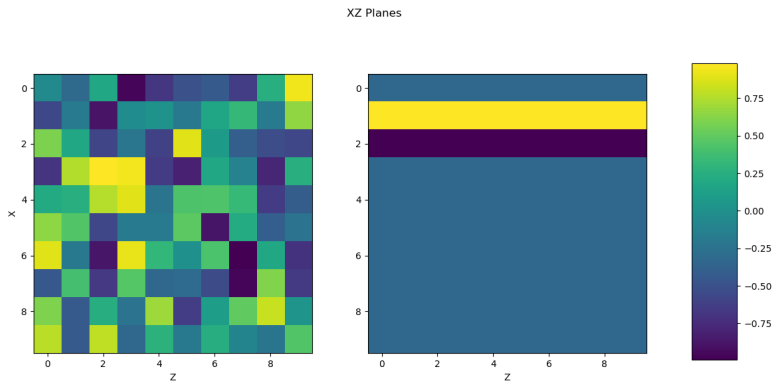


Figure 7: Final Outcome - XZ Planes

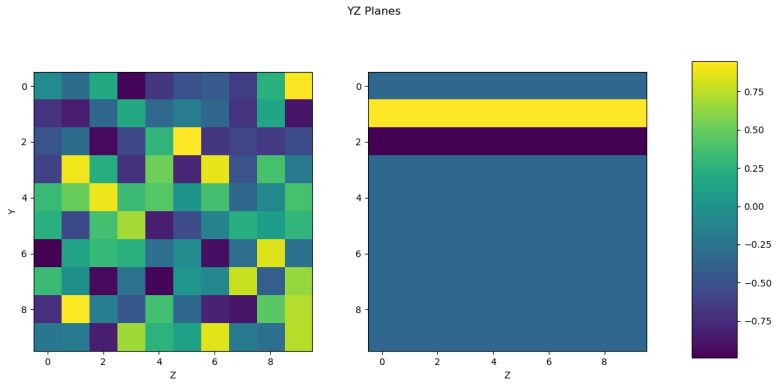


Figure 8: Final Outcome - YZ Planes

# 6 Challenges Faced during Implementation

The following challenges we faced during the implementation:

- For convolution using TensorFlow, we tried to implement using tf.nn.convolution, but this function gives us the result of linear convolution and we needed circular convolution.
- After implementing the convolution using **fft** and **ifft**, we noticed that the calculation time did not meet our expectations. So we analyzed the implementation and updated the implementation by using TensorFlow variables, sparse matrix for kernel and minimized redundant computations.

# 7 Project Architecture & Code Documentation

The source code is available in a GitLab repository, the url for which is:

*https://gitlab.com/swetaketum/GSD_SDP_SS2018.git.*

The program is divided into two main parts -

1. A POC (Proof of Concept) that runs all the methods and shows a chart comparing the time taken for all the methods. This chart confirms that the implementation for convolution using fft and ifft in tensorflow is indeed the fastest when run on a computer with GPUs.

2. The main program that runs only tensorflow code and runs gradient descent optimizer to compute the most probable space-time histories.

It is important to mention here that both of these parts use the same code for tensorflow convolution. The project structure is as follows:

```
/
├── POC
│   ├── ConvKernels.py
│   ├── LoopOps.py
│   ├── PlotGraphs.py
│   ├── PyOps.py
│   ├── TensorflowOps.py
│   └── main.py
├── ActionCalc.py
├── Plotter.py
└── run.py
```

To run the program, the following terminal commands need to be used:

1. For POC - main.py [-h] [-n NOH] [-dim dim1 dim2 dim3 dim4] [-f min max] [-dl] [-dpr] [-dpc] [-dtr] [-dtc] [-p] [-sc]
2. For Main program - run.py [-h] [-n NOH] [-dim dim1 dim2 dim3 dim4] [-f min max] [-p] [-sc]

The usage for the arguments are as follows:

1. **-h, –help** = show this help message and exit
2. **-n NOH, –noh NOH** = number of random space time histories to generate
3. **-dim dim1 dim2 dim3 dim4, –dimensions dim1 dim2 dim3 dim4** =¿ sizes for each of the 4 dimensions
4. **-f min max, –field min max** = field MIN and MAX
5. **-dl, -disable-loop** = disable loop calculation
6. **-dpr, -disable-python-roll** = disable python roll calculation
7. **-dpc, -disable-python-convolve** = disable scipy convolve calculation
8. **-dtr, -disable-tensorflow-roll** = disable tensorflow roll calculation
9. **-dtc, -disable-tensorflow-convolve** = disable tensorflow convolve calculation

10. **-p, -printall** = print each action while calculating
11. **-sc, -showchart** = plot chart in the end

Further detailed documentation of the code is available in the repository.

# A Appendix

# List of Figures

# References

[1] Jeremy Fix. *Efficient convolution using the Fast Fourier Transform, Application in C++*. May 30, 2011.

[2] https://en.wikipedia.org/wiki/Circular_convolution. Circular convolution.

[3] https://en.wikipedia.org/wiki/Circular_shift. Circular shift.

[4] https://en.wikipedia.org/wiki/Kernel_(image_processing)#Convolution. Convolution.

[5] https://en.wikipedia.org/wiki/Python_(programming_language). Python (programming language).

[6] https://matplotlib.org/. Matplotlib.

[7] https://www.scipy.org/scipylib/index.html. Scipy.

[8] https://www.tensorflow.org/. Tensorflow.

[9] http://www.numpy.org/. Numpy.

[10] Axel Maas. *Lattice quantum field theory*. Lecture in SS 2017 at the KFU Graz, 2017.