Report

Andrija Radočaj

May 5, 2025

Many thanks to my mentor, dr. Fesel-Kamenik, for the opportunity, his guidance and assistance throughout the traineeship.

Thank you to Lara, Iva, Nikiša, Peter, Bučka, Adrian and Fran for all the lunches and beautiful moments.

Special thanks to my partner in crime from the office, Lara.

Pika konc.

Contents

1 Introduction

This report explores various data structures and analyzes data from multiple particle physics data sets. The second section introduces the relevant quantities. The third section is a guide through the preparation of the environment in which the codes can be run. The fourth section deals with running the codes on a cluster by submitting them as jobs. The fifth section is a tutorial on code parallelization and implementation of shared-memory usage between parallel processes. The sixth section shows how to read three different data sets. The seventh section helps with the visualization of jets. The eighth section goes in detail on how the normalization of the histogram introduces bin correlations. The ninth section is a tutorial on all the modeling methods and tools explored. The final tenth section presents the results. The end goal of this report was to find an analytic function that could describe the probability distribution or probability density function of the radius of QCD jets.
In this report, three files have been used.

- The file `WToQQ_070.root` was extracted from `JetClass_Pythia_train_100M_part7.tar` which was downloaded from https://zenodo.org/records/6619768.
  This file was used to study the data structure in .root files and to compare the data with the other two files.
- The file `RunG_batch0.h5` was downloaded from https://www.fdr.uni-hamburg.de/record/16505.
  This file contains a mixture of jets with diverse origins, so the assumption was made that jets with transverse momentum up to approximately 500 GeV are almost exclusively of QCD origin.
- The file `QCDjj_pT_450_1200_train01.h5` was extracted from `QCD.tar.gz` which was downloaded from https://zenodo.org/records/12793616.

Python codes were run on a remote computer cluster as jobs.
The Python distribution used is Anaconda3/2023.03-1.
SSH (Secure Socket Shell) was run in WSL (Windows Subsystem for Linux) for remote connection.

The main codes `jet_dist.py`, `functions.py` and `data_root.py` are the ones responsible for the end results. They are not presented in this report directly because of their size.

2 Starting point

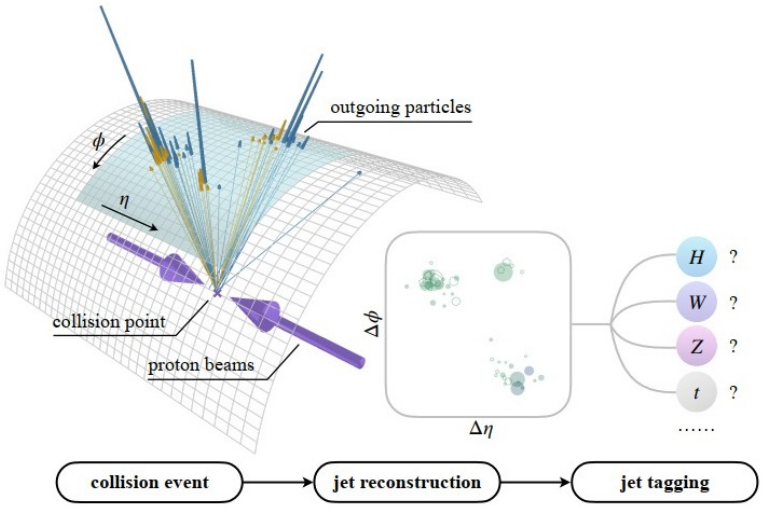

Figure 1: Illustration of jet tagging at the CERN's LHC [1]

The linear momentum of some jet's constituents is usually represented by $\vec{p} = (p_T, \eta, \phi)$ [2]. These quantities are best described by Figure 2 below and the equations (1)-(5).

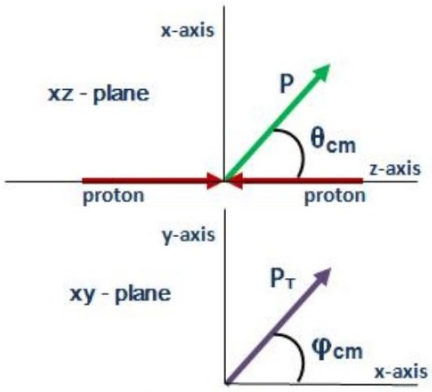

Figure 2: Coordinate system [2]

$$\eta = \frac{1}{2}\ln\left(\frac{p + p_z}{p - p_z}\right) = -\ln\left[\tan\left(\frac{\theta}{2}\right)\right] \tag{1}$$

$$p_x = p_T \cos\phi \tag{2}$$

$$p_y = p_T \sin\phi \tag{3}$$

$$p_z = p_T \sinh\eta \tag{4}$$

$$p = p_T \cosh\eta \tag{5}$$

Now, the quantity R is easily calculated with the help of equation (6) below:

$$R_{i(J)} = \sqrt{(\eta_J - \eta_{i(J)})^2 + (\phi_J - \phi_{i(J)})^2} \equiv \sqrt{(\Delta\eta_{i(J)})^2 + (\Delta\phi_{i(J)})^2}, \tag{6}$$

where $\eta_{i(J)}$ and $\phi_{i(J)}$ are from the i-th particle of the J-th jet, while η_J and ϕ_J are from the corresponding jet itself. The latter are calculated by first summing 3-momenta (p_x, p_y, p_z) of all particles in a jet and then calculating p_J and p_{J_z}.

2.1 Careful with handling $\Delta\phi$

Differences $\Delta\phi_{i(J)}$ should be treated with care because ϕ is an azimuthal angle. That is, it should be $\Delta\phi_{i(J)} \in [-\pi, \pi]$. While .root files already have $\Delta\phi_{i(J)}$ correctly calculated, in .h5 files $\Delta\phi_{i(J)}$ has to be carefully calculated with the help of ϕ_J and $\phi_{i(J)}$ as in the expressions (7) and (8) below:

$$\Delta\phi_{i(J)} \leftarrow \phi_J - \phi_{i(J)}; \tag{7}$$

$$\Delta\phi_{i(J)} \leftarrow \mathrm{MOD}\big(\Delta\phi_{i(J)} + \pi, 2\pi\big) - \pi. \tag{8}$$

3 Preparing the environment

The first two steps are to be skipped if the environment wants to be set up on a local computer. The commands from the third step then have to be run in Anaconda Powershell Prompt.

The first step is to open WSL and remotely connect to the cluster via SSH.

The second step is to run the following commands.

WSL terminal:

```
module load Anaconda3/2023.03-1 #to load the module
conda init bash #just the first time
source ~/.bashrc #to be able to activate the environment
```

Figure 3: Commands for WSL once connected via SSH to the cluster

The third step is to run the following commands.

WSL terminal / Anaconda Powershell Prompt:

```
conda update conda
conda config --add channels anaconda
conda config --add channels conda-forge
conda info --envs #list all the environments
conda create -n jet scipy numpy matplotlib #create a new environment
```

```
conda activate jet #activate the new environment
python --version #check the python version
conda install uproot #for reading .root files
conda install h5py #for reading .h5 files
conda install vector #for managing 4-vectors
conda install awkward #works closly with vector package
conda install pandas #for use of machine scientist
conda install ipywidgets #for use of machine scientist
conda install sympy #for use of machine scientist
conda install scikit-learn #for quantile transformation and KDE
conda list package_name #check version of the installed package
```

Figure 4: Commands for setting up the environment

Additionally, if one is setting up the environment on a local computer, then, aside from running the mandatory commands from the third step in Anaconda Powershell Prompt, one can optionally run the following commands there too to make the environment appear as kernel in Jupyter Notebook.

Anaconda Powershell Prompt:

```
conda install ipykernel
#make the environment appear as kernel in Jupyter Notebook:
python -m ipykernel install --user --name jet --display-name "jet_project"
conda deactivate jet
jupyter notebook
```

Figure 5: Optional commands for Anaconda Powershell Prompt

After the environment has been set up, each time a new connection through SSH in WSL is made, the following commands have to be run.

WSL terminal:

```
module load Anaconda3/2023.03-1 #to load the module
source ~/.bashrc #to be able to activate the environment
conda activate jet #activate the environment
```

Figure 6: Commands for each time a new connection is made through SSH in WSL

4 Submitting a job

The first step in running some code on a cluster is to create a Python script and open it.

WSL terminal:

```
nano code.py
```

Once a Python script is created and opened, one can write some simple code in it to test things out.

code.py:

```
print("Hello world!")
```

To exit the script, one should press Ctrl+X and then follow the instructions to save the changes or not. The second step is to create a shell script and open it.

WSL terminal:

```
nano job.sh
```

Once a shell script is created and opened, one should write the following instructions for SLURM to then read them. SLURM (Simple Linux Utility for Resource Management) is an open-source job scheduler used to manage and allocate resources on high-performance computing (HPC) clusters - it takes job requests, figures out when and where to run them, and makes sure resources are used efficiently.

job.sh:

```
# Use BASH (Bourne Again Shell) as a command-line interpreter:
#!/bin/bash

#SBATCH --job-name=steve_job

# In this file the output of a code will appear
#SBATCH --output=output.out #SLURM directive

# In this file potential errors and logging will appear
#SBATCH --error=error.err #SLURM directive

# To prevent accidental infinite loops and minimize disruption:
# - A job will be automatically killed when the time limit expires,
#   ensuring other users don't have to wait unnecessarily.
```

```
#SBATCH --time=00:00:10  # SLURM directive

# To position a job in the queue appropriately:
# - If the memory is too little for the code to run, the job will be
#   killed and may raise an error.
# - If the memory is too much, the job will wait longer in queue
#   than necessary, but once running, only the required memory
#   will be used.
#SBATCH --mem=100M  # SLURM directive

#short(<3h)
#day(3h-1day)
#long(1day-30days)
#SBATCH --partition=short #SLURM directive

# Number of CPUs to use:
#SBATCH --cpus-per-task=1 #SLURM directive

python code.py
```

Again, to exit the script, one should press Ctrl+X and then follow the instructions to save the changes or not. The last step is to submit the job.

WSL terminal:

```
sbatch job.sh
```

To see the status of a job the following command can be run.

WSL terminal:

```
squeue -u username
```

One can open the output and error file with the help of nano editor, same as with the scripts.

5 Code parallelization

To speed up code execution, parallelization of loops, where each iteration is independent of the others, can be done. In this section a couple of simple codes with parallelized loops will be presented. For running the codes on a computer cluster, a shell script has to be created.

WSL terminal:

```
nano job.sh
```

job.sh:

```
#!/bin/bash

#SBATCH --job-name=steve_job

#SBATCH --output=output.out

#SBATCH --error=error.err

#SBATCH --time=00:00:30

#SBATCH --mem=100M

#SBATCH --partition=short

# For CPUs per task put the same number as the number
# of processes in a Python script:
#SBATCH --cpus-per-task=4

python code.py
```

5.1 Function of one argument, one return value

The first code works with a function that takes only one argument and returns only one value.

WSL terminal:

```
nano code.py
```

code.py:

```
import logging
from multiprocessing import Pool

def square(x):
    logging.basicConfig(level=logging.INFO)
    logging.info(f"argument␣{x}␣in␣progress..")
    return x * x
```

```
8
9  def main():
10     # List of arguments to pass to the function
11     input_data = [1, 2, 3, 4, 5]
12
13     # Create a pool of worker processes
14     with Pool(processes=4) as pool:
15             # Use map to pass each element of data
16             # as an argument to the function
17             squared = pool.map(square, input_data)
18
19     # Print results
20     print("Input:", input_data)
21     print("Squared:", squared)
22
23 if __name__ == "__main__":
24     main()
```

`if __name__ == "__main__":` **acts as a guard to prevent certain parts of the code from being executed when child processes are created during multiprocessing.**

The next step is to submit the job.

WSL terminal:

```
sbatch job.sh
```

After submitting the job, it should be checked if it is pending, running, or done.

WSL terminal:

```
squeue -u username
```

If the job does not appear in the list with the last command, that means that it is done and one should then first check the error file.

WSL terminal:

```
nano error.err
```

error.err:

```
INFO:root:argument 2 in progress..
INFO:root:argument 1 in progress..
INFO:root:argument 5 in progress..
```

```
INFO:root:argument 4 in progress..
INFO:root:argument 3 in progress..
```

If the error file only contains log information, everything is fine, and one should proceed with opening the output file.

WSL terminal:

```
nano output.out
```

output.out:

```
Input: [1, 2, 3, 4, 5]
Squared: [1, 4, 9, 16, 25]
```

5.2 Function of one argument, two return values

The second code works with a function that takes only one argument and returns two values.

WSL terminal:

```
nano code.py
```

code.py:

```
import logging
from multiprocessing import Pool

def compute(x):
    logging.basicConfig(level=logging.INFO)
    logging.info(f"argument {x} in progress..")
    return x * x, x + x

def main():
    # List of arguments to pass to the function
    input_data = [1, 2, 3, 4, 5]

    # Create a pool of worker processes
    with Pool(processes=4) as pool:
        # Use map to pass each element of data as
        # an argument to the function
        results = pool.map(compute, input_data)

    # Create empty lists to store the results separately
```

```
    squares = []
    sums = []

    # Unpack the results manually using a for loop
    for square,summation in results:
            squares.append(square)
            sums.append(summation)

    # Print results
    print("Input:", input_data)
    print("Results:", results)
    print("Squares:", squares)
    print("Sums:", sums)

if __name__ == "__main__":
    main()
```

WSL terminal:

```
sbatch job.sh
```

WSL terminal:

```
nano error.err
```

error.err:

```
INFO:root:argument 2 in progress..
INFO:root:argument 1 in progress..
INFO:root:argument 3 in progress..
INFO:root:argument 4 in progress..
INFO:root:argument 5 in progress..
```

WSL terminal:

```
nano output.out
```

output.out:

```
Input: [1, 2, 3, 4, 5]
Results: [(1, 2), (4, 4), (9, 6), (16, 8), (25, 10)]
Squares: [1, 4, 9, 16, 25]
Sums: [2, 4, 6, 8, 10]
```

5.3 Function of two arguments, one return value

The third code works with a function that takes two arguments and returns only one value.

WSL terminal:

```
nano code.py
```

code.py:

```
import logging
from multiprocessing import Pool

def add(x, y):
    logging.basicConfig(level=logging.INFO)
    logging.info(f"argument␣({x},{y})␣in␣progress..")
    return x + y

def main():
    # List of tuples, where each tuple contains two arguments of the function
    input_data = [(1, 2), (3, 4), (5, 6), (7, 8), (9, 10)]

    # Create a pool of worker processes
    with Pool(processes=4) as pool:
            # Use starmap to pass each tuple from data as
            # arguments to the function
            sums = pool.starmap(add, input_data)

    # Print results
    print("Input:", input_data)
    print("Sums:", sums)

if __name__ == "__main__":
    main()
```

WSL terminal:

```
sbatch job.sh
```

WSL terminal:

```
nano error.err
```

error.err:

```
INFO:root:argument (1,2) in progress..
INFO:root:argument (3,4) in progress..
INFO:root:argument (9,10) in progress..
INFO:root:argument (5,6) in progress..
INFO:root:argument (7,8) in progress..
```

WSL terminal:

```
nano output.out
```

output.out:

```
Input: [(1, 2), (3, 4), (5, 6), (7, 8), (9, 10)]
Sums: [3, 7, 11, 15, 19]
```

5.4 Function of two arguments, two return values

The fourth code works with a function that takes two arguments and returns two values.

WSL terminal:

```
nano code.py
```

code.py:

```
import logging
from multiprocessing import Pool

def compute(x, y):
    logging.basicConfig(level=logging.INFO)
    logging.info(f"argument ({x},{y}) in progress..")
    return x * y, x + y

def main():
    # List of tuples, where each tuple contains two arguments of the function
    input_data = [(1, 2), (3, 4), (5, 6), (7, 8), (9, 10)]

    # Create a pool of worker processes
    with Pool(processes=4) as pool:
            # Use starmap to pass each tuple from data
            # as arguments to the function
            results = pool.starmap(compute, input_data)

```

```
    # Create empty lists to store the results separately
    products = []
    sums = []

    # Unpack the results manually using a for loop
    for product,summation in results:
            products.append(product)
            sums.append(summation)

    # Print results
    print("Input:", input_data)
    print("Results:", results)
    print("Products:", products)
    print("Sums:", sums)

if __name__ == "__main__":
    main()
```

WSL terminal:

```
sbatch job.sh
```

WSL terminal:

```
nano error.err
```

error.err:

```
INFO:root:argument (3,4) in progress..
INFO:root:argument (1,2) in progress..
INFO:root:argument (5,6) in progress..
INFO:root:argument (7,8) in progress..
INFO:root:argument (9,10) in progress..
```

WSL terminal:

```
nano output.out
```

output.out:

```
Input: [(1, 2), (3, 4), (5, 6), (7, 8), (9, 10)]
Results:, [(2, 3), (12, 7), (30, 11), (56, 15), (90, 19)]
Products: [2, 12, 30, 56, 90]
Sums: [3, 7, 11, 15, 19]
```

5.5 Implementation of shared memory usage

The fifth code shows inefficient usage of memory in code parallelization and why shared memory between processes is then useful. The goal of a code is to fill the blocks on the diagonal of some matrices. By first creating and filling small matrices and then copying them onto the diagonal blocks of a bigger matrix, extra memory is unnecessarily wasted.

WSL terminal:

```
nano code.py
```

code.py:

```
import logging
import numpy as np
from multiprocessing import Pool

# Function to compute the matrices
def compute_matrix(x, y, n):

    logging.basicConfig(level=logging.INFO)
    logging.info(f"argument ({x},{y}) in progress..")

    # Create a (n x n) matrix filled with the same value x * y
    product_matrix = np.full((n, n), x * y)

    # Create a (n x n) matrix filled with the same value x + y
    sum_matrix = np.full((n, n), x + y)

    return product_matrix, sum_matrix

def main():
    # List of tuples,
    # where each tuple contains the first two arguments of the function
    input_data = [(1, 2), (3, 4), (5, 6), (7, 8), (9, 10)]
    input_data_len=len(input_data)

    # Size of each matrix (2x2 matrix)
    n = 2

    # Prepare the list of full arguments (x, y, n) for each function call
    args=[(*input_data[i],n) for i in range(input_data_len)]

    # Create a pool of worker processes
    with Pool(processes=4) as pool:
            # Use starmap to distribute work across the pool of processes
            results=pool.starmap(compute_matrix,args)

```

```
    # Create a big matrix where each 'product_matrix'
    # will be placed as a separate block on the diagonal
    # (note: small matrices are already computed and stored in 'results')
    # - allocating extra memory here for combining into one large matrix
    final_product_matrix=np.zeros([input_data_len*n,input_data_len*n])

    # Create a big matrix where each 'sum_matrix'
    # will be placed as a separate block on the diagonal
    # (note: small matrices are already computed and stored in 'results')
    # - allocating extra memory here for combining into one large matrix
    final_sum_matrix=np.zeros([input_data_len*n,input_data_len*n])

    # Placing product_matrices and sum_matrices
    # as blocks on the diagonal of a corresponding matrix
    idx=0
    for product_matrix,sum_matrix in results:
        index1=idx*n
        index2=(idx+1)*n

        # Copy product_matrix into the diagonal block of final_product_matrix
        final_product_matrix[index1:index2,index1:index2]=product_matrix

        # Copy sum_matrix into the diagonal block of final_sum_matrix
        final_sum_matrix[index1:index2,index1:index2]=sum_matrix
        idx+=1

    # Print the final matrices
    print("Input:", input_data)
    print("Product␣block-diagonal␣matrix:")
    print(final_product_matrix)
    print("Sum␣block-diagonal␣matrix:")
    print(final_sum_matrix)

if __name__ == "__main__":
    main()
```

WSL terminal:

```
sbatch job.sh
```

WSL terminal:

```
nano error.err
```

error.err:

```
INFO:root:argument (1,2) in progress..
INFO:root:argument (3,4) in progress..
INFO:root:argument (9,10) in progress..
INFO:root:argument (5,6) in progress..
INFO:root:argument (7,8) in progress..
```

WSL terminal:

```
nano output.out
```

output.out:

```
Input: [(1, 2), (3, 4), (5, 6), (7, 8), (9, 10)]
Product block-diagonal matrix:
[[ 2.  2.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 2.  2.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0. 12. 12.  0.  0.  0.  0.  0.  0.]
 [ 0.  0. 12. 12.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0. 30. 30.  0.  0.  0.  0.]
 [ 0.  0.  0.  0. 30. 30.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0. 56. 56.  0.  0.]
 [ 0.  0.  0.  0.  0.  0. 56. 56.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0. 90. 90.]
 [ 0.  0.  0.  0.  0.  0.  0.  0. 90. 90.]]
Sum block-diagonal matrix:
[[ 3.  3.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 3.  3.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  7.  7.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  7.  7.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0. 11. 11.  0.  0.  0.  0.]
 [ 0.  0.  0.  0. 11. 11.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0. 15. 15.  0.  0.]
 [ 0.  0.  0.  0.  0.  0. 15. 15.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0. 19. 19.]
 [ 0.  0.  0.  0.  0.  0.  0.  0. 19. 19.]]
```

The sixth code shows efficient usage of memory through memory sharing between processes. This time, only one big matrix is created, and its diagonal blocks are filled in directly without an intermediate step of first creating small matrices.

WSL terminal:

```
nano code.py
```

code.py:

```
import logging
import numpy as np
from multiprocessing import Pool
from multiprocessing.shared_memory import SharedMemory

# Function to compute the matrices
def compute_matrix(x, y, idx, n, input_data_len ,shm_product_matrix_name,
    shm_sum_matrix_name):

    logging.basicConfig(level=logging.INFO)
    logging.info(f"argument ({x},{y}) in progress..")

    # Attach to the existing shared memory block
    existing_shm_product_matrix = SharedMemory(name=shm_product_matrix_name)

    # Create a NumPy array that wraps the shared memory buffer
    product_matrix = np.ndarray((n*input_data_len,n*input_data_len),
                        dtype=np.float64,buffer=existing_shm_product_matrix.buf)

    # Attach to the existing shared memory block
    existing_shm_sum_matrix = SharedMemory(name=shm_sum_matrix_name)

    # Create a NumPy array that wraps the shared memory buffer
    sum_matrix = np.ndarray((n*input_data_len,n*input_data_len),
                    dtype=np.float64,buffer=existing_shm_sum_matrix.buf)

    index1=idx*n
    index2=(idx+1)*n
    # Fill the block on the diagonal with the same value x * y
    product_matrix[index1:index2,index1:index2] = np.full((n, n), x * y)

    # Fill the block on the diagonal with the same value x + y
    sum_matrix[index1:index2,index1:index2] = np.full((n, n), x + y)

def main():
    # List of tuples,
    # where each tuple contains the first two arguments of the function
    input_data = [(1, 2), (3, 4), (5, 6), (7, 8), (9, 10)]
    input_data_len=len(input_data)

    # Size of each matrix (2x2 matrix)
    n = 2

    # Create shared memory for the big matrix
    # Each float64 takes 8 bytes
    shm_product_matrix = SharedMemory(
    create=True,size=n*input_data_len*n*input_data_len*8)
```

```
47
48      # View into the shared memory
49      product_matrix = np.ndarray((n*input_data_len,n*input_data_len),
50                                  dtype=np.float64,buffer=shm_product_matrix.buf)
51
52      # Create shared memory for the big matrix
53      # Each float64 takes 8 bytes
54      shm_sum_matrix = SharedMemory(
55      create=True,size=n*input_data_len*n*input_data_len*8)
56
57      # View into the shared memory
58      sum_matrix = np.ndarray((n*input_data_len, n*input_data_len),
59                              dtype=np.float64,buffer=shm_sum_matrix.buf)
60
61      # Prepare the list of full arguments
62      # (x,y,idx,n,input_data_len,shm_product_matrix.name,shm_sum_matrix.name)
63      # for each function call:
64      args=[(*input_data[idx],idx,n,input_data_len,
65              shm_product_matrix.name,shm_sum_matrix.name)
66              for idx in range(input_data_len)]
67
68      # Create a Pool of processes
69      with Pool(processes=4) as pool:
70              # Use starmap to distribute work across the pool of processes
71              pool.starmap(compute_matrix,args)
72
73      # Print the final matrices
74      print("Input:", input_data)
75      print("Product block-diagonal matrix:")
76      print(product_matrix)
77      print("Sum block-diagonal matrix:")
78      print(sum_matrix)
79
80      # Clean up the shared memory once everything is done
81      shm_product_matrix.close()  # Detach from the shared memory
82      shm_product_matrix.unlink()  # Release the shared memory block
83
84      # Clean up the shared memory once everything is done
85      shm_sum_matrix.close()  # Detach from the shared memory
86      shm_sum_matrix.unlink()  # Release the shared memory block
87
88  if __name__ == "__main__":
89      main()
```

WSL terminal:

```
sbatch job.sh
```

WSL terminal:

```
nano error.err
```

error.err:

```
INFO:root:argument (1,2) in progress..
INFO:root:argument (3,4) in progress..
INFO:root:argument (9,10) in progress..
INFO:root:argument (5,6) in progress..
INFO:root:argument (7,8) in progress..
```

WSL terminal:

```
nano output.out
```

output.out:

```
Input: [(1, 2), (3, 4), (5, 6), (7, 8), (9, 10)]
Product block-diagonal matrix:
[[ 2.  2.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 2.  2.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0. 12. 12.  0.  0.  0.  0.  0.  0.]
 [ 0.  0. 12. 12.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0. 30. 30.  0.  0.  0.  0.]
 [ 0.  0.  0.  0. 30. 30.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0. 56. 56.  0.  0.]
 [ 0.  0.  0.  0.  0.  0. 56. 56.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0. 90. 90.]
 [ 0.  0.  0.  0.  0.  0.  0.  0. 90. 90.]]
Sum block-diagonal matrix:
[[ 3.  3.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 3.  3.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  7.  7.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  7.  7.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0. 11. 11.  0.  0.  0.  0.]
 [ 0.  0.  0.  0. 11. 11.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0. 15. 15.  0.  0.]
 [ 0.  0.  0.  0.  0.  0. 15. 15.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0. 19. 19.]
 [ 0.  0.  0.  0.  0.  0.  0.  0. 19. 19.]]
```

6 Reading from the files

6.1 JetClass data set

Each data set mentioned consists of clustered jets. Each jet has its own data components, as well as each particle within each jet. Figure 7 and Figure 8 help in understanding the data structure of .root files. Each jet has its own dictionary for which keys represent different entries or, in other words, hold different information about the jet.

code.ipynb:

```
import uproot
filepath = "WToQQ_070.root" #change if needed
tree = uproot.open(filepath)['tree']
tree.arrays()
```

Figure 7: Jupyter Notebook cell: inspection of data structure in .root file

output:

```
[{part_px: [62.5, 68.2, ..., 0.946, 1.21], part_py: [215, ...], ...},
 {part_px: [-109, -82, ..., -1.39, -0.967], part_py: [...], part_pz: ..., ...},
 ...,
 {part_px: [-159, -81.3, ..., -0.957], part_py: [-74.2, ...], ...}]
--------------------------------------------------------------------------------
type: 100000 * {
    part_px: var * float32,
    part_py: var * float32,
    part_pz: var * float32,
    part_energy: var * float32,
    part_deta: var * float32,
    part_dphi: var * float32,
    part_d0val: var * float32,
    part_d0err: var * float32,
    part_dzval: var * float32,
    part_dzerr: var * float32,
    part_charge: var * float32,
    part_isChargedHadron: var * int32,
    part_isNeutralHadron: var * int32,
    part_isPhoton: var * int32,
    part_isElectron: var * int32,
    part_isMuon: var * int32,
    label_QCD: float32,
    label_Hbb: bool,
    label_Hcc: bool,
    label_Hgg: bool,
    label_H4q: bool,
```

```
    label_Hqql: bool,
    label_Zqq: int32,
    label_Wqq: int32,
    label_Tbqq: int32,
    label_Tbl: int32,
    jet_pt: float32,
    jet_eta: float32,
    jet_phi: float32,
    jet_energy: float32,
    jet_nparticles: float32,
    jet_sdmass: float32,
    jet_tau1: float32,
    jet_tau2: float32,
    jet_tau3: float32,
    jet_tau4: float32,
    aux_genpart_eta: float32,
    aux_genpart_phi: float32,
    aux_genpart_pid: float32,
    aux_genpart_pt: float32,
    aux_truth_match: float32 }
```

Figure 8: Data structure in .root file

In terms of reading the JetClass data set's .root files, the procedure in Figure 9 can be used. This procedure utilizes Python lists as data structures with flexible shapes, which then point to the fact that not each jet has the same number of particles.

code.py:

```python
import uproot, vector
filepath = "WToQQ_070.root" #change if needed
tree = uproot.open(filepath)['tree']
jets = {}
components = ['px', 'py', 'pz', 'energy', 'dphi', 'deta', 'isChargedHadron', '
    isNeutralHadron', 'isPhoton', 'isElectron', 'isMuon']
for component in components:
    label = 'part_' + component
    branch = tree[label]
    jets[label] = [jet for jet in branch.array().tolist()]
p4 = vector.zip({'px': jets['part_px'],
                 'py': jets['part_py'],
                 'pz': jets['part_pz'],
                 'energy': jets['part_energy']})
jets['part_pt']=p4.pt.tolist()
jets['part_eta']=p4.eta.tolist()
jets['part_phi']=p4.phi.tolist()
components = ['pt', 'eta', 'phi', 'nparticles', 'energy']
```

```
for component in components:
    label = 'jet_' + component
    branch = tree[label]
    jets[label] = [jet for jet in branch.array().tolist()]
```

Figure 9: Reading from .root file [3]

Another way of reading the data from JetClass data set's .root files is by using `read_file` function of `dataloader.py` which can be found here. This procedure involves zero-padding in order for each jet to have the same number of particles as to then enable using numpy arrays of fixed shapes for better code efficiency. The maximum number of particles can be set and each jet that has fewer particles than the set maximum is zero-padded while each jet that has more particles than the set maximum is truncated. A byproduct of zero-padding are dummy constituents which have all their features equal to zero. Each particle with its transverse momentum being zero will be considered a dummy constituent because, while technically a real particle can have zero transverse momentum, it is extremely rare. Moreover, in this report, as will be seen, particles with zero transverse momentum do not contribute to any calculations. **Important note: for the purposes of this report, the calculation and return of the one-hot encoded numpy array holding the information about the origin of a jet in `read_file` function are commented out.**

code.py:

```
import sys
sys.path.append('./')

import numpy as np
from dataloader import read_file

input="WToQQ_070.root" #change if needed

particle_features=['part_pt','part_eta','part_phi',
                   'part_deta','part_dphi','part_energy',
                   'part_isChargedHadron','part_isNeutralHadron',
                   'part_isElectron','part_isMuon','part_isPhoton']

jet_features=['jet_pt','jet_eta','jet_phi']

x_particles, x_jets = read_file(
                   filepath=input,particle_features=particle_features,
                       jet_features=jet_features)

mask = x_particles[:, 0, :] == 0

full_mask = np.repeat(mask[:, np.newaxis, :], x_particles.shape[1], axis=1)

```

```
x_particles=np.ma.masked_where(full_mask,x_particles)

print(f"x_particles.shape={x_particles.shape}")
#(num_jets,num_particle_features,num_particles)
print(f"\n",flush=True)

print(f"x_jets.shape={x_jets.shape}")
#(num_jets,num_jet_features)
print(f"\n",flush=True)

jets={}

for pf,label in enumerate(particle_features):
    jets[label]=x_particles[:,pf,:]

for jf,label in enumerate(jet_features):
    jets[label]=x_jets[:,jf]

jets['jet_nparticles']=jets['part_pt'].count(axis=1)
```

Figure 10: Reading from any JetClass data set's .root file [1]

As commented in `dataloader.py` the numpy array *x_particles*, which holds information about particle level features, is of shape *(num_jets,num_particles_features,max_num_particles)* while the numpy array *x_jets*, which holds the information about jet level features, is of shape *(num_jets,num_jet_features)*. The purpose of masking is to mark each dummy constituent as irrelevant. In order to see how masking in the above figure works, one can run the following code in Figure 11 the output of which can be seen in Figure 12.

code.ipynb:

```
import numpy as np
#two matrices in c represent two jets
#each column represents each particle
#each row represents different feature
c=np.array([[[1,0,5,7],[0,6,0,1],[0,2,3,2]],[[0,4,9,0],[0,7,5,2],[6,3,8,4]]])
print("c:")
print(c)
print("------------------------------------------------------------------------")
mask=c[:,0,:]==0
print("mask:")
print(mask)
print("------------------------------------------------------------------------")
full_mask=np.repeat(mask[:,np.newaxis,:],c.shape[1],axis =1)
print("full_mask:")
print(full_mask)
print("------------------------------------------------------------------------")
```

```
c_masked=np.ma.masked_where(full_mask,c)
print("c_masked:")
print(c_masked)
```

Figure 11: Helper cell for illustration of masking

output:

```
c:
[[[1 0 5 7]
  [0 6 0 1]
  [0 2 3 2]]

 [[0 4 9 0]
  [0 7 5 2]
  [6 3 8 4]]]
--------------------------------------------------------------------------
mask:
[[False  True False False]
 [ True False False  True]]
--------------------------------------------------------------------------
full_mask:
[[[False  True False False]
  [False  True False False]
  [False  True False False]]

 [[ True False False  True]
  [ True False False  True]
  [ True False False  True]]]
--------------------------------------------------------------------------
c_masked:
[[[1 -- 5 7]
  [0 -- 0 1]
  [0 -- 3 2]]

 [[-- 4 9 --]
  [-- 7 5 --]
  [-- 3 8 --]]]
```

Figure 12: Output of the helper cell

6.2 Rodem data set

Reading from Rodem data set's .h5 files is described in Figure 13 below and the data is also zero-padded.

code.py:

```
import h5py
import numpy as np

def load_cnsts_rodem(ifile: str, mask_pt: list):
    """Load constituents from an HDF5 file."""
    with h5py.File(ifile, "r") as f:

        cnsts=f["objects/jets/jet1_cnsts"][mask_pt,:,:3] #first three features

    mask = np.repeat(cnsts[:, :, 0] == 0, cnsts.shape[2])
    mask = mask.reshape(-1, cnsts.shape[1], cnsts.shape[2])
    cnsts = np.ma.masked_where(mask, cnsts)

    print(f"cnsts.shape={cnsts.shape}")
    #(num_jets,num_particles,num_particle_features)
    print(f"\n",flush=True)

    return cnsts[:,:,0], cnsts[:,:,1], cnsts[:,:,2] #pt,eta,phi

def load_jets_rodem(ifile: str, mask_pt: list):
    """Load jets from an HDF5 file."""
    with h5py.File(ifile, "r") as f:

        jets = f["objects/jets/jet1_obs"][mask_pt,:3] #first three features

    print(f"jets.shape={jets.shape}")
    #(num_jets,num_jet_features)
    print(f"\n",flush=True)

    return jets[:,0], jets[:,1], jets[:,2] #pt,eta,phi

#the input HDF5 file containing the relevant jets:
input = "QCDjj_pT_450_1200_train01.h5" #change if needed

#the number of jets to load:
n_jets = 400_000 #change if needed

#jet pt range:
min_pt=450 #change if needed
max_pt=1000 #change if needed

with h5py.File(input, "r") as f:
    mask_pt=[index for index,value in enumerate(f["objects/jets/jet1_obs"][:
        n_jets,0]) if ((value>=min_pt) and (value<=max_pt))]

jets={}

```

```
#rows -> jets ; columns -> particles
jets['part_pt'],jets['part_eta'],jets['part_phi']=load_cnsts_rodem(input,mask_pt
    )

#flat arrays
jets['jet_pt'],jets['jet_eta'],jets['jet_phi']=load_jets_rodem(input,mask_pt)

jets['jet_nparticles']=jets['part_pt'].count(axis=1)

print(f"Smallest jet_pt is: {np.min(jets['jet_pt'])}")
print(f"Biggest jet_pt is: {np.max(jets['jet_pt'])}",flush=True)
```

Figure 13: Reading from any Rodem data set's .h5 file [4]

To understand and visualize the masking done in the figure above it is best to run the cell given in Figure 14 of which the output can be seen in Figure 15.

code.ipynb:

```
import numpy as np
#two matrices in c represent two jets
#each row represents each particle
#each column represents different feature
c=np.array([[[1,2,5],[0,6,0],[0,2,3]],[[6,4,9],[0,7,5],[6,3,8]]])
print("c:")
print(c)
print("-------------------------------------------------------------------------")
mask=np.repeat(c[:,:,0]==0, c.shape[2])
print("mask:")
print(mask)
print("-------------------------------------------------------------------------")
mask_shaped=mask.reshape(-1, c.shape[1], c.shape[2])
print("mask_shaped:")
print(mask_shaped)
print("-------------------------------------------------------------------------")
c_masked=np.ma.masked_where(mask_shaped,c)
print("c_masked:")
print(c_masked)
print("-------------------------------------------------------------------------")
count1=c_masked.count(axis=1)
print("count1:")
print(count1)
print("-------------------------------------------------------------------------")
count2=c_masked.count(axis=1)[:,0]
print("count2:")
print(count2)
```

Figure 14: Helper cell for illustration

output:

```
c:
[[[1 2 5]
  [0 6 0]
  [0 2 3]]

 [[6 4 9]
  [0 7 5]
  [6 3 8]]]
------------------------------------------------------------------------
mask:
[False False False  True  True  True  True  True  True False False False
  True  True  True False False False]
------------------------------------------------------------------------
mask_shaped:
[[[False False False]
  [ True  True  True]
  [ True  True  True]]

 [[False False False]
  [ True  True  True]
  [False False False]]]
------------------------------------------------------------------------
c_masked:
[[[1 2 5]
  [-- -- --]
  [-- -- --]]

 [[6 4 9]
  [-- -- --]
  [6 3 8]]]
------------------------------------------------------------------------
count1:
[[1 1 1]
 [2 2 2]]
------------------------------------------------------------------------
count2:
[1 2]
```

Figure 15: Output of the helper cell

6.3 Aspen Open Jets data set

Reading from Aspen Open Jets data set's .h5 files is described in Figure 16 below and, like the previous data set, contains zero-padded data.

code.py:

```
import h5py
import vector
import awkward as ak
import numpy as np

def load_cnsts_aspen(ifile: str, mask_pt: list):
    """Load constituents from an HDF5 file."""
    with h5py.File(ifile, "r") as f:

        PFCands = f["PFCands"][mask_pt,:,:4] #px,py,pz,E

    p4_cnsts=vector.zip({'px':PFCands[:,:,0],
                         'py':PFCands[:,:,1],
                         'pz':PFCands[:,:,2],
                         'energy':PFCands[:,:,3]})

    cnsts=np.zeros([PFCands.shape[0],PFCands.shape[1],3])
    cnsts[:,:,0]=ak.to_numpy(p4_cnsts.pt,allow_missing=True)
    cnsts[:,:,1]=ak.to_numpy(p4_cnsts.eta,allow_missing=True)
    cnsts[:,:,2]=ak.to_numpy(p4_cnsts.phi,allow_missing=True)

    #masking now done the same way as for Rodem data set
    mask = np.repeat(cnsts[:, :, 0] == 0, cnsts.shape[2])
    mask = mask.reshape(-1, cnsts.shape[1], cnsts.shape[2])
    cnsts = np.ma.masked_where(mask, cnsts)

    print(f"cnsts.shape={cnsts.shape}")
    #(num_jets,num_particles,num_particle_features)
    print(f"\n",flush=True)

    return cnsts[:,:,0], cnsts[:,:,1], cnsts[:,:,2] #pt,eta,phi

def load_jets_aspen(ifile: str, mask_pt: list):
    """Load jets from an HDF5 file."""
    with h5py.File(ifile, "r") as f:

        jets = f["jet_kinematics"][mask_pt,:3]

    print(f"jets.shape={jets.shape}")
    #(num_jets,num_jet_features)
    print(f"\n",flush=True)

    return jets[:,0], jets[:,1], jets[:,2] #pt,eta,phi

#the input HDF5 file:
input = "RunG_batch0.h5"

```

```
#the number of jets to load:
n_jets = 400_000

#jet pt range:
min_pt=300 #change if needed
max_pt=500 #change if needed

jets={}

with h5py.File(input, "r") as f:

    mask_pt=[index for index,value in enumerate(f["jet_kinematics"][:n_jets,0])
        if ((value>=min_pt) and (value<=max_pt))]

    #rows -> jets ; columns -> particles
    jets['part_pt'],jets['part_eta'],jets['part_phi']=load_cnsts_aspen(input,
        mask_pt)

    #flat arrays
    jets['jet_pt'],jets['jet_eta'],jets['jet_phi']=load_jets_aspen(input,mask_pt
        )
```

Figure 16: Reading from any Aspen Open Jets data set's .h5 file [5]

7 JetClass data set: visualization of jets

After reading a file, with the help of the code in Figure 17, jets can be visualized as in Figure 18 below. Every constituent of each jet in JetClass data set's .root files is known to be a specific kind of particle through one-hot encoding.

Important: The code snippet below is a continuation of the code used to read the file.

code.py:

```
import os
import matplotlib.pyplot as plt

if not os.path.exists("./Plots"):
        os.makedirs("./Plots")

arg=np.argmax(jets['jet_pt'])
print(f"index␣of␣the␣jet␣with␣the␣highest␣pT:␣{arg}")
print(f"\n",flush=True)

fig,ax=plt.subplots(figsize=(20,12))

for i in range(jets['jet_nparticles'][arg]):
```

```
    if jets['part_isChargedHadron'][arg,i]==1:
        m='o'
        fcs='lightgreen'
    elif jets['part_isNeutralHadron'][arg,i]==1:
        m='o'
        fcs='none'
    elif jets['part_isElectron'][arg,i]==1:
        m='^'
        fcs='lightgreen'
    elif jets['part_isMuon'][arg,i]==1:
        m='v'
        fcs='lightgreen'
    elif jets['part_isPhoton'][arg,i]==1:
        m='p'
        fcs='none'
    ax.scatter(jets['part_deta'][arg,i],jets['part_dphi'][arg,i],color='
        lightgreen',marker=m,facecolors=fcs,s=jets['part_energy'][arg,i])

ax.set_xlabel(r"$\Delta\eta$")
ax.set_ylabel(r"$\Delta\phi$")
fig.savefig("./Plots/jet.jpg")
```

Figure 17: Plotting for visualization of a jet

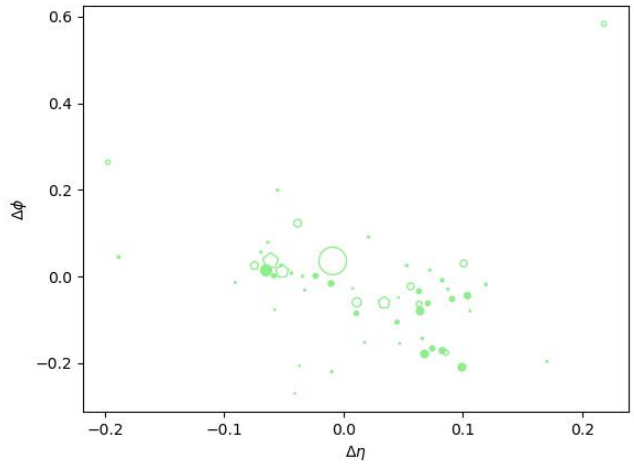

Figure 18: Jet with the highest p_T from `WToQQ_070.root` file. The circles, triangles (upward- or downward-directed), and pentagons represent the particle types, which are hadrons, leptons (electrons or muons), and photons, respectively. The solid (hollow) markers stand for electrically charged (neutral) particles [1].

8 Probability distribution of the radius

8.1 Contributions to the probability - 1D problem

The contribution of a particle that falls within some radius interval $[R_i - \Delta R, R_i + \Delta R]$ to the probability of the given interval is proportional to its transverse momentum because particles with higher transverse momentum contribute more significantly to the total momentum of the jet. These higher-energy particles are more likely to be involved in the jet structure and are therefore more likely to contribute to the probability in the specified radius interval. To obtain the probability plot of a certain jet energy or jet p_T range, numerous jets have to be considered to make a histogram and obtain the appropriate statistics. The jet's contribution to the probability of the given radius interval is then proportional to the sum of the transverse momenta of its particles that fall within the interval:

$$w^p_{ik(j)} = p_{T_{ik(J)}} \tag{9}$$

$$H^p_{i(J)} = \sum_{k(counts)} w^p_{ik(J)}, \tag{10}$$

where $p_{T_{ik(J)}}$ is the transverse momentum value of a particle that belongs to jet J and whose value R falls in the interval $[R_i - \Delta R,R_i + \Delta R]$. **For jet J, there are n_1 particles in total, and n_2 of these particles have the radius that falls within the interval $[R_i - \Delta R,R_i + \Delta R]$. The index k then runs from 1 to n_2, where each value of k corresponds to a particular particle within the jet and the interval.** In other words, $H^p_{i(J)}$ represents the unscaled contribution of a jet J to the probability of the radius interval $[R_i - \Delta R,R_i + \Delta R]$. The contributions to the probability of each jet to some radius interval have to be summed up in a certain way to ensure that the sum of all probability contributions from the whole radius range is equal to one.

$$h^p_i = \frac{1}{N(jets)} \sum_J \frac{H^p_{i(J)}}{\sum_{m(bins)} H^p_{m(J)}}. \tag{11}$$

With the applied normalization, all of the particle contributions to probability will sum up to one no matter how many jets are taken into account, while the contributions from dissimilar jets are kept uncorrelated at the same time.

8.2 Contributions to the probability density - 2D problem

Taking into account what was said in the previous subsection, the probability density at a certain value of R also must account for the fact that R is a radius in $(\Delta\eta,\Delta\phi)$ plane. The particle's contribution to the probability density of the interval $[R_i - dR,R_i + dR]$ has to stay proportional to its transverse momentum, but it has to be divided by the value of R_i because the infinitesimal surface area in a two-dimensional plane is calculated as $R{\cdot}dR{\cdot}d\alpha$, where α is just the symbol taken for the azimuthal angle in $(\Delta\eta,\Delta\phi)$ plane. The probability density of interest, called $\rho(R)$, is the one already integrated over the azimuthal angle.

$$\int_R \int_\alpha \rho(R,\alpha) \cdot R \cdot dR \cdot d\alpha = \int_R \rho(R) \cdot R \cdot dR \tag{12}$$

As with probability, to obtain the probability density plot of a certain jet energy or jet p_T range, numerous jets have to be considered in order to make a histogram and get appropriate statistics. As the contribution of a jet to the probability of a given radius, the contribution of a jet to the probability density in the interval $[R_i - \Delta R, R_i + \Delta R]$ must be proportional to the sum of transverse momenta of all its particles whose radius falls within the interval and, as described above, must be divided by R_i:

$$w^d_{ik(J)} = \frac{p_{T\,ik(J)}}{R_i} \tag{13}$$

$$H^d_{i(J)} = \sum_{k(counts)} w^d_{ik(J)}. \tag{14}$$

In other words, $H^d_{i(J)}$ represents the unscaled contribution of a jet J to the probability density of the interval $[R_i - \Delta R, R_i + \Delta R]$.

In order to ensure that the expression (12) is equal to one, the histogram must be normalized in a certain way:

$$\boxed{h^d_i = \frac{1}{N(jets)} \sum_J \frac{H^d_{i(J)}}{\Delta R \cdot \sum_{m(bins)} R_m H^d_{m(J)}},} \tag{15}$$

so that in the end the result is as follows:

$$\int_R \rho(R) \cdot R \cdot dR \to \sum_{i(bins)} h^d_i \cdot R_i \cdot \Delta R = \tag{16}$$

$$= \sum_J \frac{1}{N(jets)} \sum_{i(bins)} \frac{H^d_{i(J)}}{\Delta R \cdot \sum_{m(bins)} R_m H^d_{m(J)}} \cdot R_i \cdot \Delta R = \sum_J \frac{1}{N(jets)} = 1. \tag{17}$$

Observe again that the contributions from dissimilar jets are kept uncorrelated.

Moreover, it is now clear that the expression (15) is directly connected to the expression (11):

$$\boxed{\begin{aligned} h^d_i \cdot R_i \cdot \Delta R = \quad & (18)\\ = \frac{1}{N(jets)} \sum_J \frac{H^d_{i(J)}}{\Delta R \cdot \sum_{m(bins)} R_m H^d_{m(J)}} \cdot R_i \cdot \Delta R \quad & (19)\\ = \frac{1}{N(jets)} \sum_J \frac{H^p_{i(J)}}{\sum_{m(bins)} H^p_{m(J)}} \quad & (20)\\ = h^p_i. \quad & (21) \end{aligned}}$$

8.3 Histogram and Poisson distribution relation - in general

If the number of events is huge ($n >>$) and the probability for some event to happen is very small ($p <<$) then one is talking about a Poisson distribution. When falling into a bin of non-weighted histogram with a lot of counts ($n >>$) is considered an event with small probability ($p <<$) then the variance of this bin's value follows the expression for Poisson distribution and is equal to $Var = n \cdot p =$

$n(i)$, where $n(i)$ is the number of samples that fall into that specific bin. On the other hand, in the case where there is a histogram with weights, then one has a sum of weighted Poisson distributions. Here, $n >>$ is the number of samples with the specific weight, while p$<<$ is the probability that these samples will fall in the specific bin. In order to make the connection between weighted histograms and a linear combination of random variables, it is necessary to introduce $X_k^{(i)}$ as a random variable which for samples with weight value w_k gives 1 when they fall into the i-th bin and 0 otherwise. For the purpose of this section, all the weights $w_k^{(i)}$ whose counts fall into the i-th bin bin are arranged so that only the first m(i) weights have dissimilar values:

$$\{w_1^{(i)}, ..., w_{m(i)}^{(i)}, ..., w_{n(i)}^{(i)};\ \ m(i) \leq n(i)\}. \tag{22}$$

It is now possible to define a new random variable for each bin that represents a final histogram value:

$$X^{(i)} = \sum_{k=1}^{m(i)} w_k^{(i)} X_k^{(i)}, \tag{23}$$

For a linear combination of random uncorrelated variables $\left(X_1^{(i)},...,X_{m(i)}^{(i)}\right)$ with correspondingly different weights $\left(w_1^{(i)},..,w_{m(i)}^{(i)}\right)$ the variance is calculated in this manner:

$$Var(X^{(i)}) = \sum_{k=1}^{m(i)} {w_k^{(i)}}^2 Var(X_k^{(i)}), \tag{24}$$

where each $X_k^{(i)}$ is a Poisson distribution with variance equal to the number of samples in the i-th bin which have the same weight $w_k^{(i)}$ so essentially one is summing all of the squared weights whose counts fall into the i-th bin. Taking the latter into account, for the specific i-th bin with $n(i)$ counts, the variance is then equal to:

$$Var(X^{(i)}) = \sum_{k=1}^{n(i)} {w_k^{(i)}}^2. \tag{25}$$

Normalization of a histogram, or analogously the weights, changes the situation and introduces bin correlations. The covariance matrix must be calculated to fit the data with the complete information. **Standard deviation of the probability for the specific bin in a weighted histogram is then equal to the square root of the sum of squared weights of all samples that fall into that bin.**

8.4 Covariance matrix - in general

The propagation formula for the covariance matrix will be of much use in this report, so here is the short proof.

$$Cov(X, Y) = E[(X - E[X])(Y - E[Y])] \tag{26}$$

For $f(X_1, ..., X_n)$ the first-order Taylor expansion around $(E[X_1], ..., E[X_n])$ follows:

$$f(X_1, ..., X_n) \approx f(E[X_1], ..., E[X_n]) + \sum_i \frac{\partial f}{\partial X_i}\bigg|_{E[X_i]} (X_i - E[X_i]) \tag{27}$$

$$E[f] \approx f(E[X_1], ..., E[X_n]) + \sum_i \frac{\partial f}{\partial X_i}\bigg|_{E[X_i]} \cancelto{0}{(E[X_i] - E[X_i])} = f(E[X_1], ..., E[X_n]) \tag{28}$$

$$f(X_1, ..., X_n) \approx E[f] + \sum_i \frac{\partial f}{\partial X_i}\bigg|_{E[X_i]} (X_i - E[X_i]) \tag{29}$$

Now, for $f(X_1, ..., X_n)$ and $g(X_1, ..., X_n)$ in the Taylor expansion of first order, it stands:

$$Cov(f, g) = E[(f - E[f])(g - E[g])] = \tag{30}$$

$$= \sum_i \sum_j \frac{\partial f}{\partial X_i}\bigg|_{E[X_i]} \frac{\partial g}{\partial X_j}\bigg|_{E[X_j]} E[(X_i - E[X_i])(X_j - E[X_j])] \tag{31}$$

$$= \sum_i \sum_j \frac{\partial f}{\partial X_i}\bigg|_{E[X_i]} \frac{\partial g}{\partial X_j}\bigg|_{E[X_j]} Cov(X_i, X_j) \tag{32}$$

8.5 Covariance matrix of the probability (density) histogram

Now, the covariance matrix propagation formula can be utilized to calculate the covariance matrix of the probability (density) histogram where the correlations between bins are introduced due to normalization.

$$Cov(h_i, h_j) = \sum_{J'} \sum_{J''} \sum_{k(bins)} \sum_{l(bins)} \frac{\partial h_i}{\partial H_{k(J')}} \frac{\partial h_j}{\partial H_{l(J'')}} Cov(H_{k(J')}, H_{l(J'')}) = \tag{33}$$

$$= \sum_{J'} \sum_{k(bins)} \frac{\partial h_i}{\partial H_{k(J')}} \frac{\partial h_j}{\partial H_{k(J')}} Var(H_{k(J')}) \tag{34}$$

Now, if one is calculating the covariances between bins of the probability histogram, the following expressions need to be followed.

$$\frac{\partial h_i^p}{\partial H_{k(J')}^p} = \begin{cases} \frac{1}{N(jets)} \frac{\sum_{l(bins)} H_{l(J')}^p - H_{i(J')}^p}{\left(\sum_{l(bins)} H_{l(J')}^p\right)^2} & \text{if i=k} \\ \frac{1}{N(jets)} \frac{-H_{i(J')}^p}{\left(\sum_{l(bins)} H_{l(J')}^p\right)^2} & \text{if i}\neq\text{k} \end{cases}$$

On the other hand, if one is calculating the covariances between bins of the probability density histogram, then the following expressions need to be followed.

$$\frac{\partial h_i^d}{\partial H_{k(J')}^d} = \begin{cases} \frac{1}{N(jets)\cdot\Delta R} \frac{\sum_{l(bins)} R_l H_{l(J')}^d - R_k H_{i(J')}^d}{\left(\sum_{l(bins)} R_l H_{l(J')}^d\right)^2} & \text{if i=k} \\ \frac{1}{N(jets)\cdot\Delta R} \frac{-R_k H_{i(J')}^d}{\left(\sum_{l(bins)} R_l H_{l(J')}^d\right)^2} & \text{if i}\neq\text{k} \end{cases}$$

Because there may not be that many particles in some J' jet, the variance of $H_{k(J')}$ whose bin itself can have only one count or none, needs to be looked at closely; it cannot just be taken naively as

a sum of squared weights in that bin. There is a trick, because essentially one is dealing with an ensemble of histograms or, in other words, with a lot of jet-wise partial histograms.

$$H_k = \sum_J H_{k(J)}; \qquad k = 1, ..., N \tag{35}$$

$$S_{k_{\setminus J'}} = \sum_{J \neq J'} H_{k(J)}; \qquad k = 1, ..., N \tag{36}$$

$$H_k = H_{k(J')} + S_{k_{\setminus J'}} \qquad \rightarrow \qquad Cov(H_k, S_{k_{\setminus J'}}) = Var(S_{k_{\setminus J'}}) \tag{37}$$

$$Var(H_{k(J')}) = Var(H_k - S_{k_{\setminus J'}}) = Var(H_k) + Var(S_{k_{\setminus J'}}) - 2 \cdot Cov(H_k, S_{k_{\setminus J'}}) = \tag{38}$$

$$= Var(H_k) + Var(S_{k_{\setminus J'}}) - 2 \cdot Var(S_{k_{\setminus J'}}) = Var(H_k) - Var(S_{k_{\setminus J'}}) \tag{39}$$

Both variances from the result can now be calculated as the sum of squared weights because they are essentially the bins of two weighted histograms ($\{H_k;\ k = 1, ..., N\}$ and $\{S_{k_{\setminus J'}};\ k = 1, ..., N\}$) with many counts. As can be seen, regardless of the fact that there may be just a few counts per bin for $\{H_{k(J')};\ k = 1, ..., N\}$ histogram, variance of its k-th bin is still the sum of squared weights even if there is only one count or none.

In the case of probability, the variance of $H^p_{k(J')}$ is then calculated as follows.

$$w^p_{km(J')} = p_{T\,km(J')} \tag{40}$$

$$H^p_{k(J')} = \sum_{m(counts)} w^p_{km(J')} \tag{41}$$

$$Var(H^p_{k(J')}) = \sum_{m(counts)} [w^p_{km(J')}]^2. \tag{42}$$

On the other hand, in the case of the probability density, the variance of $H^d_{k(J')}$ is calculated as follows:

$$w^d_{km(J')} = \frac{p_{T\,km(J')}}{R_k} \tag{43}$$

$$H^d_{k(J')} = \sum_{m(counts)} w^d_{km(J')} \tag{44}$$

$$Var(H^d_{k(J')}) = \sum_{m(counts)} [w^d_{km(J')}]^2. \tag{45}$$

It can be seen now that if the weights of all the counts of a jet J' are multiplied by the same constant, the result in (34) would not change:

$$w_{km(J')} \to const(J') \cdot w_{km(J')} \tag{46}$$

$$\frac{\partial h_i}{\partial H_{k(J')}} \to \frac{1}{const(J')} \cdot \frac{\partial h_i}{\partial H_{k(J')}} \tag{47}$$

$$\frac{\partial h_j}{\partial H_{k(J')}} \to \frac{1}{const(J')} \cdot \frac{\partial h_j}{\partial H_{k(J')}} \tag{48}$$

$$Var(H_{k(J')}) \to const(J')^2 \cdot Var(H_{k(J')}) \tag{49}$$

$$\frac{\partial h_i}{\partial H_{k(J')}} \frac{\partial h_j}{\partial H_{k(J')}} Var(H_{k(J')}) \to \frac{\partial h_i}{\partial H_{k(J')}} \frac{\partial h_j}{\partial H_{k(J')}} Var(H_{k(J')}) \tag{50}$$

Namely, if one chooses this constant in a smart way, it can make numerical calculations more stable and avoid dealing with summing big weights or their squares. The constant is chosen in such a way that the new weights are now:

$$[w^p_{km(J')}]' = \frac{w^p_{km(J')}}{N(jets) \cdot \sum_{l(bins)} \sum_{n(counts)}, w^p_{ln(J')}}, \tag{51}$$

$$[w^d_{km(J')}]' = \frac{w^d_{km(J')}}{N(jets) \cdot \Delta R \cdot \sum_{l(bins)} R_l \sum_{n(counts)}, w^d_{ln(J')}}. \tag{52}$$

It can also be seen that with these new weights in place, the calculations of h^p_k and h^d_k is now the straightforward sum of all weights whose counts fall into the interval $[R_k - \Delta R, R_k + \Delta R]$:

$$h^p_k = \sum_{J'} \sum_{m} [w^p_{km(J')}]' = \tag{53}$$

$$= \frac{1}{N(jets)} \sum_{J'} \frac{\sum_m w^p_{km(J')}}{\sum_{l(bins)} \sum_{n(counts)} w^p_{ln(J')}} \tag{54}$$

$$= \frac{1}{N(jets)} \sum_{J'} \frac{H^p_{k(J')}}{\sum_{l(bins)} H^p_{l(J')}}, \tag{55}$$

$$h^d_k = \sum_{J'} \sum_{m} [w^d_{km(J')}]' = \tag{56}$$

$$= \frac{1}{N(jets)} \sum_{J'} \frac{\sum_m w^d_{km(J')}}{\Delta R \cdot \sum_{l(bins)} R_l \sum_{n(counts)} w^d_{ln(J')}} \tag{57}$$

$$= \frac{1}{N(jets)} \sum_{J'} \frac{H^d_{k(J')}}{\Delta R \cdot \sum_{l(bins)} R_l H^d_{l(J')}}. \tag{58}$$

9 Explored modeling methods/tools

The objective was to identify an analytic function that could describe the probability distribution or probability density function of the radius of QCD jets. Furthermore, the aim was to develop a parametric fit: a single functional form that could model the R distributions across narrow jet p_T bins, with parameters that would also need to be fitted for each case. This chapter provides an overview of the methods and tools that were tested.

9.1 Bayesian machine scientist

Bayesian machine scientist is a powerful package for finding analytical fit functions for complex data [6]. Python implementation can be found here, where a tutorial is also presented. In this subsection, a simple example of usage is given. As always, one has to first create a shell script.

WSL terminal:

```
nano job.sh
```

job.sh:

```
#!/bin/bash

#SBATCH --job-name=steve_job

#SBATCH --output=output.out

#SBATCH --error=error.err

#SBATCH --time=00:10:00

#SBATCH --mem=2G

#SBATCH --partition=short

#SBATCH --cpus-per-task=1

python code.py
```

WSL terminal:

```
nano code.py
```

code.py:

```
import os
import sys
sys.path.append('./machine/') #downloaded files are here

import warnings
warnings.filterwarnings('ignore')
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

from copy import deepcopy
from mcmc import * #downloaded mcmc.py file
from parallel import * #downloaded parallel.py file
from fit_prior import read_prior_par #downloaded read_prior_par.py file

print(sys.version) #3.12.3
print(f"\n",flush=True)

num_points = 20
x1 = np.random.uniform(-5, 5, num_points)
x2 = np.random.uniform(-5, 5, num_points)

# True function
y_true = 0.4 * x1**2 - 3.2167 * x2**3

# Add noise (Gaussian noise with mean 0 and standard deviation 1)
noise = np.random.normal(0, 1, size=num_points)
y_noisy = y_true + noise

x=np.zeros([num_points,2])
x[:,0]=x1
x[:,1]=x2

x=pd.DataFrame(data=x,columns=['x1','x2'])
y=pd.Series(data=y_noisy)
XLABS=['x1','x2']

# Read the hyperparameters for the prior from downloaded file
file='./machine/final_prior_param_sq.named_equations.nv13.np13.2016-09-01␣17
    _05_57.196882.dat'
prior_par = read_prior_par(file)

# Set the temperatures for the parallel tempering
Ts = [1] + [1.04**k for k in range(1, 20)]

# Initialize the parallel machine scientist
pms = Parallel(
```

```
    Ts,
    variables=XLABS,
    parameters=['a%d' % i for i in range(13)],
    x=x, y=y,
    prior_par=prior_par,
)

# Number of MCMC steps
nstep = 3000

# MCMC
description_lengths, mdl, mdl_model = [], np.inf, None
for i in range(nstep):
    # MCMC update
    pms.mcmc_step() # MCMC step within each T
    pms.tree_swap() # Attempt to swap two randomly selected consecutive temps
    # Add the description length to the trace
    description_lengths.append(pms.t1.E)
    # Check if this is the MDL expression so far
    if pms.t1.E < mdl:
        mdl, mdl_model = pms.t1.E, deepcopy(pms.t1)
    # Keep track of the progress
    #if (i+1) % 100 == 0:
    #    print(f"Step {i+1}/{nstep}")
    #    print(f"\n",flush=True)

print("Best model:\t", mdl_model)
print("Desc. length:\t", mdl)
print("Parameter values:")
print(mdl_model.par_values)
print(f"\n",flush=True)

if not os.path.exists("./Plots"):
    os.makedirs("./Plots")

plt.figure(figsize=(6, 6))
plt.scatter(mdl_model.predict(x), y)
plt.plot((y_noisy.min(),y_noisy.max()),(y_noisy.min(),y_noisy.max()))
plt.xlabel('MDL model predictions', fontsize=14)
plt.ylabel('Actual values', fontsize=14)
plt.savefig("./Plots/model.jpg")
```

WSL terminal:

```
sbatch job.sh
```

WSL terminal:

```
nano output.out
```

output.out:

```
3.12.3 | packaged by conda-forge | (main, Apr 15 2024, 18:38:13) [GCC 12.3.0]

Best model:      (((x2 * _a9_) ** 3) + ((x1 ** 2) * _a3_))
Desc. length:    65.4369873882440
Parameter values:
{'d0': {'_a3_': np.float64(0.39042008279477214), '_a9_': np.float64
    (-1.4766385375672322), '_a0_': 1.0, '_a1_': 1.0, '_a2_': 1.0, '_a4_': 1.0, '
    _a5_': 1.0, '_a6_': 1.0, '_a7_': 1.0, '_a8_': 1.0, '_a10_': 1.0, '_a11_':
    1.0, '_a12_': 1.0}}
```

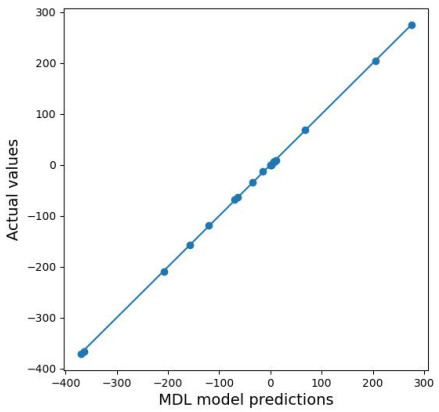

Figure 19: Quality of the returned model: model.jpg

However, as will be shown, machine scientist can sometimes return a rather unwieldy function. Moreover, since it does not account for the data uncertainties, the resulting function does not guarantee stable behavior when used later in error-weighted fitting.

9.2 Quantile transformation

Quantile transformation in Python is a technique that transforms features to follow a uniform or normal distribution by mapping the original data to quantiles. Quantiles are cut points that divide the range of a probability distribution into continuous intervals, each containing an equal probability mass. The following script helps with visualization of what the procedure can do. The scripts in this subsection were run in the Jupyter Notebook on a local computer for simplicity.

quantiles1.ipynb:

```
import os
import numpy as np
import matplotlib.pyplot as plt
from numpy import exp
from sklearn.preprocessing import QuantileTransformer

# **GENERATE SOME DATA:**

# Set a seed for reproducibility
np.random.seed(10)

n_samples=1000
# Generate Gaussian data sample
data = np.random.randn(n_samples)

# Add a skew to the data distribution
data = exp(data)

#-------------------------------------------------------------------------------

# **PLOTTING THE HISTOGRAM OF THE ORIGINAL DATA:**

if not os.path.exists("./Plots"):
    os.makedirs("./Plots")

# Histogram of the raw data with a skew
plt.figure(figsize=(6,6))
plt.hist(data, bins=25, ec='black')
plt.xlabel("data")
plt.ylabel("counts")
plt.savefig("./Plots/data.jpg")

#-------------------------------------------------------------------------------

# **PLOTTING THE HISTOGRAM OF THE TRANSFORMED DATA:**

# Reshape data to have rows and columns
data = data.reshape(-1,1)

# Quantile transform the raw data
qt = QuantileTransformer(output_distribution='normal')
data_trans = qt.fit_transform(data)

# Histogram of the transformed data
plt.figure(figsize=(6,6))
plt.hist(data_trans, bins=25, ec='black')
plt.xlabel("data_trans")
```

```
48 plt.ylabel("counts")
49 plt.savefig("./Plots/data_trans.jpg")
```

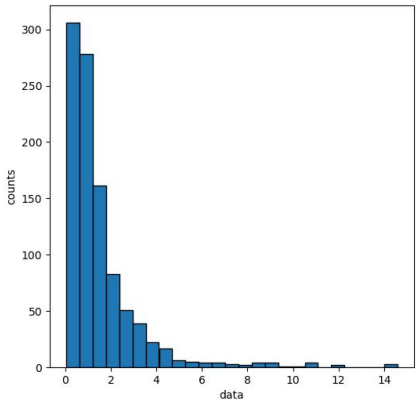

Figure 20: Original skewed data: data.jpg

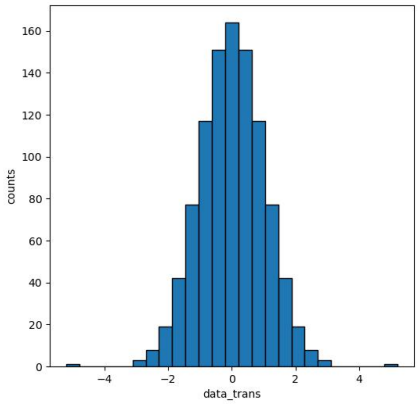

Figure 21: Transformed data: data_trans.jpg

The way this method calculates quantiles is described in detail in the following explanation and then checked in the code.

Quantiles Calculation

1. qt.fit procedure sorts the data → sorted_data
2. splitting indices of sorted_data in (n_quantiles-1) parts
   → step between two subsequent indices: (n_samples-1)/(n_quantiles-1)
3. get the indices of elements in sorted_data which will go in qt.quantiles_
   → {0,1,2,...,k,...,n_quantiles-1} · (n_samples-1)/(n_quantiles-1) =
   = {idx(0),idx(1),...,idx(k),....,idx(n_quantiles-1)}
   → observe: idx(0)=0 and idx(n_quantiles-1)=n_samples-1
   → observe: if n_quantiles=n_samples → idx(k)=k (∀ k)

4. if for example idx(k)=3.7 then:
   qt.quantiles_[k,0]=sorted_data[3]+0.7·(sorted_data[4]-sorted_data[3])
   → observe: if n_quantiles=n_samples → idx(k)=k (∀ k) → qt.quantiles_[:,0]=sorted_data

In addition to helping one understand how quantiles are calculated, the following script shows how the transformation of data is then performed.

quantiles2.ipynb:

```
import random
import numpy as np
import matplotlib.pyplot as plt
from numpy import exp
from scipy.stats import norm
from sklearn.preprocessing import QuantileTransformer

# **GENERATE SOME DATA:**

# Set a seed for reproducibility
np.random.seed(10)

n_samples=1000
# Generate Gaussian data sample
data = np.random.randn(n_samples)
# Add a skew to the data distribution
data = exp(data)
# Reshape data to have rows and columns
data = data.reshape(-1,1)

#-------------------------------------------------------------------------------

# **CALCULATING THE QUANTILES AND PERCENTILES:**

# Setting the transformer
n_quantiles=6
qt = QuantileTransformer(
    output_distribution='normal',subsample=None,n_quantiles=n_quantiles)

# Finding the quantiles
qt.fit(X=data.reshape(-1,1))

quantiles=qt.quantiles_.flatten()
percentiles=np.zeros([n_quantiles])

idx=[((n_samples-1)/(n_quantiles-1))*k for k in range(n_quantiles)]
quantiles_check=np.zeros([n_quantiles])
sorted_data=np.sort(data,axis=0).flatten()

```

```
for k in range(n_quantiles):

    percentiles[k]=k/(n_quantiles-1)

    print(f"{k}. quantile: {quantiles[k]} -> percentile: {percentiles[k]*100}%")

    if idx[k]!=int(idx[k]):

        # Manual check:
        percentage=(idx[k]-int(idx[k]))
        step=sorted_data[int(idx[k])+1]-sorted_data[int(idx[k])]
        addition=percentage*step
        quantiles_check[k]=sorted_data[int(idx[k])]+addition

    else:

        # Manual check:
        quantiles_check[k]=sorted_data[int(idx[k])]

    print(f"quantile manual check: {quantiles_check[k]}")

print(f"\n")

#-------------------------------------------------------------------------------

# **TRANSFORMING THE QUANTILES:**

for k in range(n_quantiles):

    print(f"Percentile point function check for {k}. quantile:")
    print("qt.transform(quantile):")
    print(qt.transform(np.array([quantiles[k]]).reshape(-1,1)))

    if k==0:
        print(f"norm.ppf(percentile):")
        print(norm.ppf(percentiles[k]+1e-7,loc=0,scale=1.0))
    elif k==n_quantiles-1:
        print(f"norm.ppf(percentile):")
        print(norm.ppf(percentiles[k]-1e-7,loc=0,scale=1.0))
    else:
        print(f"norm.ppf(percentile):")
        print(norm.ppf(percentiles[k],loc=0,scale=1.0))

    print(f"\n")

#-------------------------------------------------------------------------------

# **PLAYING WITH A POINT THAT IS POSITIONED BETWEEN TWO QUANTILES:**
```

```
88
89 random.seed(10)
90
91 # Point that needs to be transformed
92 # is somewhere between quantiles[k] and quantiles[k+1]:
93 k=random.randint(0,n_quantiles-2)
94 num=random.random()
95 x=quantiles[k]+num*(quantiles[k+1]-quantiles[k])
96
97 # Percentile is calculated with linear interpolation
98 a=percentiles[k]
99 b=((percentiles[k+1]-percentiles[k])/(quantiles[k+1]-quantiles[k]))
100 x0=quantiles[k]
101 percentile=a+b*(x-x0)
102 x_transformed=norm.ppf(percentile)
103
104 print(f"x={x}␣->␣percentile(x)={percentile}")
105 print(f"\n")
106 print(f"qt.transform(x):")
107 print(qt.transform(np.array(x).reshape(-1,1)))
108 print("x_transformed:")
109 print(x_transformed)
110 print(f"\n")
111
112 # Transform the point back to the original space
113 percentile=norm.cdf(x_transformed)
114 x_back=(percentile-a)/b+x0
115
116 print("qt.inverse_transform(qt.transform(x)):")
117 print(qt.inverse_transform(qt.transform(np.array(x).reshape(-1,1))))
118 print("x_back:")
119 print(x_back)
```

output:

```
0. quantile: 0.04058318972369623 -> percentile: 0.0%
quantile manual check: 0.04058318972369623
1. quantile: 0.45261614214031254 -> percentile: 20.0%
quantile manual check: 0.45261614214031254
2. quantile: 0.7880576872337507 -> percentile: 40.0%
quantile manual check: 0.7880576872337507
3. quantile: 1.2357563673533 -> percentile: 60.0%
quantile manual check: 1.2357563673533
4. quantile: 2.1275282556894264 -> percentile: 80.0%
quantile manual check: 2.1275282556894264
5. quantile: 14.583785188055503 -> percentile: 100.0%
quantile manual check: 14.583785188055503
```

```
Percentile point function check for 0. quantile:
qt.transform(quantile):
[[-5.19933758]]
norm.ppf(percentile):
-5.1993375821928165


Percentile point function check for 1. quantile:
qt.transform(quantile):
[[-0.84162123]]
norm.ppf(percentile):
-0.8416212335729142


Percentile point function check for 2. quantile:
qt.transform(quantile):
[[-0.2533471]]
norm.ppf(percentile):
-0.2533471031357997


Percentile point function check for 3. quantile:
qt.transform(quantile):
[[0.2533471]]
norm.ppf(percentile):
0.2533471031357997


Percentile point function check for 4. quantile:
qt.transform(quantile):
[[0.84162123]]
norm.ppf(percentile):
0.8416212335729143


Percentile point function check for 5. quantile:
qt.transform(quantile):
[[5.19933758]]
norm.ppf(percentile):
5.199337582290661


x=2.5334162010006205 -> percentile(x)=0.806517013056411


qt.transform(x):
[[0.86513267]]
```

```
x_transformed:
0.8651326722548073


qt.inverse_transform(qt.transform(x)):
[[2.5334162]]
x_back:
2.533416201000622
```

The next presented script aims to try to use this technique in a non-conventional way to fit the original data, to be precise, a weighted histogram. Since the method only works with non-weighted data, sampling has to be done first. Once the sampling is done, the quantile transformation can be applied, and the transformed data can be fitted. After fitting the transformed data with a known distribution, one can then sample from it and transfer these newly created samples back to the original space. The script uses the module `data_inspect.py` to read the data and create a histogram that represents the radius probability distribution of a jet p_T bin.

data_inspect.py:

```python
import h5py
import numpy as np

# **READING FROM A FILE:**

def load_cnsts_rodem(ifile: str, mask_pt: list):
    """Load constituents from an HDF5 file."""
    with h5py.File(ifile, "r") as f:

        cnsts=f["objects/jets/jet1_cnsts"][mask_pt,:,:3] #first three features

    mask = np.repeat(cnsts[:, :, 0] == 0, cnsts.shape[2])
    mask = mask.reshape(-1, cnsts.shape[1], cnsts.shape[2])
    cnsts = np.ma.masked_where(mask, cnsts)

    print(f"cnsts.shape={cnsts.shape}")
    #(num_jets,num_particles,num_particle_features)
    print(f"\n")

    return cnsts[:,:,0],cnsts[:,:,1],cnsts[:,:,2] #pt,eta,phi

def load_jets_rodem(ifile: str, mask_pt: list):
    with h5py.File(ifile, "r") as f:

        jets = f["objects/jets/jet1_obs"][mask_pt,:3] #first three features

    print(f"jets.shape={jets.shape}")
```

```
    #(num_jets,num_jet_features)
    print(f"\n")

    return jets[:,0],jets[:,1],jets[:,2] #pt,eta,phi

#--------------------------------------------------------------------------------

# **ARRANGE DATA IN A DICTIONARY:**

def read_data(input,mask_pt):

    jets={}

    #rows -> jets ; columns -> particles
    jets['part_pt'],jets['part_eta'],jets['part_phi']=load_cnsts_rodem(input,
        mask_pt)

    #flat arrays
    jets['jet_pt'],jets['jet_eta'],jets['jet_phi']=load_jets_rodem(input,mask_pt
        )

    jets['jet_nparticles']=jets['part_pt'].count(axis=1)

    print(f"The␣smallest␣jet_pt␣is:␣{np.min(jets['jet_pt'])}")
    print(f"The␣biggest␣jet_pt␣is:␣{np.max(jets['jet_pt'])}")
    print(f"\n")

    return jets

#--------------------------------------------------------------------------------

# **CREATE A HISTOGRAM AND CALCULATE ITS COVARIANCE MATRIX:**

def hist_cov(R,bins_r,weights):

    # **MAKING A HISTOGRAM:**

    n_jets=len(R)

    R_middles = (bins_r[1:]+bins_r[:-1])/2.
    num_R_bins=len(R_middles)

    weights=np.ma.array(weights/(n_jets*np.sum(weights,axis=1).reshape(-1,1)),
                        dtype=np.float64)

    W=np.sum(weights,axis=1)

    hist=np.zeros([num_R_bins],dtype=np.float64)
```

```
    hist, _ = np.histogram(R.compressed(),
                           bins=bins_r,weights=weights.compressed())

    #-------------------------------------------------------------------------

    # **COMPUTE THE COVARIANCE MATRIX:**

    H=np.zeros([n_jets,num_R_bins],dtype=np.float64)
    H2=np.zeros([n_jets,num_R_bins],dtype=np.float64)

    for J in range(n_jets):

        H[J,:], _ = np.histogram(R[J].compressed(),
                            bins=bins_r,weights=weights[J].compressed())
        H2[J,:], _ = np.histogram(R[J].compressed(),
                            bins=bins_r,weights=weights[J].compressed()**2)

    a=-H[:,:,np.newaxis]*np.ones(num_R_bins)

    mask=np.eye(num_R_bins,dtype=bool)

    a[:,mask]+=np.repeat(W,num_R_bins).reshape(a.shape[0],a.shape[1])

    a=a/(n_jets*(W ** 2)[:, np.newaxis, np.newaxis])

    b=a

    cov=np.zeros([num_R_bins,num_R_bins],dtype=np.float64)
    cov=np.einsum('mik,mjk,mk->ij', a, b, H2)

    """
    for i in range(num_R_bins):

        for j in range(num_R_bins):

            for J in range(n_jets):

                for k in range(num_R_bins):

                    if k==i:
                        a=(W[J]-H[J,i])/(n_jets*W[J]**2)
                    else:
                        a=-H[J,i]/(n_jets*W[J]**2)

                    if k==j:
                        b=(W[J]-H[J,j])/(n_jets*W[J]**2)
                    else:
                        b=-H[J,j]/(n_jets*W[J]**2)
```

```
                    cov[i,j]+=a*b*H2[J,k]
    """

    #-------------------------------------------------------------------------

    return hist,cov
```

quantiles3.ipynb:

```
import sys
sys.path.append("./")

import os
import h5py
import numpy as np
import matplotlib.pyplot as plt
from data_inspect import *
from scipy.stats import norm
from sklearn.preprocessing import QuantileTransformer

# **SETTING THE DATA STRUCTURE:**

# The input HDF5 file containing the relevant jets:
input = "QCDjj_pT_450_1200_train01.h5" #change if needed

# The number of jets to load:
n_jets = 100_000 #change if needed

# Jet pt range:
min_pt=450 #change if needed
max_pt=1000 #change if needed
width_pt=10

# Choosing only the jets from the set jet pt range
with h5py.File(input, "r") as f:

    mask_pt=[index for index,value in enumerate(
        f["objects/jets/jet1_obs"][:n_jets,0])
             if ((value>=min_pt) and (value<=max_pt))]

R_max=1.0
width=0.01

jets=read_data(input,mask_pt)

#-------------------------------------------------------------------------

```

```
# **R DISTRIBUTION OF JETS:**

#phi angle differences between the constituents and the associated jet:
#rows -> jets ; columns -> particles
delta_phi=jets['jet_phi'].reshape(-1,1)-jets['part_phi']

#because the phi angle difference should be inside [-np.pi,+np.pi]:
delta_phi_new=np.ma.mod(delta_phi+np.pi,2*np.pi)-np.pi

#eta angle differences between the constituents and the associated jet:
#rows -> jets ; columns -> particles
delta_eta=jets['jet_eta'].reshape(-1,1)-jets['part_eta']

#rows -> jets ; columns -> particles
R=np.ma.sqrt(delta_eta**2+delta_phi_new**2)

print(f"number␣of␣particles␣loaded:␣\
{len(R.compressed())}")

print(f"number␣of␣particles␣with␣R=0:␣\
{len(np.where(R.compressed()==0.0)[0])}")

print(f"percentage␣of␣particles␣with␣R=0:␣\
{np.round(len(np.where(R.compressed()==0.0)[0])*100/len(R.compressed()),3)}%")

print(f"number␣of␣particles␣with␣R>={R_max}:␣\
{len(np.where(R.compressed()>=R_max)[0])}")

print(f"percentage␣of␣particles␣with␣R>={R_max}:␣\
{np.round(len(np.where(R.compressed()>=R_max)[0])*100/len(R.compressed()),3)}%")

print(f"\n")

jets['part_pt']=np.ma.masked_where(R<0,jets['part_pt'])
jets['part_pt']=np.ma.masked_where(R>=R_max,jets['part_pt'])

R=np.ma.masked_where(R<0,R)
R=np.ma.masked_where(R>=R_max,R)

#-------------------------------------------------------------------------------

# **SETTING pT BINS:**

bins_pt=np.arange(min_pt,max_pt+width_pt,width_pt)
pt_middles = (bins_pt[1:]+bins_pt[:-1])/2.
num_pt_bins=len(pt_middles)

print(f"num_pt_bins={num_pt_bins}")
```

```
print(f"\n")

#------------------------------------------------------------------------------

# **SETTING R BINS:**

bins_r=np.arange(0,R_max,width)
R_middles = (bins_r[1:]+bins_r[:-1])/2.
num_R_bins=len(R_middles)

print(f"num_R_bins={num_R_bins}")
print(f"\n",flush=True)

#------------------------------------------------------------------------------

mask_binpt=[]

for k in range(num_pt_bins):

        mask_binpt.append([index for index,value in enumerate(
            jets['jet_pt']) if ((value>bins_pt[k])
                                and (value<=bins_pt[k+1]))])

num_jets=[len(sublist) for sublist in mask_binpt]

#------------------------------------------------------------------------------


# **CREATING THE HISTOGRAM AND CALCULATING ITS COVARIANCE MATRIX**

idx=2

weights=jets['part_pt'][mask_binpt[idx]]

hist,cov_matrix=hist_cov(R[mask_binpt[idx]],bins_r,weights)

#------------------------------------------------------------------------------

# **SAMPLING:**

weights=weights/(num_jets[idx]*np.sum(weights,axis=1).reshape(-1,1))

n_samples = 500000
data = np.random.choice(R[mask_binpt[idx],:].compressed(),
                        size=n_samples,p=weights.compressed())

#------------------------------------------------------------------------------

```

```
# **SAMPLING VISUALIZED:**

if not os.path.exists("./Plots"):
    os.makedirs("./Plots")

fig,ax=plt.subplots(3,1,figsize=(6,15))

ax[0].set_title(f"{num_jets[idx]} jets with pT in range "
                f"{bins_pt[idx]}-{bins_pt[idx+1]} GeV")
ax[0].bar(R_middles,hist,width=width,ec='black')
ax[0].set_xlabel("R - original")
ax[0].set_ylabel("original weights")

ax[1].set_title(f"{num_jets[idx]} jets with pT in range "
                f"{bins_pt[idx]}-{bins_pt[idx+1]} GeV")
ax[1].hist(data,bins=bins_r,ec='black')
ax[1].set_xlabel("R - samples")
ax[1].set_ylabel("counts")

a=1/n_samples*np.ones(shape=n_samples)
ax[2].set_title(f"{num_jets[idx]} jets with pT in range "
                f"{bins_pt[idx]}-{bins_pt[idx+1]} GeV")
ax[2].hist(data,bins=bins_r,weights=a,ec='black')
ax[2].set_xlabel("R - samples")
ax[2].set_ylabel("weights = counts / n_samples")

fig.savefig("./Plots/sampling.jpg")

#-------------------------------------------------------------------------------

# **TRANSFORMING SAMPLED R, PLOTTING IT AND FITTING**:

# Choose either "uniform" or "normal":
choice="normal"

# Setting the transformer
n_quantiles=100
qt = QuantileTransformer(
    output_distribution=choice,subsample=None,n_quantiles=n_quantiles)

# Transforming:
data_trans = qt.fit_transform(X=data.reshape(-1,1))

fig,ax=plt.subplots(1,2,figsize=(12,5))

ax[0].set_title(f"{num_jets[idx]} jets with pT in range "
                f"{bins_pt[idx]}-{bins_pt[idx+1]} GeV")
ax[0].hist(data_trans,bins=len(bins_r),ec='black')
```

```
ax[0].set_xlabel("R - transformed samples")
ax[0].set_ylabel("counts")

if choice=="uniform":
    # Fitting uniform:
    x=np.linspace(min(data_trans.flatten()),max(data_trans.flatten()),10000)
    pdf=np.ones(len(x))*1/(max(data_trans.flatten())-min(data_trans.flatten()))

elif choice=="normal":
    # Fitting a Gaussian:
    x=np.linspace(min(data_trans.flatten()),max(data_trans.flatten()),10000)
    mu,sigma = norm.fit(data_trans)
    pdf = norm.pdf(x,loc=mu,scale=sigma)

ax[1].set_title(f"{num_jets[idx]} jets with pT in range "
                f"{bins_pt[idx]}-{bins_pt[idx+1]} GeV")
ax[1].plot(x, pdf, 'r-')
ax[1].hist(data_trans,bins=len(bins_r),ec='black',density=True)
ax[1].set_xlabel("R - transformed samples")
ax[1].set_ylabel("weights = counts / [sum(counts) * width]")

fig.savefig(f"./Plots/trans_{choice}.jpg")

#-------------------------------------------------------------------------------

# **SAMPLING FROM THE FITTED DISTRIBUTION:**

prob=pdf/np.sum(pdf)
samples = np.random.choice(x,size=n_samples,p=prob)

plt.figure(figsize=(6,5))
plt.hist(samples,bins=len(bins_r),density=True,ec='black')
plt.plot(x,pdf,'r-')
plt.xlabel(f"samples from the {choice} distribution fit")
plt.ylabel("weights = counts / [sum(counts) * width]")
plt.savefig(f"./Plots/samples_{choice}.jpg")

#-------------------------------------------------------------------------------

# **TRASNFORMING THE FIT BACK TO THE ORIGINAL SPACE**:

samples_back=qt.inverse_transform(samples.reshape(-1,1))

hist_fit=[]
middles_fit=[]

# Making the fit as smooth as possible
dif=0.001
```

```
for i in range(0,70):
    h, be = np.histogram(samples_back.flatten(),
                         bins=len(bins_r),weights=a,range=(0+dif*i,R_max+dif*i))
    hist_fit.extend(h)
    bm = (be[1:]+be[:-1])/2.
    middles_fit.extend(bm)

sorting=np.argsort(np.argsort(middles_fit))
hist_fit = [val for _, val in sorted(zip(sorting,hist_fit))]
middles_fit=[val for _, val in sorted(zip(sorting,middles_fit))]

#-------------------------------------------------------------------------------

# **PLOT OF THE FIT IN THE ORIGINAL SPACE**:

plt.figure(figsize=(6,5))
plt.title(f"{num_jets[idx]} jets with pT in range "
          fr"{bins_pt[idx]}-{bins_pt[idx+1]} GeV $\rightarrow$ {choice}, "
          f"n_quantiles={n_quantiles}")
plt.scatter(R_middles,hist,color='b',marker='x')
plt.plot(middles_fit,hist_fit,'r-')
plt.xlabel("R - original")
plt.ylabel("original weights")
plt.savefig(f"./Plots/fit_{choice}.jpg")
```

output:

```
cnsts.shape=(98758, 100, 3)

jets.shape=(98758, 3)

The smallest jet_pt is: 450.00129118847076
The biggest jet_pt is: 999.8620172940161

number of particles loaded: 5365439
number of particles with R=0: 1
percentage of particles with R=0: 0.0%
number of particles with R>=1.0: 14630
percentage of particles with R>=1.0: 0.273%

num_pt_bins=55

num_R_bins=99
```

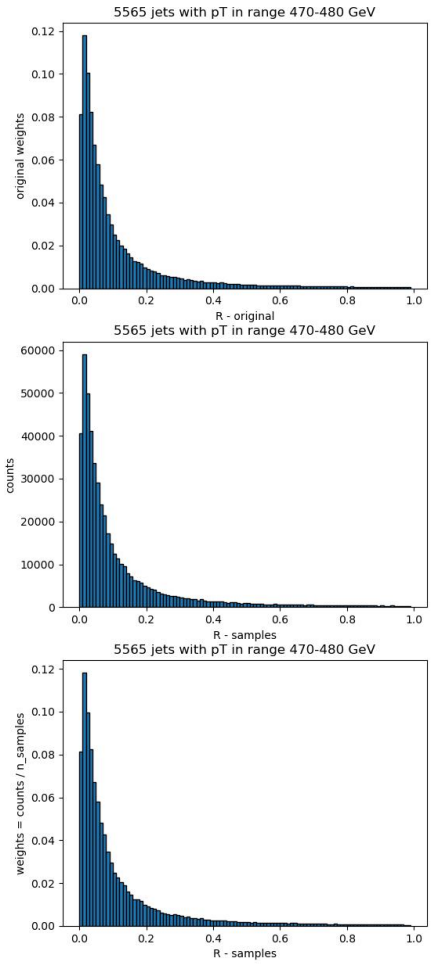

Figure 22: Sampling of the weighted data: sampling.jpg

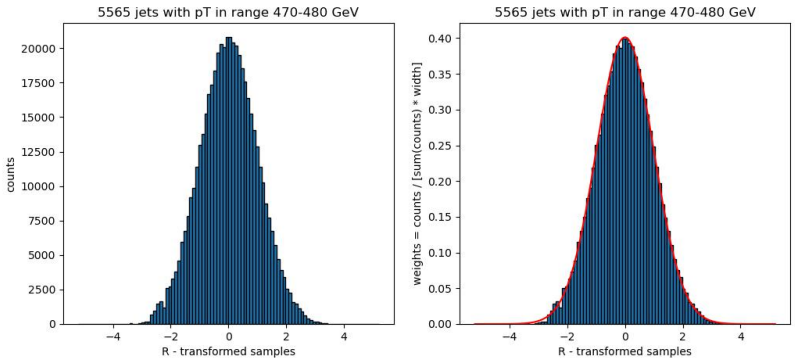

Figure 23: Fitting of transformed sampled data: trans_normal.jpg

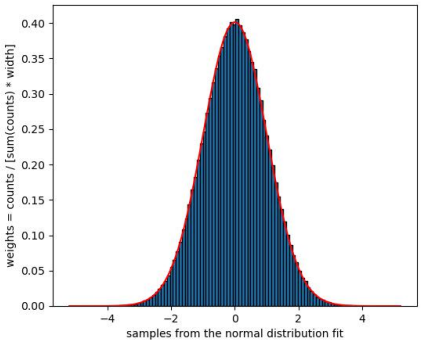

Figure 24: Sampling from the fitted distribution: samples_normal.jpg

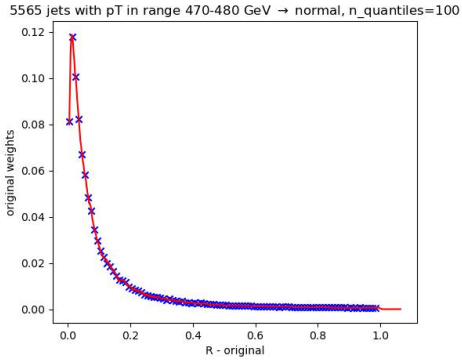

Figure 25: Fitted distribution transformed back to the original space: fit_normal.jpg

As one can see from the figures above, the fit of the known distribution transformed back to the original space is quite good for the chosen jet p_T bin. However, the problem is that this method is nonparametric, so one does not end up with some expression for the fit function.

9.3 Box-Cox transformation

In an attempt to find some parametric method of transforming the data, fitting it and then transforming the fit back to the original space, the Box-Cox transformation was found and explored. The transformation is given by the following expression.

$$y = \begin{cases} \frac{(x^\lambda - 1)}{\lambda}, \lambda \neq 0 \\ log(x), \lambda = 0 \end{cases}$$

The algorithm finds the best parameter λ when the data is loaded. The goal of the transformation is to end up with the normal distribution of the data. Compared to the quantile transformation, this transformation is parametric. Moreover, unlike the quantile transformation, it preserves the data structure - there is no sorting of the data before the transformation is applied. However, the quantile transformation technically always works, while the Box-Cox transformation does not.

The script in this subsection was also run in the Jupyter Notebook on a local computer for simplicity. The first part of the script mirrors the procedure introduced in `quantiles3.ipynb`.

boxcox.ipynb:

```
import sys
sys.path.append("./")

import os
import h5py
import numpy as np
import matplotlib.pyplot as plt
from data_inspect import *
from inspect import signature
from scipy.stats import norm,boxcox
from scipy.special import inv_boxcox
from scipy.optimize import curve_fit

# **SETTING THE DATA STRUCTURE:**

# The input HDF5 file containing the relevant jets:
input = "QCDjj_pT_450_1200_train01.h5" #change if needed

# The number of jets to load:
n_jets = 100_000 #change if needed

# Jet pt range:
min_pt=450 #change if needed
max_pt=1000 #change if needed
width_pt=10

# Choosing only the jets from the set jet pt range
with h5py.File(input, "r") as f:

    mask_pt=[index for index,value in enumerate(
        f["objects/jets/jet1_obs"][:n_jets,0])
             if ((value>=min_pt) and (value<=max_pt))]

R_max=1.0
width=0.01

jets=read_data(input,mask_pt)

#------------------------------------------------------------------------------

# **R DISTRIBUTION OF JETS:**

#phi angle differences between the constituents and the associated jet:
#rows -> jets ; columns -> particles
delta_phi=jets['jet_phi'].reshape(-1,1)-jets['part_phi']

#because the phi angle difference should be inside [-np.pi,+np.pi]:
```

```
delta_phi_new=np.ma.mod(delta_phi+np.pi,2*np.pi)-np.pi

#eta angle differences between the constituents and the associated jet:
#rows -> jets ; columns -> particles
delta_eta=jets['jet_eta'].reshape(-1,1)-jets['part_eta']

#rows -> jets ; columns -> particles
R=np.ma.sqrt(delta_eta**2+delta_phi_new**2)

print(f"number␣of␣particles␣loaded:␣\
{len(R.compressed())}")

print(f"number␣of␣particles␣with␣R=0:␣\
{len(np.where(R.compressed()==0.0)[0])}")

print(f"percentage␣of␣particles␣with␣R=0:␣\
{np.round(len(np.where(R.compressed()==0.0)[0])*100/len(R.compressed()),3)}%")

print(f"number␣of␣particles␣with␣R>={R_max}:␣\
{len(np.where(R.compressed()>=R_max)[0])}")

print(f"percentage␣of␣particles␣with␣R>={R_max}:␣\
{np.round(len(np.where(R.compressed()>=R_max)[0])*100/len(R.compressed()),3)}%")

print(f"\n")

jets['part_pt']=np.ma.masked_where(R<0,jets['part_pt'])
jets['part_pt']=np.ma.masked_where(R>=R_max,jets['part_pt'])

R=np.ma.masked_where(R<0,R)
R=np.ma.masked_where(R>=R_max,R)

#----------------------------------------------------------------------------

# **SETTING pT BINS:**

bins_pt=np.arange(min_pt,max_pt+width_pt,width_pt)
pt_middles = (bins_pt[1:]+bins_pt[:-1])/2.
num_pt_bins=len(pt_middles)

print(f"num_pt_bins={num_pt_bins}")
print(f"\n")

#----------------------------------------------------------------------------

# **SETTING R BINS:**

bins_r=np.arange(0,R_max,width)
```

```
R_middles = (bins_r[1:]+bins_r[:-1])/2.
num_R_bins=len(R_middles)

print(f"num_R_bins={num_R_bins}")
print(f"\n",flush=True)

#-------------------------------------------------------------------------------

mask_binpt=[]

for k in range(num_pt_bins):

        mask_binpt.append([index for index,value in enumerate(
            jets['jet_pt']) if ((value>bins_pt[k])
                                and (value<=bins_pt[k+1]))])

num_jets=[len(sublist) for sublist in mask_binpt]

#-------------------------------------------------------------------------------

# **CREATING THE HISTOGRAMS AND CALCULATING THEIR COVARIANCE MATRICES:**

hist=np.zeros([num_pt_bins,num_R_bins])
cov_matrix=np.zeros([num_pt_bins,num_R_bins,num_R_bins])

for idx in range(num_pt_bins):

    print(f"hist_cov␣of␣bin␣{idx}/{num_pt_bins-1}␣in␣progress..")

    weights=jets['part_pt'][mask_binpt[idx]]

    hist[idx],cov_matrix[idx]=hist_cov(R[mask_binpt[idx]],bins_r,weights)

print(f"\n")

#-------------------------------------------------------------------------------

# **SAMPLING:**

idx=2

weights=jets['part_pt'][mask_binpt[idx]]
weights=weights/(num_jets[idx]*np.sum(weights,axis=1).reshape(-1,1))

n_samples = 500000
data = np.random.choice(R[mask_binpt[idx]].compressed(),
                        size=n_samples,p=weights.compressed())

```

```
#-------------------------------------------------------------------------------

# **TRANSFORMING SAMPLED R, PLOTTING IT AND FITING:**

# Transforming:
data_trans,best_lambda = boxcox(data)

if not os.path.exists("./Plots"):
        os.makedirs("./Plots")

fig,ax=plt.subplots(1,2,figsize=(12,5))

ax[0].set_title(f"{num_jets[idx]} jets with pT in range "
                f"{bins_pt[idx]}-{bins_pt[idx+1]} GeV")
ax[0].hist(data_trans,bins=len(bins_r),ec='black')
ax[0].set_xlabel("R - transformed samples")
ax[0].set_ylabel("counts")

# Fitting a Gaussian:
x=np.linspace(min(data_trans.flatten()),max(data_trans.flatten()),10000)
mu,sigma = norm.fit(data_trans)
pdf = norm.pdf(x,loc=mu,scale=sigma)

ax[1].set_title(f"{num_jets[idx]} jets with pT in range "
                f"{bins_pt[idx]}-{bins_pt[idx+1]} GeV")
ax[1].plot(x,pdf,'r-')
ax[1].hist(data_trans,bins=len(bins_r),ec='black',density=True)
ax[1].set_xlabel("R - transformed samples")
ax[1].set_ylabel("weights = counts / [sum(counts) * width]")

fig.savefig("./Plots/trans_boxcox.jpg")

#-------------------------------------------------------------------------------

# **SAMPLING FROM THE GAUSSIAN:**

prob=pdf/np.sum(pdf)
samples = np.random.choice(x,size=n_samples,p=prob)

plt.figure(figsize=(6,5))
plt.hist(samples,bins=len(bins_r),density=True,ec='black')
label="Fitted Gaussian"
label+=f"\n"
label+=fr"$\mu$={mu:.2f}, $\sigma$={sigma:.2f}"
plt.plot(x,pdf,'r-',label=label)
plt.legend()
plt.savefig(f"./Plots/samples_boxcox.jpg")

```

```
#------------------------------------------------------------------------------

# **TRASNFORMING THE FIT BACK TO THE ORIGINAL SPACE**:

samples_back = inv_boxcox(samples,best_lambda)

hist_fit=[]
middles_fit=[]

# Making the fit as smooth as possible
dif=0.001
a=1/n_samples*np.ones(shape=n_samples)
for i in range(0,70):
    h, be = np.histogram(samples_back,
                         bins=len(bins_r),weights=a,range=(0+dif*i,R_max+dif*i))
    hist_fit.extend(h)
    bm = (be[1:]+be[:-1])/2.
    middles_fit.extend(bm)

sorting=np.argsort(np.argsort(middles_fit))
hist_fit = [val for _, val in sorted(zip(sorting,hist_fit))]
middles_fit=[val for _, val in sorted(zip(sorting,middles_fit))]

#------------------------------------------------------------------------------

# **PLOT OF THE FIT IN THE ORIGINAL SPACE**:

plt.figure(figsize=(6,5))
plt.title(f"{num_jets[idx]} jets with pT in range "
          f"{bins_pt[idx]}-{bins_pt[idx+1]} GeV")
plt.scatter(R_middles,hist[idx],color='b',marker='x')
plt.plot(middles_fit,hist_fit,'r-')
plt.xlabel("R - original")
plt.ylabel("original weights")
plt.savefig("./Plots/fit_boxcox.jpg")

#------------------------------------------------------------------------------

# **FITTING WITH THE NEWLY DISCOVERED FUNCTION AND PREPARING FOR THE PLOTS:**

def model(x,lmbda,loc,scale):
    return norm.pdf(boxcox(x,lmbda=lmbda),loc=loc,scale=scale)

rs=np.linspace(min(R_middles),max(R_middles),1000)
p0=[0.05,-2.5,1.2]

num_params=len(signature(model).parameters)-1

```

```
params=np.zeros([num_pt_bins,num_params])
params_err=np.zeros([num_pt_bins,num_params])
y_mean=np.zeros([num_pt_bins,len(rs)])
y_std=np.zeros([num_pt_bins,len(rs)])

for idx in range(num_pt_bins):

        print(f"fitting␣of␣bin␣{idx}/{num_pt_bins-1}␣in␣progress..")

        params[idx],cov = curve_fit(
            model,xdata=R_middles,ydata=hist[idx,:],
            sigma=cov_matrix[idx],p0=p0)

        params_err[idx] = np.sqrt(np.diag(cov))

        n_samples=1000
        params_samples=np.random.multivariate_normal(
            params[idx,:],cov,size=n_samples)
        y_samples=np.array([model(rs,*sample) for sample in params_samples])
        y_mean[idx,:]=np.mean(y_samples,axis=0)
        y_std[idx,:]=np.std(y_samples,axis=0)

print(f"\n")

#------------------------------------------------------------------------------

# **PLOT THE FIT OF SOME pT BIN:**

def digit(x):

    str="{:.10e}".format(x)

    base,exp=str.split('e')
    exp=int(exp)

    return -exp

if not os.path.exists("./Plots"):
        os.makedirs("./Plots")

idx=15

fig,ax=plt.subplots(figsize=(10,6))

label=r"$\frac{1}{\sqrt{2\pi\sigma^2}}$"
label+="␣"
label+=r"$e^{\frac{(\frac{R^\lambda-1}{\lambda}-\mu)^2}{2\sigma^2}}$"
label+=f"\n"
```

```
err_rounded=round(params_err[idx,0],digit(params_err[idx,0]))
mean_rounded=round(params[idx,0],digit(err_rounded))
label+=fr"$\lambda$={mean_rounded}$\pm${err_rounded};"
label+=f"\n"
err_rounded=round(params_err[idx,1],digit(params_err[idx,1]))
mean_rounded=round(params[idx,1],digit(err_rounded))
label+=fr"$\mu$={mean_rounded}$\pm${err_rounded};"
label+=f"\n"
err_rounded=round(params_err[idx,2],digit(params_err[idx,2]))
mean_rounded=round(params[idx,2],digit(err_rounded))
label+=fr"$\sigma$={mean_rounded}$\pm${err_rounded};"

ax.set_title(f"{num_jets[idx]} jets with pT in range "
             f"{bins_pt[idx]}-{bins_pt[idx+1]} GeV")
ax.errorbar(R_middles,hist[idx],np.sqrt(np.diag(cov_matrix[idx])),
            ls='none',capsize=2)
ax.scatter(R_middles,hist[idx],color='b',s=4)
ax.plot(rs,y_mean[idx],color='red',label=label)
ax.fill_between(rs,y_mean[idx]-y_std[idx],y_mean[idx]+y_std[idx],
                color='red',alpha=0.3)
ax.set_xlabel("R")
ax.set_ylabel("P - probability")
ax.legend()
fig.savefig("./Plots/norm.pdf(boxcox).jpg")

#-------------------------------------------------------------------------------

# **PLOTTING AND FITTING OF THE PARAMETERS:**

def lin(x,a,b):
    x=x-np.min(pt_middles)
    x=x/(np.max(pt_middles)-np.min(pt_middles))
    return a+b*x

pts=np.linspace(min(pt_middles),max(pt_middles),1000)

p=[r"$\lambda$",r"$\mu$",r"$\sigma$"]

fig,ax=plt.subplots(3,1,figsize=(6,15))

for n,s in enumerate(p):

    pars,cov=curve_fit(
        lin,xdata=pt_middles,ydata=params[:,n],sigma=params_err[:,n])

    pars_err=np.sqrt(np.diag(cov))

    n_samples=1000
```

```
336     pars_samples=np.random.multivariate_normal(pars,cov,size=n_samples)
337     param_samples=np.array([lin(pts,*sample) for sample in pars_samples])
338     param_mean=np.mean(param_samples,axis=0)
339     param_std=np.std(param_samples,axis=0)
340 
341     label=r"$f(p_T)$=a+b$\cdot$"
342     label+=fr"$\frac{{p_T-{np.min(pt_middles)}}}{{{np.max(pt_middles)-np.min(
            pt_middles)}}}$;"
343 
344     label+=f"\n"
345     pars_err[0]=round(pars_err[0],digit(pars_err[0]))
346     pars[0]=round(pars[0],digit(pars_err[0]))
347     label+=fr"a={pars[0]}$\pm${pars_err[0]};"
348     label+=f"\n"
349     pars_err[1]=round(pars_err[1],digit(pars_err[1]))
350     pars[1]=round(pars[1],digit(pars_err[1]))
351     label+=fr"b={pars[1]}$\pm${pars_err[1]}"
352 
353     ax[n].errorbar(pt_middles,params[:,n],params_err[:,n],
354                    ls='none',capsize=2)
355     ax[n].scatter(pt_middles,params[:,n],s=4)
356     ax[n].plot(pts,param_mean,color='red',label=label)
357     ax[n].fill_between(pts,param_mean-param_std,param_mean+param_std,
358                        color='red',alpha=0.3)
359     ax[n].set_xlabel(r"jet␣$p_T$")
360     ax[n].set_ylabel(f"{s}")
361     ax[n].legend()
362 
363 fig.savefig(f"./Plots/norm.pdf(boxcox)_params.jpg")
```

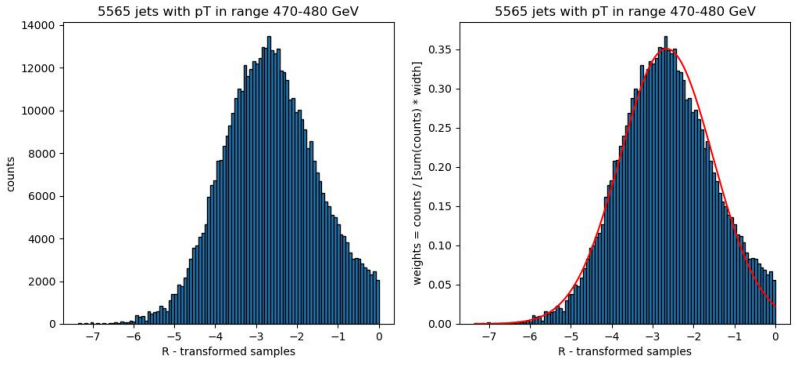

Figure 26: Fitting of transformed sampled data: trans_boxcox.jpg

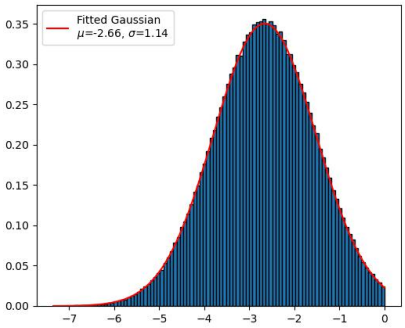

Figure 27: Sampling from the fitted distribution: samples_boxcox.jpg

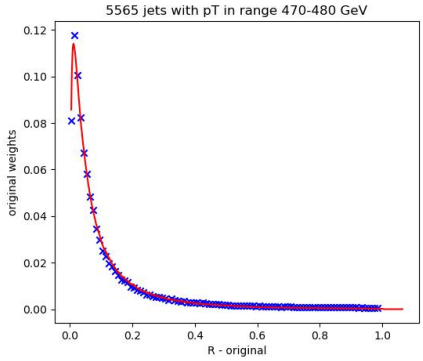

Figure 28: Fitted distribution transformed back to the original space: fit_boxcox.jpg

Still, even with the parametric transformation of the data samples to the space where the sample points are normally distributed, one key problem remains. Both the quantile transformation and the Box-Cox transformation work with the samples rather than with the actual data. The problem becomes the most apparent when one has to sample from the fitted distribution to then transfer these new samples back to the original space; the fit itself is not directly inversely transformed, but the samples of it. This does not allow one to find the expression of the final fit in the original space, even though the transformation is parametric. While analyzing the Box-Cox transformation, one function form that follows the data shape was discovered by accident and explored in the second part of the script.

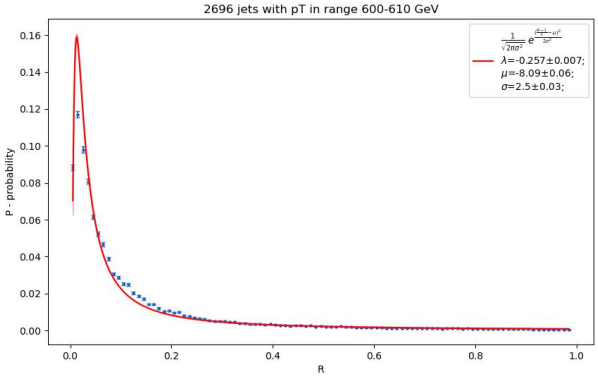

Figure 29: Probability fitted with the newly discovered function: norm.pdf(boxcox).jpg

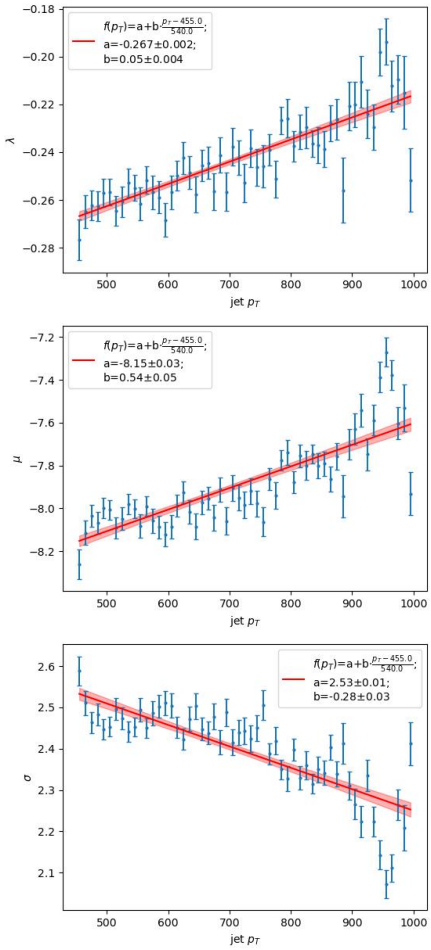

Figure 30: The parameters fitted across jet p_T bins: norm.pdf(boxcox)_params.jpg

As can be seen from the figure above, the newly found function gave some hope but it tends to overshoot the peak.

9.4 Kernel density estimation

Kernel density estimation is a method for estimating the probability density function of a continuous variable by smoothing the data using a kernel function. It works only with the data samples, so again, sampling has to be done first to apply the method. The procedure places a smooth kernel function on each sample point and sums them up. The number of kernels then equals the number of samples. Because of this, kernel density estimation is not a fitting method in the traditional sense - it does not assume an underlying parametric model or learn parameters from data, but instead builds the estimate directly from the sample distribution, making it a purely data-driven, nonparametric approach. Although not a traditional fitting method, the underlying procedure was still explored in this subsection and in the following script.

kde.ipynb:

```
import os
import numpy as np
import matplotlib.pyplot as plt
from sklearn.neighbors import KernelDensity

# **THE SETUP:**

# The kernel function
def K(x):
    return np.exp(-x**2/2)/np.sqrt(2*np.pi)

# Dummy dataset
dataset = np.array([1.33,1.33,1.33,0.3,0.97,0.97,0.96,1.1,0.1,1.4,0.4])

# x-value range for plotting KDEs
x_range = np.linspace(dataset.min()-0.3,dataset.max()+0.3,num=600)

# Bandwith values for experimentation
H = [0.3,0.1,0.03]
n_samples = dataset.size

# Line properties for different bandwith values
color_list = ['green','black','maroon']
alpha_list = [0.8,1,0.8]
width_list = [1.7,2.5,1.7]

#-------------------------------------------------------------------------------

# **CALCULATION AND PLOTTING:**

if not os.path.exists("./Plots"):
    os.makedirs("./Plots")

plt.figure(figsize=(10,4))

# Iterate over bandwith values
for h, color, alpha, width in zip(H,color_list,alpha_list,width_list):

    # **METHOD DONE MANUALLY:**

    total_sum = 0
    # Iterate over datapoints
    for i, xi in enumerate(dataset):
        total_sum += K((x_range-xi)/h)

    # Normalization
    y_range = total_sum/(h*n_samples)
```

```
48
49      # Plotting
50      plt.plot(x_range, y_range,
51          color=color,alpha=alpha,linewidth=width,label=f"{h}")
52
53      #-------------------------------------------------------------------------
54
55      # **BUILT-IN METHOD:**
56
57      kde = KernelDensity(
58          kernel='gaussian',bandwidth=h).fit(dataset[:,np.newaxis])
59
60      log_density = kde.score_samples(x_range[:,np.newaxis])
61
62      # Plotting
63      plt.plot(x_range,np.exp(log_density),
64          color='grey',alpha=0.8,linewidth=width,linestyle='--',label=f"{h}")
65
66      #-------------------------------------------------------------------------
67
68  plt.hist(dataset,bins=50)
69  plt.xlabel("$x$",fontsize=22)
70  plt.ylabel("$f(x)$",fontsize=22,rotation="horizontal",labelpad=20)
71  plt.legend(fontsize=10,shadow=True,title="$h$",title_fontsize=16)
72  plt.savefig("./Plots/kde.jpg")
```

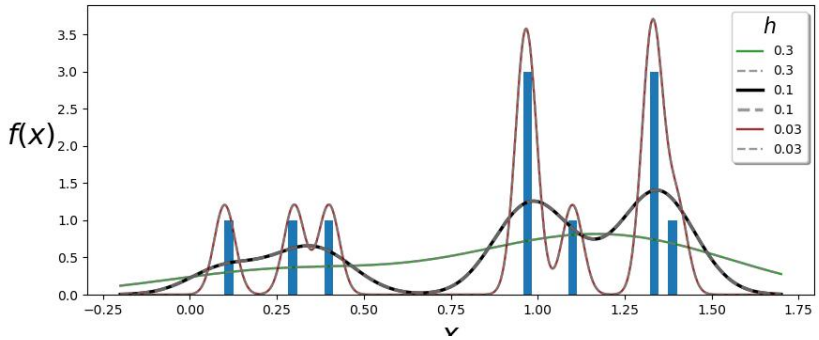

Figure 31: Showcase of Kernel Density Estimation method: kde.jpg

10 Results

10.1 Moments

The initial objective was to examine how the radius distributions behave across jet p_T bins for different data sets, using moment analysis. The procedure is as follows: A jet p_T bin is selected, and then the moment is calculated for each jet individually. A particle's contribution to the jet's moment is defined as its radius raised to the relevant power and multiplied by its contribution to the probability. As described in Section 8, this contribution is given by the transverse momentum of the particle divided

by the total transverse momentum of all particles in the same jet, further normalized by the number of jets considered, which in this case is one. The n-th moment of a jet is then calculated as follows.

$$M_J^{(n)} = \sum_{i(particles)} R_{i(J)}^n \cdot \frac{p_{T\,i(J)}}{\sum_{k(particles)} p_{T\,k(J)}} \tag{59}$$

Once the moment of each jet within a given jet p_T bin is calculated, the mean and standard error can be easily calculated. These summary statistics were then used to generate the plots shown below. **It is important to note that these are not the moments of the radius probability distributions corresponding to the jet p_T bins, but rather the average values of the jet-wise moments for all jets falling within a given p_T bin.**

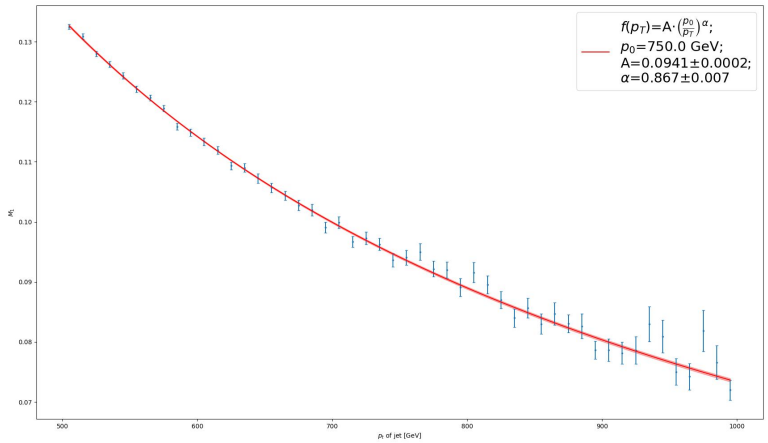

Figure 32: First moment behavior in `WToQQ_070.root` ($\sim$ 100000 jets considered): moments_root.jpg

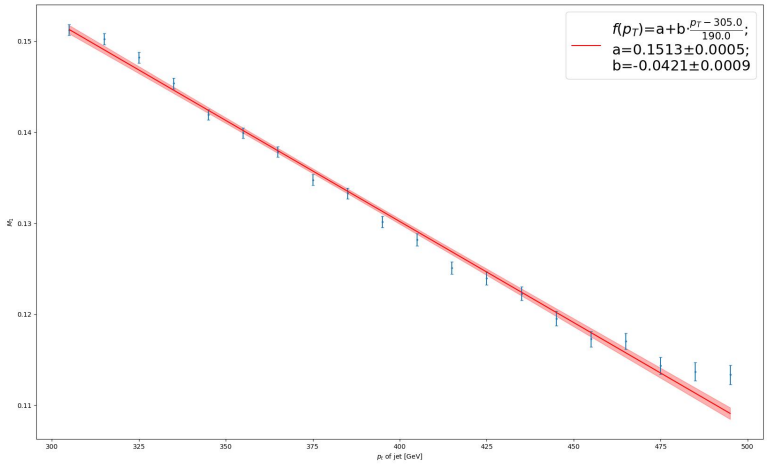

Figure 33: First moment behavior in `RunG_batch0.h5` ($\sim$ 400000 jets considered): moments_lowpt.jpg

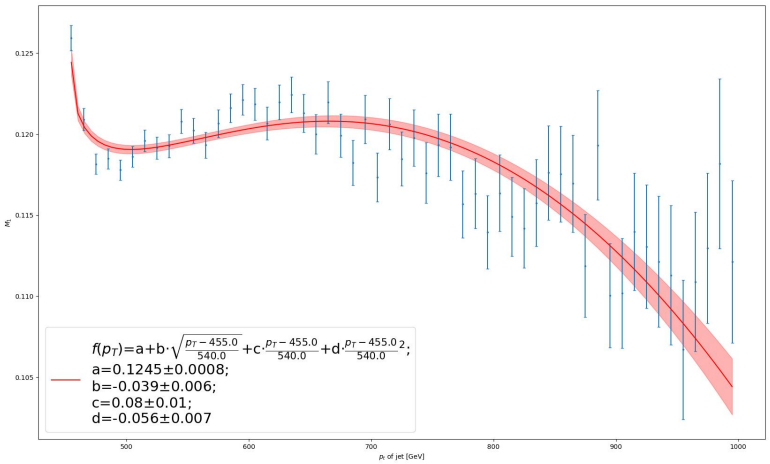

Figure 34: First moment behavior in `QCDjj_pT_450_1200_train01.h5` ($\sim$ 400000 jets considered): moments_highpt.jpg

10.2 Radius probability distribution

The next step was to look at the radius probability distribution directly. For each data set, a single jet p_T bin was selected to illustrate the shape of the distribution.

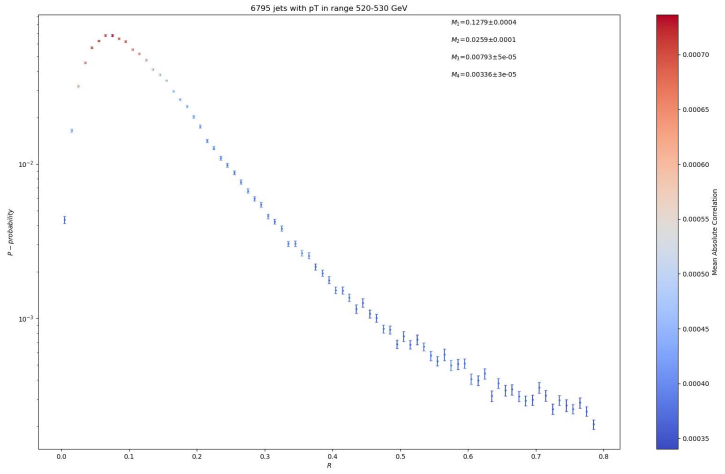

Figure 35: A jet p_T bin from `WToQQ_070.root`: probability_root.jpg

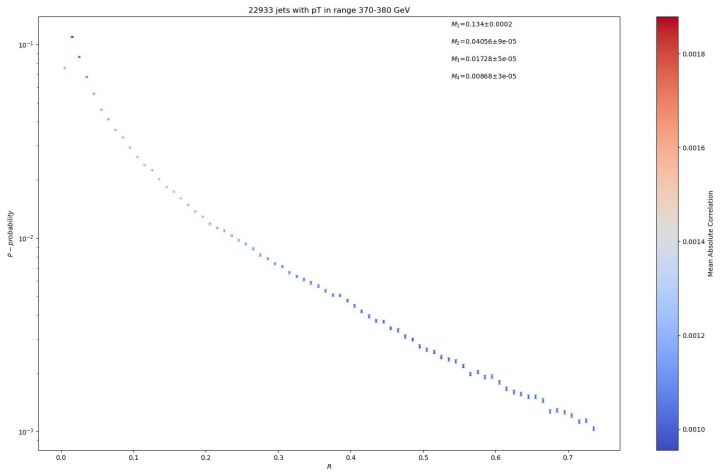

Figure 36: A jet p_T bin from `RunG_batch0.h5`: probability_lowpt.jpg

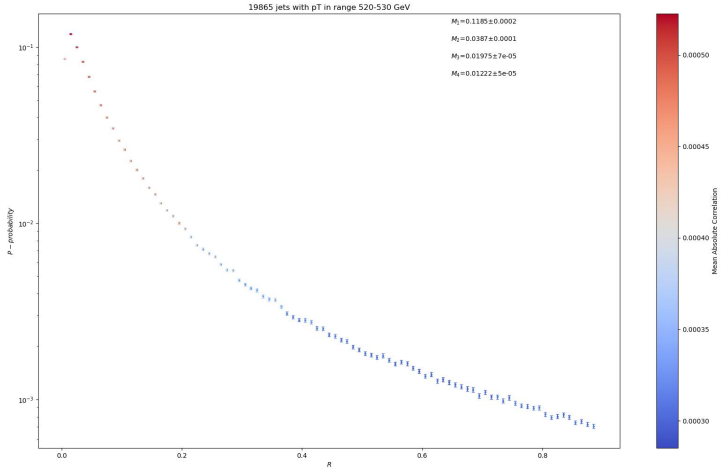

Figure 37: A jet p_T bin from `QCDjj_pT_450_1200_train01.h5`: probability_highpt.jpg

The moments of the given histograms are calculated as in the expression (60) below. Moment error is calculated with the help of the error propagation formula given in the expression (63) which, for this specific case, results in the expression (64). **It is important to note that these are the moments of the radius probability distribution corresponding to the jet p_T bin, not the average values of the jet-wise moments for all jets falling within the given p_T bin.**

$$\overline{M_n} = \sum_{k(bins)} (\text{k-th bin middle})^n \cdot \overline{\text{P}}(\text{k-th bin}) \tag{60}$$

$$x = F(a, b, c, ...) \tag{61}$$

$$\overline{x} = F(\overline{a}, \overline{b}, \overline{c}, ...) \tag{62}$$

$$x^2_{stdev} = \sum_{k=a,b,c,...} \left(\left. \frac{\partial F}{\partial k} \right|_{\substack{a=\overline{a} \\ b=\overline{b} \\ c=\overline{c}}} \cdot k_{stdev} \right)^2 \tag{63}$$

$$M_n^{(stdev)} = \sqrt{\sum_{k(bins)} (\text{k-th bin middle})^{2n} \cdot \text{P}^{(stdev)}(\text{k-th bin})^2} \tag{64}$$

$$M_n = \overline{M_n} \pm M_n^{(stdev)} \tag{65}$$

10.3 Radius probability density function

After many attempts to fit the radius probability distribution for each chosen jet p_T bin with a single functional form, it was realized that it would be easier to fit the radius probability density since the troubling peak at low radius then disappears. Once the peak was gone, the exponential polynomial function was utilized to obtain the final model. The chosen functional form for the parametric fit was a linear combination of Legendre polynomials inside an exponential function.

$$f(x) = C \cdot exp(a_1 L_1(x) + a_2 L_2(x) + a_4 L_4(x) + a_5 L_5(x) + a_6 L_6(x) + a_7 L_7(x) + a_8 L_8(x)), \tag{66}$$

$$x = 2R - 1$$

The term involving $L_3(x)$ was not included, as this specific model configuration had initially been selected during an earlier phase when parts of the procedure were not yet fully correct. Despite that, the model has since proven effective even under the corrected conditions and was therefore retained in this simplified form. Importantly, the implementation remains flexible, allowing for straightforward testing of alternative configurations if needed.
The modeling was done in three steps described below.

Fitting Procedure

- **Step 1: Initial independent fits**
  - Fit the radius probability density function independently for each jet p_T bin, using only the variances of the radius bins.
  - Each fit yields a set of top-level parameters, specific to that jet p_T bin.
  - Then, for each top-level parameter, fit a smooth function to model its dependence on jet p_T. The parameters of these functions are referred to as bottom-level parameters, and they remain uncorrelated and are fit separately for each top-level parameter.
- **Step 2: Refined independent fits with full covariance**

- Repeat the fits for each jet p_T bin, now using the full covariance matrix of the radius bins.
- The top-level parameters are determined again independently for each bin.
- The initial guesses for these parameters are obtained by evaluating the smooth trends from Step 1, that is, using bottom-level parameters from the previous step.
- As in Step 1, the bottom-level parameters are not jointly optimized; they serve only to guide the initial guesses.

- **Step 3: Global fit with shared bottom-level parameters**
  - Perform a global fit of all jet p_T bins simultaneously by combining their covariance matrices into a single block-diagonal matrix.
  - In this step, the top-level parameters for each p_T bin are no longer treated as independent values; instead, they are constrained to follow a common functional form governed by shared bottom-level parameters.
  - These bottom-level parameters, which define how the top-level parameters vary with jet p_T, are now jointly optimized across all jet p_T bins and may become correlated through global fit.
  - The initial guess for the global fit is constructed by concatenating the bottom-level parameters obtained in Step 2.

The following plots present the fits of the top-level parameters from Step 2 for each data set.

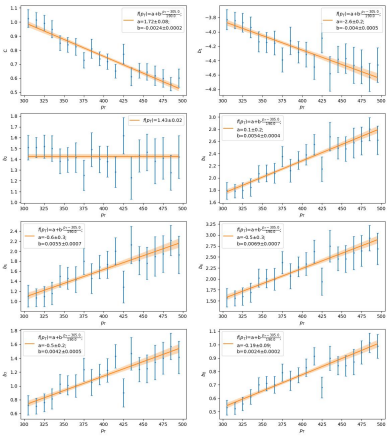

Figure 38: Parameter fits for `RunG_batch0.h5`: param_fits_lowpt.jpg

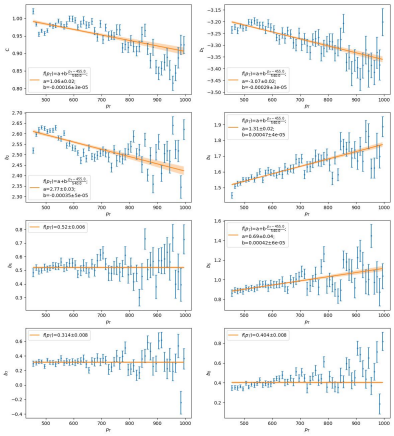

Figure 39: Parameter fits for `QCDjj_pT_450_1200_train01.h5`: param_fits_highpt.jpg

The bottom-level parameters presented in the legends of each data set are then used for the initial guess of the global fit.

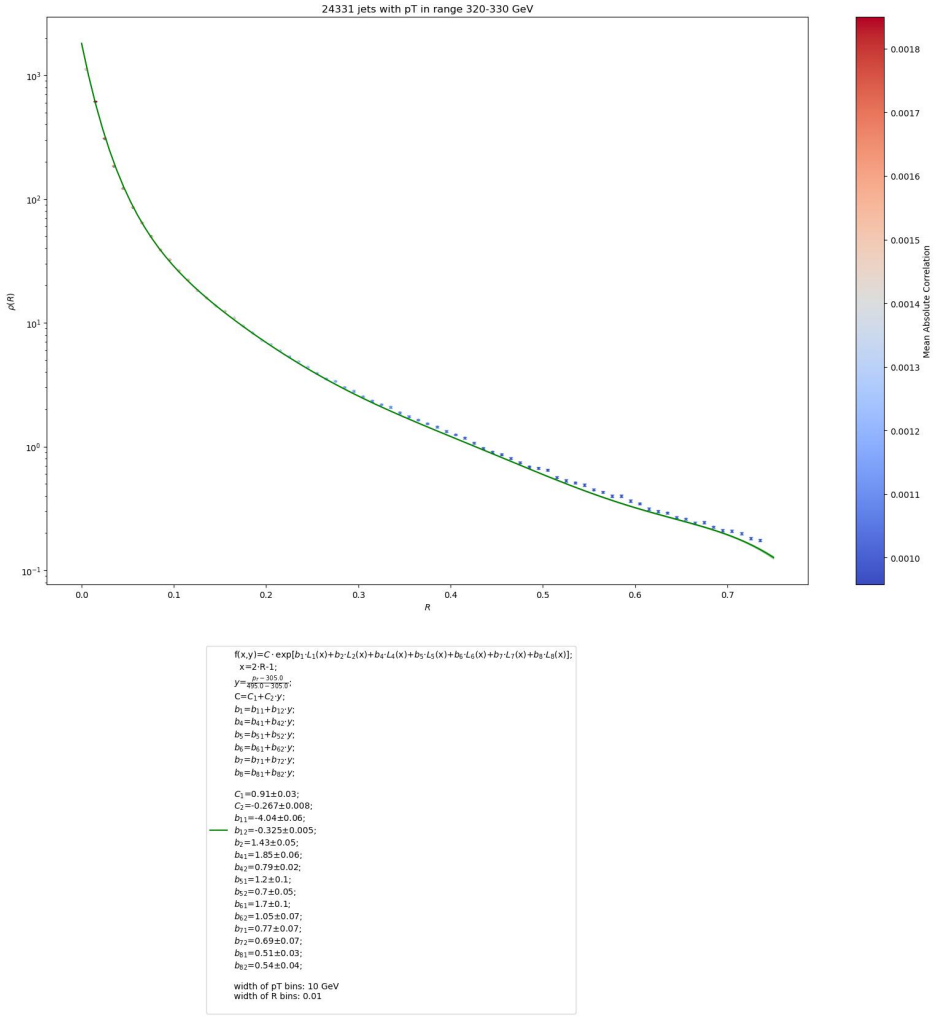

Figure 40: The final fit for `RunG_batch0.h5`: FINAL_FIT_lowpt.jpg

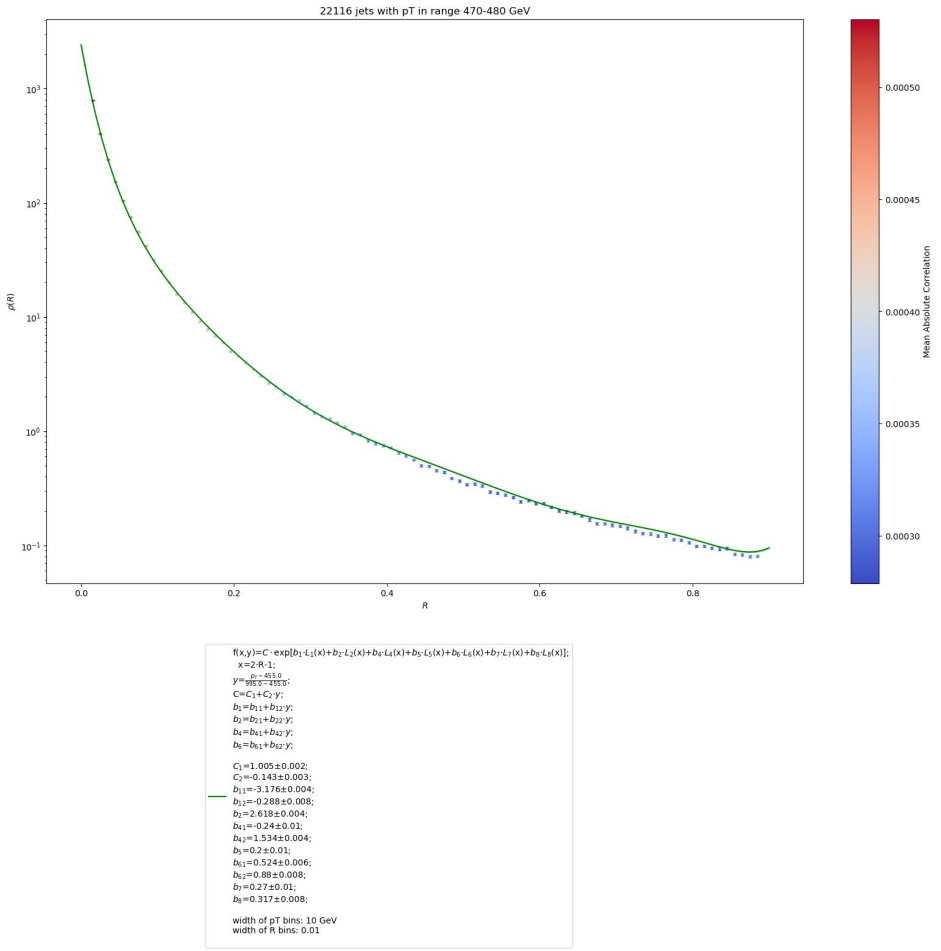

Figure 41: The final fit for `QCDjj_pT_450_1200_train01.h5`: FINAL_FIT_highpt.jpg

The only issue arises in the higher jet p_T range, where the model fails to capture the ideal tail behavior - instead of tapering off, the tail does not decline as expected.

10.4 Bayesian machine scientist

This last subsection allows for exploring alternative models. That said, as mentioned earlier, machine scientist may yield a function that is difficult to work with. In addition, because it does not incorporate data uncertainties, the resulting function may not ensure stable performance in the subsequent error-weighted fitting. As before, the data is structured so that the width of the jet p_T bins is 10GeV, while the width of the radius bins is 0.01.

`RunG_batch0.h5` file - probability distribution

Jet p_T range 300-500GeV, $\approx$ 400000 jets loaded, $R_{max} = 0.75$

Machine scientist 2000 steps:

((((((sin((((R + _a0_) / _a9_) / ((R + _a0_) ** _a6_))) + (_a0_ * _a5_)) ** pt_middles) * (R ** _a11_)) ** (pt_middles ** _a2_)) + _a7_) / _a10_)

{"_a0_": -0.002757271313716955,

”_a2_”: -0.718451171271083,

”_a5_”: 85.60360674050297,

”_a6_”: 1.231599559670358,

”_a7_”: -5.551102369878217e-05,

”_a9_”: 2.273954508004391,

”_a10_”: 0.15121348929160183,

”_a11_”: 37.49953585484166,

”_a1_”: 1.0,

”_a3_”: 1.0,

”_a4_”: 1.0,

”_a8_”: 1.0,

”_a12_”: 1.0}

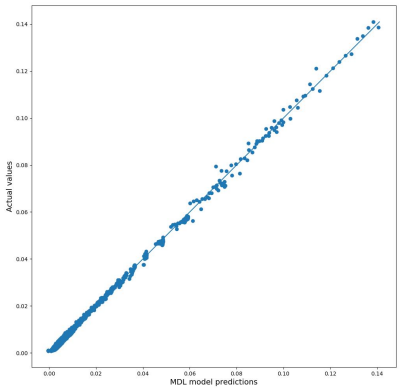

Figure 42: Quality of the probability model for `RunG_batch0.h5` after 2000 steps: ms_prob_lowpt_2000.jpg

`RunG_batch0.h5` file - probability density function

Jet p_T range 300-500GeV, $\approx$ 400000 jets loaded, $R_{max} = 0.75$

Machine scientist 2000 steps:

((((((exp((R * (_a10_ ** pt_middles))) + _a7_) ** _a1_) * (((pt_middles * _a6_) + _a9_) / (_a1_ + pt_middles))) * R) + _a2_) ** pt_middles)

{”_a1_”: -1.180835600652233,

”_a2_”: 0.987189266832617,

”_a6_”: 0.0121791494053336,

”_a7_”: -0.9988267443494638,

”_a9_”: 2.259618173395361,

”_a10_”: 1.000357042489772,

”_a0_”: 1.0,

"_a3_": 1.0,

"_a4_": 1.0,

"_a5_": 1.0,

"_a8_": 1.0,

"_a11_": 1.0,

"_a12_": 1.0}

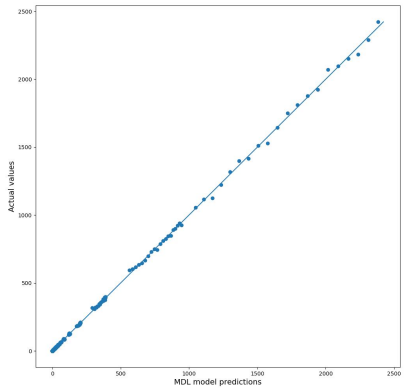

Figure 43: Quality of the probability density model for `RunG_batch0.h5` after 2000 steps: ms_dens_lowpt_2000.jpg

`QCDjj_pT_450_1200_train01.h5` file - probability distribution

Jet p_T range 450-1000GeV, $\approx$ 400000 jets loaded, $R_{max} = 0.9$

Machine scientist 3000 steps:

((((((pt_middles + _a8_) * ((pt_middles ** _a2_) * (R ** _a12_))) + ((R / _a7_) + ((R ** _a3_) * (_a5_ ** _a9_)))) ** (_a9_ * _a2_)) / ((_a3_ + _a1_) + pt_middles)) * _a5_)

{"_a1_": 2237.0787838570122,

"_a2_": -2.42575003999361,

"_a3_": 0.19927590151212884,

"_a5_": 0.004443476340136262,

"_a7_": 16.219474949990126,

"_a8_": -258.73252705457355,

"_a9_": 0.8992038653292346,

"_a12_": -0.7672415482817039,

"_a0_": 1.0,

"_a4_": 1.0,

"_a6_": 1.0,

"_a10_": 1.0,

"_a11_": 1.0}

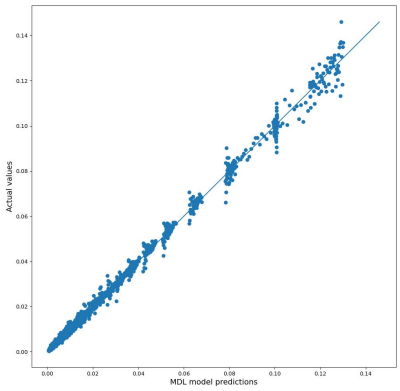

Figure 44: Quality of the probability model for `QCDjj_pT_450_1200_train01.h5` after 3000 steps: ms_prob_highpt_3000.jpg

`QCDjj_pT_450_1200_train01.h5` file - probability density function

Jet p_T range 450-1000GeV, $\approx$ 400000 jets loaded, $R_{max} = 0.9$

Machine scientist 3000 steps:

(((((((((_a4_ * _a4_) * _a7_) ** (pt_middles * _a12_)) + _a0_) + _a0_) ** (_a6_ * R)) + _a2_) / (R / (_a11_ + ((_a4_ ** pt_middles) / R)))) * _a12_)

{"_a0_": 4.517402226061548,
"_a2_": -0.8545345597067965,
"_a4_": 0.999760720675125,
"_a6_": -20.7029615044342,
"_a7_": 0.9933407738924719,
"_a11_": -1.7808752481862429,
"_a12_": -0.4823028433387591,
"_a1_": 1.0,
"_a3_": 1.0,
"_a5_": 1.0,
"_a8_": 1.0,
"_a9_": 1.0,
"_a10_": 1.0}

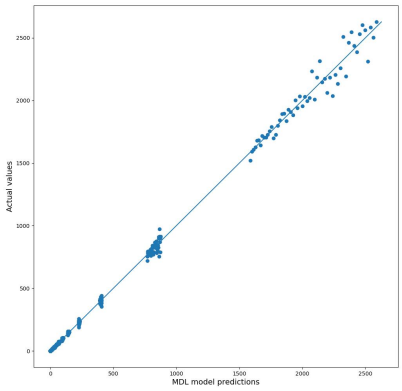

Figure 45: Quality of the probability density model for `QCDjj_pT_450_1200_train01.h5` after 3000 steps: ms_dens_highpt_3000.jpg

From the figures above it can be concluded, as expected, that the probability density function is easier to model than the probability distribution.

Bibliography

[1] Qu, H., Li, C., & Qian, S. (2022). JetClass: A Large-Scale Dataset for Deep Learning in Jet Physics (1.0.0) [Data set]. Zenodo. https://doi.org/10.5281/zenodo.6619768

[2] https://www.lhc-closer.es/taking_a_closer_look_at_lhc/0.momentum

[3] Tomažič, T. Analiza generiranih podatkov, 2024.

[4] K. Zoch, J. A. Raine, D. Sengupta, and T. Golling. RODEM Jet Datasets. Available on Zenodo: 10.5281/zenodo.12793616. Aug. 2024. arXiv: 2408.11616 [hep-ph]

[5] Amram, Oz, Anzalone, Luca, Birk, Joschka, Faroughy, Darius A., Hallin, Anna, Kasieczka, Gregor, Shih, David. (2024). Aspen Open Jets: a real-world ML-ready dataset for jet physics (Version 1.0) [Data set]. http://doi.org/10.25592/uhhfdm.16505

[6] R. Guimer, I. Reichardt, A. Aguilar-Mogas, F. A. Massucci, M. Miranda, J. Pallars, M. Sales-Pardo, A Bayesian machine scientist to aid in the solution of challenging scientific problems. Sci. Adv. 6, eaav6971 (2020). https://seeslab.info/publications/bayesian-machine-scientist-aid-solution-challenging-scientific-problems/