**Note that for this analysis Squared Axes Ratio is mentioned as arxy (aspect ratio).**

In [1]:
```
import pandas as pd

pd.set_option('display.max_columns', None)

pd.set_option('display.max_colwidth', None)

pd.set_option('display.max_rows', None)


ip08 = pd.read_csv('/kaggle/input/nfl-big-data-bowl-2026-analytics/114239_nfl_competition_files_published_analytics_final/train/input_2023_w08.csv')

op08 = pd.read_csv('/kaggle/input/nfl-big-data-bowl-2026-analytics/114239_nfl_competition_files_published_analytics_final/train/output_2023_w08.csv')

sup_data = pd.read_csv('/kaggle/input/nfl-big-data-bowl-2026-analytics/114239_nfl_competition_files_published_analytics_final/supplementary_data.csv')
```

```
/tmp/ipykernel_39/2835549976.py:14: DtypeWarning: Columns (25) have
mixed types. Specify dtype option on import or set low_memory=Fals
e.
  sup_data = pd.read_csv('/kaggle/input/nfl-big-data-bowl-2026-anal
ytics/114239_nfl_competition_files_published_analytics_final/supple
mentary_data.csv')
```

In [2]:

```
import pandas as pd
import pywt
import numpy as np
import json

from tqdm import tqdm

# Load the datasets
# games = pd.read_csv('/kaggle/input/nfl-big-data-bowl-2025/games.csv')
# plays = pd.read_csv('/kaggle/input/nfl-big-data-bowl-2025/plays.csv')
# players = pd.read_csv('/kaggle/input/nfl-big-data-bowl-2025/players.csv')
# player_play = pd.read_csv('/kaggle/input/nfl-big-data-bowl-2025/player_play.csv')

# sup_data = pd.read_csv('/kaggle/input/nfl-big-data-bowl-2026-analytics/114239_nfl_competition_files_published_analytics_final/supplementary_data.csv')

tracking_data_dict_play = {}

# for week in tqdm(range(6, 8)):
#     tracking_week_i = pd.read_csv(f'/kaggle/input/nfl-big-data-bowl-2025/tracking_week_{week}.csv')

print("input files")
tracking_data_ip = pd.concat([
#     pd.read_csv(f'/kaggle/input/nfl-big-data-bowl-2025/tracking_week_{week}.csv') for week in tqdm(range(6, 8))
    # pd.read_csv(f'/kaggle/input/nfl-big-data-bowl-2025/tracking_week_{week}.csv') for week in tqdm(range(1, 10))
    pd.read_csv(f'/kaggle/input/nfl-big-data-bowl-2026-analytics/114239_nfl_competition_files_published_analytics_final/train/input_2023_w{week:02d}.csv') for week in tqdm(range(1, 19))
])


print("output files")
tracking_data_op = pd.concat([
#     pd.read_csv(f'/kaggle/input/nfl-big-data-bowl-2025/tracking_week_{week}.csv') for week in tqdm(range(6, 8))
    # pd.read_csv(f'/kaggle/input/nfl-big-data-bowl-2025/tracking_week_{week}.csv') for week in tqdm(range(1, 10))
```

```
    pd.read_csv(f'/kaggle/input/nfl-big-data-bowl-2026-analytics/114239_nfl_competition_files_published_analytics_final/train/output_2023_w{week:02d}.csv') for week in tqdm(range(1, 19))
])


print("supplementary files")
sup_data = pd.read_csv('/kaggle/input/nfl-big-data-bowl-2026-analytics/114239_nfl_competition_files_published_analytics_final/supplementary_data.csv')
sup_data["play_action"] = sup_data["play_action"].astype("float64")



# # dtype_dict = {'passResult':'str', 'foulName1':'str', 'foulName2': 'str'}

# # for (gameId, playId), group_data in tracking_data.groupby(['gameId', 'playId']):
# for (gameId, playId), group_data in tqdm(tracking_data.groupby(['gameId', 'playId']), total=tracking_data.groupby(['gameId', 'playId']).ngroups):

# #     print('>',end="")

#     if (gameId, playId) not in tracking_data_dict_play:
#         tracking_data_dict_play[(gameId, playId)] = {
#             'tracking_data': group_data,
#             'meta_data': {
#                 'gameId': gameId, 'playId': playId

#             }
#         }
#     else:
#         print('x',end="")

#         tracking_data_dict_play[(gameId, playId)]['tracking_data'] = group_data
#         tracking_data_dict_play[(gameId, playId)]['meta_data'] = {'gameId': gameId, 'playId': playId
#             }
```

```
input files

100%|██████████| 18/18 [00:19<00:00,  1.10s/it]

output files

100%|██████████| 18/18 [00:00<00:00, 32.63it/s]

supplementary files

/tmp/ipykernel_39/851042370.py:38: DtypeWarning: Columns (25) have
mixed types. Specify dtype option on import or set low_memory=Fals
e.
  sup_data = pd.read_csv('/kaggle/input/nfl-big-data-bowl-2026-anal
ytics/114239_nfl_competition_files_published_analytics_final/supple
mentary_data.csv')
```

In [3]:
```
game_id = 2023091709 #2023102908      #2023122406
play_id = 1228 #596 # 1653
nfl_id = 44959

sup_data[(sup_data['game_id']==game_id) & (sup_data['play_id']==play_id)]
```

```
/usr/local/lib/python3.11/dist-packages/pandas/io/formats/format.py:1458: RuntimeWarning: invalid value encountered in greater
  has_large_values = (abs_vals > 1e6).any()
/usr/local/lib/python3.11/dist-packages/pandas/io/formats/format.py:1459: RuntimeWarning: invalid value encountered in less
  has_small_values = ((abs_vals < 10 ** (-self.digits)) & (abs_vals > 0)).any()
/usr/local/lib/python3.11/dist-packages/pandas/io/formats/format.py:1459: RuntimeWarning: invalid value encountered in greater
  has_small_values = ((abs_vals < 10 ** (-self.digits)) & (abs_vals > 0)).any()
```

Out[3]:

| | game_id | season | week | game_date | game_time_eastern | home_team_abbr | visitor_tea |
|---|---|---|---|---|---|---|---|
| 1385 | 2023091709 | 2023 | 2 | 09/17/2023 | 16:05:00 | LA | SF |

Adding player side

In [4]:
```
ip_side = tracking_data_ip[['game_id','play_id','nfl_id','player_sid
e']].drop_duplicates()



tracking_data_op = tracking_data_op.merge(
    ip_side,
    on=['game_id','play_id','nfl_id'],
    how='left'
)
```

In [5]:
```
tracking_data_op['player_side'].isna().sum()
```

Out[5]:
```
0
```

In [6]:
```
# tracking_data_op.head()
tracking_data_op['player_side'].value_counts()
```

Out[6]:
```
player_side
Defense    402576
Offense    160360
Name: count, dtype: int64
```

In [7]:
```
tracking_data_op.columns
```

Out[7]:
```
Index(['game_id', 'play_id', 'nfl_id', 'frame_id', 'x', 'y', 'playe
r_side'], dtype='object')
```

In [8]:
```
tracking_data_op.shape
tracking_data_op.head()
```

Out[8]:

| | game_id | play_id | nfl_id | frame_id | x | y | player_side |
|---|---|---|---|---|---|---|---|
| 0 | 2023090700 | 101 | 46137 | 1 | 56.22 | 17.28 | Defense |
| 1 | 2023090700 | 101 | 46137 | 2 | 56.63 | 16.88 | Defense |
| 2 | 2023090700 | 101 | 46137 | 3 | 57.06 | 16.46 | Defense |
| 3 | 2023090700 | 101 | 46137 | 4 | 57.48 | 16.02 | Defense |
| 4 | 2023090700 | 101 | 46137 | 5 | 57.91 | 15.56 | Defense |

In [ ]:
```
```

In [9]:
```
tracking_data_ip['frame_id'].max() #123
tracking_data_op['frame_id'].max() #94
len(tracking_data_ip['frame_id']) #4880579
len(tracking_data_op['frame_id']) #562936
len(tracking_data_ip)#4880579
len(tracking_data_op) #562936
```

Out[9]:
```
562936
```

In [10]:

```
# dtype_dict = {'passResult':'str', 'foulName1':'str', 'foulName2': 'st
r'}

ip_gb  = tracking_data_ip.groupby(["game_id","play_id"], sort=False)
# # list(ip_gb.groups.keys())[:10]   # first 10 keys
# ip_gb = list(tracking_data_ip.groupby(["game_id","play_id"], sort=Fals
e).groups.keys())

op_gb  = tracking_data_op.groupby(["game_id","play_id"], sort=False)
sup_gb = sup_data.groupby(["game_id","play_id"], sort=False)


tracking_data_dict_play = {}

from collections import defaultdict
from tqdm import tqdm



tracking_data_dict_play = defaultdict(lambda: {
    "input": None,
    "output": None,
    "supplementary": None,
    "meta_data": None,
})


# for key, grp in tqdm(ip_gb): #ip_grp:
#     if key not in tracking_data_dict_play:
#         tracking_data_dict_play[key]["input"] = grp
#         tracking_data_dict_play[key]["meta_data"] = {"gameId": key[0],
"playId": key[1]}
#     else: print('x ip')

# INPUT

for key, grp in tqdm(ip_gb):
    d = tracking_data_dict_play[key]   # auto-created if missing
    d["input"] = grp
    d["meta_data"] = {"gameId": key[0], "playId": key[1]}

# OUTPUT
```

```
for (game_id, play_id), grp in op_gb:
    d = tracking_data_dict_play[(game_id, play_id)]
    d["output"] = grp
    if d["meta_data"] is None:
        d["meta_data"] = {"gameId": game_id, "playId": play_id}

# SUPPLEMENTARY

for (game_id, play_id), grp in sup_gb:
    d = tracking_data_dict_play[(game_id, play_id)]
    d["supplementary"] = grp
    if d["meta_data"] is None:
        d["meta_data"] = {"gameId": game_id, "playId": play_id}



# for key, grp in op_gb:
#     if key not in tracking_data_dict_play:
#         tracking_data_dict_play[key]["output"] = grp
#         if tracking_data_dict_play[key]["meta_data"] is None:
#             tracking_data_dict_play[key]["meta_data"] = {"gameId": key
[0], "playId": key[1]}
#     else: print('x op')

# for key, grp in sup_gb:
#     if key not in tracking_data_dict_play:
#         tracking_data_dict_play[key]["supplementary"] = grp  # ALWAYS
a DataFrame
#         if tracking_data_dict_play[key]["meta_data"] is None:
#             tracking_data_dict_play[key]["meta_data"] = {"gameId": key
[0], "playId": key[1]}
#     else: print('x sup')



# for key, grp in ip_grp:
#     tracking_data_dict_play[key]["input"] = grp
#     tracking_data_dict_play[key]["meta_data"] = {"gameId": key[0], "pl
ayId": key[1]}

# for key, grp in op_gb:
#     tracking_data_dict_play[key]["output"] = grp
#     if tracking_data_dict_play[key]["meta_data"] is None:
#         tracking_data_dict_play[key]["meta_data"] = {"gameId": key[0],
```

```
"playId": key[1]}

# for key, grp in sup_gb:
#     tracking_data_dict_play[key]["supplementary"] = grp  # ALWAYS a Da
taFrame
#     if tracking_data_dict_play[key]["meta_data"] is None:
#         tracking_data_dict_play[key]["meta_data"] = {"gameId": key[0],
"playId": key[1]}



# # for (gameId, playId), group_data in tracking_data.groupby(['gameId',
'playId']):
# for (gameId, playId), group_data_ip in tqdm(tracking_data_ip.groupby
(['gameId', 'playId']), total=tracking_data_ip.groupby(['gameId', 'playI
d']).ngroups):

# #     print('>',end="")

#     if (gameId, playId) not in tracking_data_dict_play:
#         tracking_data_dict_play[(gameId, playId)] = {
#             'tracking_data': group_data_ip,
#             'meta_data': {
#                 'gameId': gameId, 'playId': playId

#             }
#         }
#     else:
#         print('x',end="")

#         tracking_data_dict_play[(gameId, playId)]['tracking_data'] = g
roup_data
#         tracking_data_dict_play[(gameId, playId)]['meta_data'] = {'gam
eId': gameId, 'playId': playId
#             }
```

```
100%|██████████| 14108/14108 [00:01<00:00, 7090.07it/s]
```

In [ ]:

In [32]:
```
ip_keys  = set(ip_gb.groups.keys())
op_keys  = set(op_gb.groups.keys())
sup_keys = set(sup_gb.groups.keys())


# Are input and output keys identical?
print("ip == op:", ip_keys == op_keys)

# Are input and supplementary keys identical?
print("ip == sup:", ip_keys == sup_keys)

# Any common keys at all?
print("ip ∩ op:", len(ip_keys & op_keys))
print("ip ∩ sup:", len(ip_keys & sup_keys))
print("op ∩ sup:", len(op_keys & sup_keys))

print(len(ip_keys), len(op_keys), len(sup_keys))
```

```
ip == op: True
ip == sup: False
ip ∩ op: 14108
ip ∩ sup: 14108
op ∩ sup: 14108
14108 14108 18009
```

In [33]:
```
# KEYS ORDER

ip_list = list(ip_gb.groups.keys())
op_list = list(op_gb.groups.keys())

ip_list == op_list
```

Out[33]:
```
True
```

In [ ]:
```

```

In [34]:

```
ip_keys  = set(ip_gb.groups.keys())
sup_keys = set(sup_gb.groups.keys())

extra_sup_keys = sup_keys - ip_keys
print("Rows in sup but not in ip:", len(extra_sup_keys))


mask = sup_data.apply(lambda row: (row["game_id"], row["play_id"]) in e
xtra_sup_keys, axis=1)
sup_extra_rows = sup_data[mask]

sup_extra_rows.head()
```

```
Rows in sup but not in ip: 3901


/usr/local/lib/python3.11/dist-packages/pandas/io/formats/format.p
y:1458: RuntimeWarning: invalid value encountered in greater
  has_large_values = (abs_vals > 1e6).any()
/usr/local/lib/python3.11/dist-packages/pandas/io/formats/format.p
y:1459: RuntimeWarning: invalid value encountered in less
  has_small_values = ((abs_vals < 10 ** (-self.digits)) & (abs_vals
> 0)).any()
/usr/local/lib/python3.11/dist-packages/pandas/io/formats/format.p
y:1459: RuntimeWarning: invalid value encountered in greater
  has_small_values = ((abs_vals < 10 ** (-self.digits)) & (abs_vals
> 0)).any()
```

Out[34]:

| | game_id | season | week | game_date | game_time_eastern | home_team_abbr | visitor_te |
|---|---|---|---|---|---|---|---|
| 14108 | 2024120500 | 2024 | 14 | 12/05/2024 | 20:15:00 | DET | GB |
| 14109 | 2024120500 | 2024 | 14 | 12/05/2024 | 20:15:00 | DET | GB |
| 14110 | 2024120500 | 2024 | 14 | 12/05/2024 | 20:15:00 | DET | GB |
| 14111 | 2024120500 | 2024 | 14 | 12/05/2024 | 20:15:00 | DET | GB |
| 14112 | 2024120500 | 2024 | 14 | 12/05/2024 | 20:15:00 | DET | GB |

In [ ]:

In [ ]:

In [35]:
```
ip_gb3  = tracking_data_ip.groupby(["game_id","play_id", 'nfl_id'], sor
t=False)
# # list(ip_gb.groups.keys())[:10]   # first 10 keys
# ip_gb = list(tracking_data_ip.groupby(["game_id","play_id"], sort=Fals
e).groups.keys())

op_gb3  = tracking_data_op.groupby(["game_id","play_id", "nfl_id"], sor
t=False)
# sup_gb3 = sup_data.groupby(["game_id","play_id", 'nfl_id'], sort=Fals
e)
```

In [36]:
```
ip_keys3  = set(ip_gb3.groups.keys())
op_keys3  = set(op_gb3.groups.keys())
# sup_keys = set(sup_gb.groups.keys())


# Are input and output keys identical?
print("ip == op:", ip_keys3 == op_keys3)

# Are input and supplementary keys identical?
# print("ip == sup:", ip_keys == sup_keys)

# Any common keys at all?
print("ip ∩ op:", len(ip_keys3 & op_keys3))
# print("ip ∩ sup:", len(ip_keys & sup_keys))
# print("op ∩ sup:", len(op_keys & sup_keys))

print(len(ip_keys3), len(op_keys3))#, len(sup_keys))
```

```
ip == op: False
ip ∩ op: 4
173150 4
```

In [37]:
```
op_keys3.issubset(ip_keys3)
```

Out[37]:
```
True
```

In [ ]:
```
```

In [38]:
```
# sup_extra_rows = sup_data.merge(
#     tracking_data_ip[["game_id","play_id"]].drop_duplicates(),
#     on=["game_id","play_id"],
#     how="left",
#     indicator=True
# ).query('_merge == "left_only"').drop(columns="_merge")

# sup_extra_rows.head()
```

In [ ]:
```
```

In [39]:
```
game_id = 2023091709 #2023102908      #2023122406
play_id = 1228 #596 # 1653
nfl_id = 44959 #52433 #47834    #54475
```

In [40]:
```
tracking_data_ip[(tracking_data_ip['game_id'] == game_id) & (tracking_d
ata_ip['play_id'] == play_id)]
tracking_data_ip[(tracking_data_ip['game_id'] == game_id) & (tracking_d
ata_ip['play_id'] == play_id)]['nfl_id'].unique()
```

Out[40]:
```
array([44878, 55942, 52844, 54677, 53532, 52607, 52835, 54718, 4482
0,
       47819, 40078, 56118, 54727, 44959])
```

In [41]:
```
# tracking_data_ip[(tracking_data_ip['game_id'] == game_id) & (tracking_
data_ip['play_id'] == play_id)]
tracking_data_op[(tracking_data_op['game_id'] == game_id) & (tracking_d
ata_op['play_id'] == play_id)]
# sup_data[(sup_data['game_id'] == game_id) & (sup_data['play_id'] == pl
ay_id)] # and sup_data['game_id'] == n_id
tracking_data_op[(tracking_data_op['game_id'] == game_id) & (tracking_d
ata_op['play_id'] == play_id)]['nfl_id'].unique()
```

Out[41]:
```
array([], dtype=int64)
```

In [42]:
```
tracking_data_ip.shape
# tracking_data_op.shape
```

Out[42]:
```
(4880579, 23)
```

In [43]:
```
print("output files")
tracking_data_op = pd.concat([
#     pd.read_csv(f'/kaggle/input/nfl-big-data-bowl-2025/tracking_week_
{week}.csv') for week in tqdm(range(6, 8))
    # pd.read_csv(f'/kaggle/input/nfl-big-data-bowl-2025/tracking_week_
{week}.csv') for week in tqdm(range(1, 10))
    pd.read_csv(f'/kaggle/input/nfl-big-data-bowl-2026-analytics/114239
_nfl_competition_files_published_analytics_final/train/output_2023_w{we
ek:02d}.csv') for week in tqdm(range(1, 19))
])

tracking_data_op.shape
```

```
output files

100%|██████████| 18/18 [00:00<00:00, 52.35it/s]
```

Out[43]:
```
(562936, 6)
```

In [ ]:

In [ ]:

In [44]:
```
# tracking_data_ip['nfl_id'].value_counts(dropna=False)
# ball_rows = tracking_data_ip[tracking_data_ip['nfl_id'].isna()]
# ball_rows.head()

# ball_rows = tracking_data_op[tracking_data_ip['nfl_id'].isna()]
# ball_rows.head()

# ball_rows = sup_data[tracking_data_ip['nfl_id'].isna()]
# ball_rows.head()
```

In [ ]:

In [ ]:

## Functions

In [45]:

```
import numpy as np

# def calculate_eccentricity(A, B, C):
def calculate_eccentricity(A, B, C, s, acc, dis, o, dir_, theta_) :
    # Step 1: Calculate eigenvalues for semi-major and semi-minor axis l
engths
    eig_val1 = 0.5 * (A + C + np.sqrt((A - C)**2 + B**2))
    eig_val2 = 0.5 * (A + C - np.sqrt((A - C)**2 + B**2))
    a = np.sqrt(1 / np.abs(eig_val1))  # Major axis length
    b = np.sqrt(1 / np.abs(eig_val2))  # Minor axis length
#     a, b = sorted([a, b], reverse=True)  # Ensure a >= b

    # Step 2: Calculate the orientation angle of the principal axes
#     theta = 0.5 * np.arctan2(B, A - C)
#     theta_deg = np.degrees(theta_)
#     theta_deg = torch.degrees(theta_)
    theta_deg = torch.rad2deg(theta_)


 # Return eccentricity and orientation information
#     eccentricity = np.sqrt(1 - (b**2 / a**2)) if B**2 - 4 * A * C < 0
else np.sqrt(1 + (b**2 / a**2))
#      eccentricity =


    # Determine if alignment is close to horizontal or vertical
#     is_horizontal = np.isclose(theta % (np.pi / 2), 0, atol=1e-3)
#         Determine if close to horizontal or vertical
#     if np.isclose(theta_deg % 180, 0, atol=5):  # e.g., -5° to +5°
    if torch.isclose((theta_deg % 180), torch.tensor(0.0, device=theta_
deg.device), atol=5.0):
        orientation = "horizontal"
        eccentricity = a**2/b**2
#     elif np.isclose(theta_deg % 180, 90, atol=5):  # e.g., 85° to 95°
    elif torch.isclose((theta_deg % 180), torch.tensor(90.0, device=the
ta_deg.device), atol=5.0):
        orientation = "vertical"
        eccentricity = b**2/a**2
```

```
    else:
        orientation = "diagonal"  # if not close to 0° or 90°
        eccentricity = b**2/a**2
#         eccentricity = a**2/b**2


#     # Return eccentricity and orientation information
#     eccentricity = np.sqrt(1 - (b**2 / a**2)) if B**2 - 4 * A * C < 0
else np.sqrt(1 + (b**2 / a**2))
    return eccentricity#, is_horizontal, theta
```

In [46]:

```
import torch
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
device

# device(type='cuda')
```

Out[46]:

```
device(type='cpu')
```

In [47]:

```
import pandas as pd
import numpy as np
from tqdm import tqdm
from scipy.optimize import minimize


# # Objective function to minimize
# def objective_fn(params, x, y):
#     return np.sum(conics(params, x, y)**2)

# Objective function to minimize
def objective_fn(params, x, y):
    return torch.sum(conics(params, x, y)**2)


import torch
from torch.optim import LBFGS

# # Define the closure function for optimization
# def closure():
#     optimizer.zero_grad()  # Clear previous gradients
#     loss = objective_fn(params, sx, sy)  # Compute loss using objectiv
e_fn
#     loss.backward()  # Backpropagate the gradients
#     return loss


# def closure():
def closure(optimiser, sx ,sy, params ):
    optimizer.zero_grad()
    # Split params into sx and sy (assuming sx and sy are of the same si
ze)
#     sx = params[:len(sx)]  # First part of params for sx
#     sy = params[len(sx):]  # Second part of params for sy

    # Now call the objective function with sx and sy
    loss = objective_fn(params, sx, sy)
    loss.backward()
    return loss



# Define conic equation
```

```
def conics(params, x, y):
    a, b, c, d, e, f = params
    return a * x**2 + b * x * y + c * y**2 + d * x + e * y + f

# # Objective function to minimize
# def objective_fn(params, x, y):
#     return np.sum(conics(params, x, y)**2)

def curvature_conic(A, B, C, D, E, x, y):
    # First derivative (dy/dx)
    numerator_first = -2*A*x - D
    denominator_first = B*x + 2*C*y + E
    dy_dx = numerator_first / denominator_first

    # Second derivative (d^2y/dx^2)
    # Derivatives of the numerator and denominator
    d_numerator_first_dx = -2*A
    d_denominator_first_dx = B

    # Using the quotient rule for the second derivative
    numerator_second = (d_numerator_first_dx * denominator_first) - (nu
merator_first * d_denominator_first_dx)
    denominator_second = denominator_first**2

    d2y_dx2 = numerator_second / denominator_second

    # Curvature calculation
    curvature = d2y_dx2 / (1 + dy_dx**2)**(3/2)

    return curvature

# def average_curvature(A, B, C, D, E, x_start, x_end, num_points=100):
#     x_values = np.linspace(x_start, x_end, num_points)
#     curvature_values = []

#     for x in x_values:
#         # To find corresponding y values, we would typically need to s
olve the conic equation.
#         # For simplicity, we can use a fixed value of y for average ca
lculation, but this may not be accurate.
#         # Here, just for demonstration, we are considering y = 0, whic
h might not always lie on the conic.
#         y = 0
#         curv = curvature_conic(A, B, C, D, E, x, y)
```

```
#         curvature_values.append(curv)

#     # Calculate average curvature
#     average_curv = np.mean(curvature_values)
#     return average_curv

def average_curvature(A, B, C, D, E, x_start, x_end, num_points=100):
    # Create a tensor of x values, using torch.linspace
    x_values = torch.linspace(x_start, x_end, num_points, device='cuda'
if torch.cuda.is_available() else 'cpu')

    # Initialize the curvature_values tensor
    curvature_values = torch.zeros_like(x_values)

    # Compute the curvature for each x value
    for i, x in enumerate(x_values):
        y = 0  # Fixed y value for the calculation (this can be modified
based on specific requirements)
        curvature_values[i] = curvature_conic(A, B, C, D, E, x, y)

    # Calculate the average curvature using torch.mean
    average_curv = torch.mean(curvature_values)

    return average_curv









# #hyperbolic only
# def calculate_eccentricity(A, B, C, s, acc, dis, o, dir_) :
def calculate_eccentricity_fx(A, B, C, s, acc, dis, o, dir_, theta_) :
    eig_val1 = 0.5 * (A + C + np.sqrt((A - C)**2 + B**2))
    eig_val2 = 0.5 * (A + C - np.sqrt((A - C)**2 + B**2))
    a = np.sqrt(1 / np.abs(eig_val1))
    b = np.sqrt(1 / np.abs(eig_val2))
#     a, b = sorted([a, b], key=abs)
#     a, b = sorted([a, b], key=abs, reverse=True)
#     a, b = sorted([a, b])
    a, b = sorted([a, b], reverse=True)

#         eccentricity = np.sqrt(1 - (b**2 / a**2))
#     eccentricity = np.sqrt(1 + (b**2 / a**2))
```

```
#     return np.sqrt(1 + (b**2 / a**2)) #
#     return (b**2 / a**2)
#     return np.log(b**2 / a**2)
#     return (a / b) #asect ratio
    return (a**2/b**2) # not ^
#     return 2*b**2/a  #latus rectum lr
#     return np.abs(theta_)
#     return B**2 - 4 * A * C #dis
#     return (b**2 / a**2) #asect ratio
#     return np.sqrt((b**2 / a**2))
#else return np.nan



# Initial guess for the conic parameters
# p0 = [1, 1, 1, 1, 1, 1]
p0 = torch.tensor([1, 1, 1, 1, 1, 1], dtype=torch.float32, requires_gra
d=True, device=device)
```

In [ ]:

```
import pandas as pd
import numpy as np
from tqdm import tqdm
from scipy.optimize import minimize


# # Objective function to minimize
# def objective_fn(params, x, y):
#     return np.sum(conics(params, x, y)**2)

# Objective function to minimize
def objective_fn(params, x, y):
    return torch.sum(conics(params, x, y)**2)


import torch
from torch.optim import LBFGS

# # Define the closure function for optimization
# def closure():
#     optimizer.zero_grad()  # Clear previous gradients
#     loss = objective_fn(params, sx, sy)  # Compute loss using objectiv
e_fn
#     loss.backward()  # Backpropagate the gradients
#     return loss


# def closure():
def closure(optimiser, sx ,sy, params ):
    optimizer.zero_grad()
    # Split params into sx and sy (assuming sx and sy are of the same si
ze)
#     sx = params[:len(sx)]  # First part of params for sx
#     sy = params[len(sx):]  # Second part of params for sy

    # Now call the objective function with sx and sy
    loss = objective_fn(params, sx, sy)
    loss.backward()
    return loss



# Define conic equation
```

```
def conics(params, x, y):
    a, b, c, d, e, f = params
    return a * x**2 + b * x * y + c * y**2 + d * x + e * y + f

# # Objective function to minimize
# def objective_fn(params, x, y):
#     return np.sum(conics(params, x, y)**2)

def curvature_conic(A, B, C, D, E, x, y):
    # First derivative (dy/dx)
    numerator_first = -2*A*x - D
    denominator_first = B*x + 2*C*y + E
    dy_dx = numerator_first / denominator_first

    # Second derivative (d^2y/dx^2)
    # Derivatives of the numerator and denominator
    d_numerator_first_dx = -2*A
    d_denominator_first_dx = B

    # Using the quotient rule for the second derivative
    numerator_second = (d_numerator_first_dx * denominator_first) - (nu
merator_first * d_denominator_first_dx)
    denominator_second = denominator_first**2

    d2y_dx2 = numerator_second / denominator_second

    # Curvature calculation
    curvature = d2y_dx2 / (1 + dy_dx**2)**(3/2)

    return curvature

# def average_curvature(A, B, C, D, E, x_start, x_end, num_points=100):
#     x_values = np.linspace(x_start, x_end, num_points)
#     curvature_values = []

#     for x in x_values:
#         # To find corresponding y values, we would typically need to s
olve the conic equation.
#         # For simplicity, we can use a fixed value of y for average ca
lculation, but this may not be accurate.
#         # Here, just for demonstration, we are considering y = 0, whic
h might not always lie on the conic.
#         y = 0
#         curv = curvature_conic(A, B, C, D, E, x, y)
```

```
#         curvature_values.append(curv)

#     # Calculate average curvature
#     average_curv = np.mean(curvature_values)
#     return average_curv

def average_curvature(A, B, C, D, E, x_start, x_end, num_points=100):
    # Create a tensor of x values, using torch.linspace
    x_values = torch.linspace(x_start, x_end, num_points, device='cuda'
if torch.cuda.is_available() else 'cpu')

    # Initialize the curvature_values tensor
    curvature_values = torch.zeros_like(x_values)

    # Compute the curvature for each x value
    for i, x in enumerate(x_values):
        y = 0  # Fixed y value for the calculation (this can be modified
based on specific requirements)
        curvature_values[i] = curvature_conic(A, B, C, D, E, x, y)

    # Calculate the average curvature using torch.mean
    average_curv = torch.mean(curvature_values)

    return average_curv









# #hyperbolic only
# def calculate_eccentricity(A, B, C, s, acc, dis, o, dir_) :
def calculate_eccentricity_fx(A, B, C, s, acc, dis, o, dir_, theta_) :
    eig_val1 = 0.5 * (A + C + np.sqrt((A - C)**2 + B**2))
    eig_val2 = 0.5 * (A + C - np.sqrt((A - C)**2 + B**2))
    a = np.sqrt(1 / np.abs(eig_val1))
    b = np.sqrt(1 / np.abs(eig_val2))
#     a, b = sorted([a, b], key=abs)
#     a, b = sorted([a, b], key=abs, reverse=True)
#     a, b = sorted([a, b])
    a, b = sorted([a, b], reverse=True)

#         eccentricity = np.sqrt(1 - (b**2 / a**2))
#     eccentricity = np.sqrt(1 + (b**2 / a**2))
```

```
#     return np.sqrt(1 + (b**2 / a**2)) #
#     return (b**2 / a**2)
#     return np.log(b**2 / a**2)
#     return (a / b) #asect ratio
    return (a**2/b**2) # not ^
#     return 2*b**2/a  #latus rectum lr
#     return np.abs(theta_)
#     return B**2 - 4 * A * C #dis
#     return (b**2 / a**2) #asect ratio
#     return np.sqrt((b**2 / a**2))
#else return np.nan



# Initial guess for the conic parameters
# p0 = [1, 1, 1, 1, 1, 1]
p0 = torch.tensor([1, 1, 1, 1, 1, 1], dtype=torch.float32, requires_gra
d=True, device=device)


# Initialize the main dictionary to store no transforms
no_transforms = {}

# Summary stats for tracking processing
play_count_with_snap_event = 0
play_count_without_snap_event = 0
events_without_snap = {}

n = 200 #50 #500  # Subset for testing
# tracking_data_dict_play_subset = dict(list(tracking_data_dict_play.ite
ms())[:n])
tracking_data_dict_play_subset = dict(list(tracking_data_dict_play.item
s()))


# Device setup: Use GPU if available
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")


for (gameId, playId), group_data in tqdm(tracking_data_dict_play_subse
t.items()) :

    # Initialize gameId in no_transforms if not already present
    if gameId not in no_transforms:
```

```
        no_transforms[gameId] = {}

    # Initialize playId within the gameId if not already present
    if playId not in no_transforms[gameId]:
        no_transforms[gameId][playId] = {
            'offense': {},
            'defense': {},
        }


    # offensiveTeamAbbr = plays[(plays['gameId'] == gameId) & (plays['pl
ayId'] == playId)].iloc[0]['possessionTeam']
    # defensiveTeamAbbr = plays[(plays['gameId'] == gameId) & (plays['pl
ayId'] == playId)].iloc[0]['defensiveTeam']
    offensiveTeamAbbr = sup_data[(sup_data['game_id'] == gameId) & (sup
_data['play_id'] == playId)].iloc[0]['possession_team']
    defensiveTeamAbbr = sup_data[(sup_data['game_id'] == gameId) & (sup
_data['play_id'] == playId)].iloc[0]['defensive_team']

    # Retrieve tracking data for the current game/play
    # tracking_data = group_data['tracking_data']
    tracking_data = group_data['input']
    tracking_data_op = group_data['output']



    # Process tracking data for each player (nflId)
    for nflId, player_data in tracking_data.groupby('nfl_id'):
        if pd.isna(nflId):  # Skip if it's NaN (ball)
            continue

        # ball_snap_idx = player_data[player_data['event'] == 'ball_sna
p'].index.min()
        # snap_direct_idx = player_data[player_data['event'] == 'snap_di
rect'].index.min()

        # snap_event_idx = min(ball_snap_idx, snap_direct_idx) if pd.not
na(snap_direct_idx) and pd.notna(ball_snap_idx) else (ball_snap_idx if p
d.notna(ball_snap_idx) else snap_direct_idx)



#         # Assuming player_data is sorted by time, so that the last row
represents the latest event
```

```
#         last_event_idx = player_data.groupby('player_id')['event'].idx
max()  # Adjust to get the last event by time
#         last_event = player_data.loc[last_event_idx]


        snap_event_idx = 30
        # player_data_truncated = player_data

        if pd.notna(snap_event_idx):
            # player_data_truncated = player_data.loc[:snap_event_idx]
            player_data_truncated = player_data
            # player_data_truncated = player_data["input"]
            # play_count_with_snap_event += 1
#         else :
#             print("No")
#             player_data_truncated = None

            # print(player_data.columns)









#             x_ = player_data_truncated['x'].values
            x_ = torch.tensor(player_data_truncated['x'].values, device
=device)
            y_ = torch.tensor(player_data_truncated['y'].values, device
=device)
            s_ = torch.tensor(player_data_truncated['s'].values, device
=device)
            a_ = torch.tensor(player_data_truncated['a'].values, device
=device)
            # dis_ = torch.tensor(player_data_truncated['dis'].values, d
evice=device)
            dir_ = torch.tensor(player_data_truncated['dir'].values, de
vice=device)

            o_ = torch.tensor(player_data_truncated['o'].values, device
=device)

            #
# #             disp = np.sum(dis_) #$disall = np.sum(dis_)
#             disp = player_data_truncated['dis'].sum()  # Instead of n
```

```
p.sum
#             xl = x_[-1]
#             yl = y_[-1]
# #             x_mean = np.mean(x_)
# #             y_mean = np.mean(y_)
#             x_mean = player_data_truncated['x'].mean()
#             y_mean = player_data_truncated['y'].mean()

#             # Sum of 'dis' (already in CUDA if player_data_truncated
['dis'] is transferred as tensor)
#             disp = dis_.sum()  # PyTorch automatically performs the su
m on GPU


            #
#             # Last elements of x_ and y_ (directly available in GPU te
nsors)
#             xl = x_[-1]
#             yl = y_[-1]

#             # Mean calculations using PyTorch
#             x_mean = x_.mean()  # Mean calculated on GPU
#             y_mean = y_.mean()

            #
            #optimized version
            # disp = dis_.sum()  # Sum of 'dis' directly on GPU
            xl = x_[-1]        # Last x-coordinate
            yl = y_[-1]        # Last y-coordinate
            x_mean = x_.mean() # Mean of x-coordinates on GPU
            y_mean = y_.mean() # Mean of y-coordinates on GPU



            #
#             dir_rad = np.radians(dir_)
#             sx = s_*np.sin(dir_rad) #np.mean(s_*np.sin()
#             sy = s_*np.cos(dir_rad)
#             ax = a_*np.sin(dir_rad)
#             ay = a_*np.cos(dir_rad)
#             param_s = minimize(objective_fn, p0, args=(sx, sy), method
='L-BFGS-B') #xs
#             param_a = minimize(objective_fn, p0, args=(sy, ay), method
='L-BFGS-B')
#             sa,sb,sc,sd,se,sf = param_s.x #$params_s.
```

```
#             aa,ab,ac,ad,ae,af = param_a.x
#             theta_s = np.arctan2(sb,sa-sc)/2
#             theta_a = np.arctan2(ab,aa-ac)/2

            #
#             # Convert dir_ to radians
#             dir_rad = torch.radians(dir_)

#             # Trigonometric operations (PyTorch versions)
#             sx = s_ * torch.sin(dir_rad)  # PyTorch handles this on GP
U
#             sy = s_ * torch.cos(dir_rad)
#             ax = a_ * torch.sin(dir_rad)
#             ay = a_ * torch.cos(dir_rad)
#             param_s = minimize(objective_fn, p0, args=(sx.cpu().numpy
(), sy.cpu().numpy()), method='L-BFGS-B')
#             param_a = minimize(objective_fn, p0, args=(sy.cpu().numpy
(), ay.cpu().numpy()), method='L-BFGS-B')

#             # Extract parameters
#             sa, sb, sc, sd, se, sf = param_s.x
#             aa, ab, ac, ad, ae, af = param_a.x

#             # Trigonometric calculations
#             theta_s = torch.arctan2(torch.tensor(sb), torch.tensor(sa
- sc)) / 2
#             theta_a = torch.arctan2(torch.tensor(ab), torch.tensor(aa
- ac)) / 2



            dir_rad = torch.deg2rad(dir_) #not radians
            sx = s_ * torch.sin(dir_rad)  # PyTorch handles this on GPU
            sy = s_ * torch.cos(dir_rad)
            ax = a_ * torch.sin(dir_rad)
            ay = a_ * torch.cos(dir_rad)

            # Define initial parameters for optimization
#             params = torch.tensor(p0, requires_grad=True, device=devic
e)
            params = p0.clone().detach().requires_grad_(True).to(devic
e)
```

```
            # Create LBFGS optimizer
            optimizer = LBFGS([params])

#             # Step through the optimizer
#             optimizer.step(closure)

            # Use closure for (sx, sy)
            optimizer.step(lambda: closure(optimizer, sx, sy, params))
#             optimizer.step(lambda: closure(optimizer, sx, sy))

            # After optimization, the parameters are updated. You can no
w access the optimized values:
            optimized_params = params.detach()  # Detach the tensor fro
m the computation graph

            # Extract the optimized parameters (sa, sb, sc, sd, se, sf)
for further use
            sa, sb, sc, sd, se, sf = optimized_params.tolist()  # Conve
rt tensor to list for easy usage

            # Perform any further calculations or use these parameters i
n your code
#             theta_s = torch.arctan2(sb, sa - sc) / 2
            theta_s = torch.arctan2(torch.tensor(sb, device=device), to
rch.tensor(sa - sc, device=device)) / 2




#             params = torch.tensor(p0, requires_grad=True, device=devic
e)
            params = p0.clone().detach().requires_grad_(True).to(devic
e)
            optimizer = LBFGS([params])
            optimizer.step(lambda: closure(optimizer, ax, ay, params))
            optimized_params = params.detach()
            aa, ab, ac, ad, ae, af = optimized_params.tolist()
#             theta_a = torch.arctan2(ab, aa - ac) / 2
            theta_a = torch.arctan2(torch.tensor(ab, device=device), to
rch.tensor(aa - ac, device=device)) / 2
```

```
#             params = torch.tensor(p0, requires_grad=True, device=devic
e)
            params = p0.clone().detach().requires_grad_(True).to(devic
e)
            optimizer = LBFGS([params])
            optimizer.step(lambda: closure(optimizer, x_, y_, params))
# xx or xa
            optimized_params = params.detach()
            a,b,c,d,e,f = optimized_params.tolist() # aa bb cc ..
#             theta_ = torch.arctan2(ab, aa - ac) / 2
            theta_ = torch.arctan2(torch.tensor(b, device=device), torc
h.tensor(a - c, device=device)) / 2
            cos_theta_, sin_theta_ = torch.cos(theta_), torch.sin(theta
_)




#             params = minimize(objective_fn, p0, args=(x_, y_), method
='L-BFGS-B')
#             a,b,c,d,e,f = params.x

#             theta_ = np.arctan2(b,a-c)/2
#             cos_theta_, sin_theta_ = np.cos(theta_), np.sin(theta_)

            #####%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
#             a_prime = a*cos_theta_**2 +  b*cos_theta_*sin_theta_ +  c*
sin_theta_**2
#             b_prime = 2*(c-a)*cos_theta_*sin_theta_ + b*(cos_theta_**2
-sin_theta_**2)
#             c_prime = a*sin_theta_**2 - b*cos_theta_*sin_theta_ + c*co
s_theta_**2


#             params = torch.tensor(p0, requires_grad=True, device=devic
e)
#             optimizer = LBFGS([params])
#             optimizer.step(closure)
#             optimized_params = params.detach()
#             sa, sb, sc, sd, se, sf = optimized_params.tolist()
#             theta_s = torch.arctan2(sb, sa - sc) / 2
#             theta_a = torch.arctan2(ab, aa - ac) / 2
```

```
#             eccentricity = calculate_eccentricity(a, b, c, np.mean(s
_), np.mean(a_), np.mean(dis_), np.mean(o_), np.mean(dir_), theta_)
#             eccentricity = calculate_eccentricity(a, b, c, torch.mean
(s_), torch.mean(a_), torch.mean(dis_), torch.mean(o_), torch.mean(dir
_), theta_)
# %%%%%%%%%%%%%%%%%%%%%%%%%%%%

#             curvature = curvature_conic(a,b,c,d,e, np.mean(x_), np.mea
n(y_) )
#             curvature = curvature_conic(a, b, c, d, e, torch.mean(x_),
torch.mean(y_))
    # %%%%%%%%%%%%%%%%%%%%%%%%%%%%

#             avg_cuvx = average_curvature(a,b,c,d,e, x_[0], x_[-1] )
#             avg_cuvy = average_curvature(a,b,c,d,e, y_[0], y_[-1] )
#             discriminant = b**2 - 4 * a * c
  # %%%%%%%%%%%%%%%%%%%%%%%%%%%%

#             eig_val1 = 0.5 * (a + c + np.sqrt((a - c)**2 + b**2))
#             eig_val2 = 0.5 * (a + c - np.sqrt((a - c)**2 + b**2))
#             al = np.sqrt(1 / np.abs(eig_val1))
#             bl = np.sqrt(1 / np.abs(eig_val2))
#             al, bl = sorted([al, bl], reverse=True)
    # %%%%%%%%%%%%%%%%%%%%%%%%%%%%


#             arxy = calculate_eccentricity_fx(a, b, c, np.mean(s_), np.
mean(a_), np.mean(dis_), np.mean(o_), np.mean(dir_), theta_) #aspect rat
io
#             ars = calculate_eccentricity_fx(sa, sb, sc, np.mean(s_), n
p.mean(a_), np.mean(dis_), np.mean(o_), np.mean(dir_), theta_)
#             ara = calculate_eccentricity_fx(aa, ab, ac, np.mean(s_), n
p.mean(a_), np.mean(dis_), np.mean(o_), np.mean(dir_), theta_)
#             arxy = calculate_eccentricity_fx(a, b, c, torch.mean(s_),
torch.mean(a_), torch.mean(dis_), torch.mean(o_), torch.mean(dir_), thet
a_) #aspect ratio
#             ars = calculate_eccentricity_fx(sa, sb, sc, torch.mean(s
_), torch.mean(a_), torch.mean(dis_), torch.mean(o_), torch.mean(dir_),
theta_)
#             ara = calculate_eccentricity_fx(aa, ab, ac, torch.mean(s
_), torch.mean(a_), torch.mean(dis_), torch.mean(o_), torch.mean(dir_),
theta_)
# %%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
            # if player_data['club'].iloc[0] == offensiveTeamAbbr:
            if player_data['player_side'].iloc[0] == "Offense":
#                 no_transforms[gameId][playId]['offense'][nflId] = {'ec
centricity': eccentricity, 'disp':disp, 'theta': theta_, 'cuv':curvatur
e, 'cuvx': avg_cuvx, 'cuvy': avg_cuvy, 'disc':discriminant, 'major':al,
'minor':bl, 'arxy':arxy, 'ars':ars, 'ara': ara, 'theta_s': theta_s, 'the
ta_a':theta_a, 'xl':xl, 'yl':yl, 'x_mean':x_mean, 'y_mean':y_mean, 'game
id': group_data['meta_data']['gameId'], 'playid': group_data['meta_dat
a']['playId'], 'a'=a,'b'= b,'c'=c,'d'=d,'e'=e,'f'=f, 'sa'=sa, 'sb'=sb,
'sc'=sc, s'sd'=d, 'se'=se, 'sf'sf, 'aa'=aa, 'ab'=ab, 'ac'= ac, 'ad'=ad,
'ae'=ae, 'af'=af}
#                 no_transforms[gameId][playId]['offense'][nflId] =
{'a'=a,'b'= b,'c'=c,'d'=d,'e'=e,'f'=f, 'sa'=sa, 'sb'=sb, 'sc'=sc, s'sd'=
d, 'se'=se, 'sf'sf, 'aa'=aa, 'ab'=ab, 'ac'= ac, 'ad'=ad, 'ae'=ae, 'af'=a
f}
                no_transforms[gameId][playId]['offense'][nflId] = {'a':
a,'b': b,'c': c,'d': d,'e': e,'f': f, 'sa': sa, 'sb': sb, 'sc':sc, 's
d': sd, 'se':se, 'sf': sf, 'aa': aa, 'ab': ab, 'ac': ac, 'ad': ad, 'a
e': ae, 'af': af, 'xl':xl, 'yl':yl, 'x_mean':x_mean, 'y_mean':y_mean,
'theta_':theta_, 'theta_s':theta_s, 'theta_a':theta_a, 'gameid': group_
data['meta_data']['gameId'], 'playid': group_data['meta_data']['playI
d']} #, 'last_event'=last_event}
                # 'disp':disp,

            # elif player_data['club'].iloc[0] == defensiveTeamAbbr:
            elif player_data['player_side'].iloc[0] == "Defense":
#                 no_transforms[gameId][playId]['defense'][nflId] = {'ec
centricity': eccentricity, 'disp':disp, 'theta': theta_, 'cuv':curvatur
e,  'cuvx': avg_cuvx, 'cuvy': avg_cuvy, 'disc':discriminant, 'major':al,
'minor':bl, 'arxy':arxy, 'ars':ars, 'ara': ara, 'theta_s': theta_s, 'the
ta_a':theta_a, 'xl':xl, 'yl':yl, 'x_mean':x_mean, 'y_mean':y_mean, 'game
id': group_data['meta_data']['gameId'], 'playid': group_data['meta_dat
a']['playId']}
                no_transforms[gameId][playId]['defense'][nflId] = {'a':
a,'b': b,'c': c,'d': d,'e': e,'f': f, 'sa': sa, 'sb': sb, 'sc':sc, 's
d': sd, 'se':se, 'sf': sf, 'aa': aa, 'ab': ab, 'ac': ac, 'ad': ad, 'a
e': ae, 'af': af, 'xl':xl, 'yl':yl, 'x_mean':x_mean, 'y_mean':y_mean,
'theta_':theta_, 'theta_s':theta_s, 'theta_a':theta_a, 'gameid': group_
data['meta_data']['gameId'], 'playid': group_data['meta_data']['playI
d']} #, 'last_event'=last_event}
                # 'disp':disp,

        else:
```

```
            play_count_without_snap_event += 1
            # events_without_snap[(gameId, playId)] = player_data['even
t'].unique()  # Store unique events

# Final summary
print(f"Total plays processed: {len(tracking_data_dict_play_subset)}")
print(f"Plays with 'ball_snap' or 'snap_direct' event: {play_count_with
_snap_event}")
print(f"Plays without 'ball_snap' or 'snap_direct' event: {play_count_w
ithout_snap_event}")


# torch.save(no_transforms, 'no_transforms[all].pth')

#  12%|▉         | 402/3347 [14:22<1:43:40,  2.11s/it]j
#   0%|          | 13/3362 [01:03<4:29:05,  4.82s/it] #gpu


#   7%|▋         | 1070/16124 [25:11<5:35:34,  1.34s/it]
#  51%|█████     | 8165/16124 [3:20:31<3:22:41,  1.53s/it]
```

In [ ]:

```
import pandas as pd
import numpy as np
from tqdm import tqdm
from scipy.optimize import minimize


# Objective function to minimize
def objective_fn(params, x, y):
    return torch.sum(conics(params, x, y)**2)


import torch
from torch.optim import LBFGS

# # Define the closure function for optimization


# def closure():
def closure(optimiser, sx ,sy, params ):
    optimizer.zero_grad()
    # Split params into sx and sy (assuming sx and sy are of the same si
ze)


    # Now call the objective function with sx and sy
    loss = objective_fn(params, sx, sy)
    loss.backward()
    return loss



# Define conic equation
def conics(params, x, y):
    a, b, c, d, e, f = params
    return a * x**2 + b * x * y + c * y**2 + d * x + e * y + f


def curvature_conic(A, B, C, D, E, x, y):
    # First derivative (dy/dx)
    numerator_first = -2*A*x - D
    denominator_first = B*x + 2*C*y + E
    dy_dx = numerator_first / denominator_first
```

```
    # Second derivative (d^2y/dx^2)
    # Derivatives of the numerator and denominator
    d_numerator_first_dx = -2*A
    d_denominator_first_dx = B

    # Using the quotient rule for the second derivative
    numerator_second = (d_numerator_first_dx * denominator_first) - (nu
merator_first * d_denominator_first_dx)
    denominator_second = denominator_first**2

    d2y_dx2 = numerator_second / denominator_second

    # Curvature calculation
    curvature = d2y_dx2 / (1 + dy_dx**2)**(3/2)

    return curvature



def average_curvature(A, B, C, D, E, x_start, x_end, num_points=100):
    # Create a tensor of x values, using torch.linspace
    x_values = torch.linspace(x_start, x_end, num_points, device='cuda'
if torch.cuda.is_available() else 'cpu')

    # Initialize the curvature_values tensor
    curvature_values = torch.zeros_like(x_values)

    # Compute the curvature for each x value
    for i, x in enumerate(x_values):
        y = 0  # Fixed y value for the calculation (this can be modified
based on specific requirements)
        curvature_values[i] = curvature_conic(A, B, C, D, E, x, y)

    # Calculate the average curvature using torch.mean
    average_curv = torch.mean(curvature_values)

    return average_curv







# #hyperbolic only
```

```
# def calculate_eccentricity(A, B, C, s, acc, dis, o, dir_) :
def calculate_eccentricity_fx(A, B, C, s, acc, dis, o, dir_, theta_) :
    eig_val1 = 0.5 * (A + C + np.sqrt((A - C)**2 + B**2))
    eig_val2 = 0.5 * (A + C - np.sqrt((A - C)**2 + B**2))
    a = np.sqrt(1 / np.abs(eig_val1))
    b = np.sqrt(1 / np.abs(eig_val2))

    a, b = sorted([a, b], reverse=True)


    return (a**2/b**2)



# Initial guess for the conic parameters
# p0 = [1, 1, 1, 1, 1, 1]
p0 = torch.tensor([1, 1, 1, 1, 1, 1], dtype=torch.float32, requires_gra
d=True, device=device)


# Initialize the main dictionary to store no transforms
no_transforms = {}

# Summary stats for tracking processing
play_count_with_snap_event = 0
play_count_without_snap_event = 0
events_without_snap = {}

n = 200 #50 #500  # Subset for testing
# tracking_data_dict_play_subset = dict(list(tracking_data_dict_play.ite
ms())[:n])
tracking_data_dict_play_subset = dict(list(tracking_data_dict_play.item
s()))


# Device setup: Use GPU if available
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(device)


for (gameId, playId), group_data in tqdm(tracking_data_dict_play_subse
t.items()) :

    # Initialize gameId in no_transforms if not already present
```

```
    if gameId not in no_transforms:
        no_transforms[gameId] = {}

    # Initialize playId within the gameId if not already present
    if playId not in no_transforms[gameId]:
        no_transforms[gameId][playId] = {
            'offense': {},
            'defense': {},
        }


    # offensiveTeamAbbr = plays[(plays['gameId'] == gameId) & (plays['pl
ayId'] == playId)].iloc[0]['possessionTeam']
    # defensiveTeamAbbr = plays[(plays['gameId'] == gameId) & (plays['pl
ayId'] == playId)].iloc[0]['defensiveTeam']
    offensiveTeamAbbr = sup_data[(sup_data['game_id'] == gameId) & (sup
_data['play_id'] == playId)].iloc[0]['possession_team']
    defensiveTeamAbbr = sup_data[(sup_data['game_id'] == gameId) & (sup
_data['play_id'] == playId)].iloc[0]['defensive_team']

    # Retrieve tracking data for the current game/play
    # tracking_data = group_data['tracking_data']
    tracking_data = group_data['input']
    tracking_data_op = group_data['output']



    # Process tracking data for each player (nflId)
    for nflId, player_data in tracking_data.groupby('nfl_id'):
        if pd.isna(nflId):  # Skip if it's NaN (ball)
            continue



        snap_event_idx = 30
        # player_data_truncated = player_data

        if pd.notna(snap_event_idx):
            # player_data_truncated = player_data.loc[:snap_event_idx]
            player_data_truncated = player_data
```

```
#             x_ = player_data_truncated['x'].values
            x_ = torch.tensor(player_data_truncated['x'].values, device
=device)
            y_ = torch.tensor(player_data_truncated['y'].values, device
=device)
            s_ = torch.tensor(player_data_truncated['s'].values, device
=device)
            a_ = torch.tensor(player_data_truncated['a'].values, device
=device)
            # dis_ = torch.tensor(player_data_truncated['dis'].values, d
evice=device)
            dir_ = torch.tensor(player_data_truncated['dir'].values, de
vice=device)

            o_ = torch.tensor(player_data_truncated['o'].values, device
=device)

            #
# #             disp = np.sum(dis_) #$disall = np.sum(dis_)
#             disp = player_data_truncated['dis'].sum()  # Instead of n
p.sum
#             xl = x_[-1]
#             yl = y_[-1]
# #             x_mean = np.mean(x_)
# #             y_mean = np.mean(y_)
#             x_mean = player_data_truncated['x'].mean()
#             y_mean = player_data_truncated['y'].mean()

#             # Sum of 'dis' (already in CUDA if player_data_truncated
['dis'] is transferred as tensor)
#             disp = dis_.sum()  # PyTorch automatically performs the su
m on GPU

            #
#             # Last elements of x_ and y_ (directly available in GPU te
nsors)
#             xl = x_[-1]
#             yl = y_[-1]

#             # Mean calculations using PyTorch
#             x_mean = x_.mean()  # Mean calculated on GPU
#             y_mean = y_.mean()
```

```
            #
            #optimized version
            # disp = dis_.sum()  # Sum of 'dis' directly on GPU
            xl = x_[-1]        # Last x-coordinate
            yl = y_[-1]        # Last y-coordinate
            x_mean = x_.mean() # Mean of x-coordinates on GPU
            y_mean = y_.mean() # Mean of y-coordinates on GPU



            #
#             dir_rad = np.radians(dir_)
#             sx = s_*np.sin(dir_rad) #np.mean(s_*np.sin()
#             sy = s_*np.cos(dir_rad)
#             ax = a_*np.sin(dir_rad)
#             ay = a_*np.cos(dir_rad)
#             param_s = minimize(objective_fn, p0, args=(sx, sy), method
='L-BFGS-B') #xs
#             param_a = minimize(objective_fn, p0, args=(sy, ay), method
='L-BFGS-B')
#             sa,sb,sc,sd,se,sf = param_s.x #$params_s.
#             aa,ab,ac,ad,ae,af = param_a.x
#             theta_s = np.arctan2(sb,sa-sc)/2
#             theta_a = np.arctan2(ab,aa-ac)/2

            #
#             # Convert dir_ to radians
#             dir_rad = torch.radians(dir_)

#             # Trigonometric operations (PyTorch versions)
#             sx = s_ * torch.sin(dir_rad)  # PyTorch handles this on GP
U
#             sy = s_ * torch.cos(dir_rad)
#             ax = a_ * torch.sin(dir_rad)
#             ay = a_ * torch.cos(dir_rad)
#             param_s = minimize(objective_fn, p0, args=(sx.cpu().numpy
(), sy.cpu().numpy()), method='L-BFGS-B')
#             param_a = minimize(objective_fn, p0, args=(sy.cpu().numpy
(), ay.cpu().numpy()), method='L-BFGS-B')

#             # Extract parameters
#             sa, sb, sc, sd, se, sf = param_s.x
#             aa, ab, ac, ad, ae, af = param_a.x
```

```
#             # Trigonometric calculations
#             theta_s = torch.arctan2(torch.tensor(sb), torch.tensor(sa
- sc)) / 2
#             theta_a = torch.arctan2(torch.tensor(ab), torch.tensor(aa
- ac)) / 2



            dir_rad = torch.deg2rad(dir_) #not radians
            sx = s_ * torch.sin(dir_rad)  # PyTorch handles this on GPU
            sy = s_ * torch.cos(dir_rad)
            ax = a_ * torch.sin(dir_rad)
            ay = a_ * torch.cos(dir_rad)

            # Define initial parameters for optimization
#             params = torch.tensor(p0, requires_grad=True, device=devic
e)
            params = p0.clone().detach().requires_grad_(True).to(devic
e)



            # Create LBFGS optimizer
            optimizer = LBFGS([params])

#             # Step through the optimizer
#             optimizer.step(closure)

            # Use closure for (sx, sy)
            optimizer.step(lambda: closure(optimizer, sx, sy, params))
#             optimizer.step(lambda: closure(optimizer, sx, sy))

            # After optimization, the parameters are updated. You can no
w access the optimized values:
            optimized_params = params.detach()  # Detach the tensor fro
m the computation graph

            # Extract the optimized parameters (sa, sb, sc, sd, se, sf)
for further use
            sa, sb, sc, sd, se, sf = optimized_params.tolist()  # Conve
rt tensor to list for easy usage

            # Perform any further calculations or use these parameters i
n your code
```

```
#             theta_s = torch.arctan2(sb, sa - sc) / 2
            theta_s = torch.arctan2(torch.tensor(sb, device=device), to
rch.tensor(sa - sc, device=device)) / 2




#             params = torch.tensor(p0, requires_grad=True, device=devic
e)
            params = p0.clone().detach().requires_grad_(True).to(devic
e)
            optimizer = LBFGS([params])
            optimizer.step(lambda: closure(optimizer, ax, ay, params))
            optimized_params = params.detach()
            aa, ab, ac, ad, ae, af = optimized_params.tolist()
#             theta_a = torch.arctan2(ab, aa - ac) / 2
            theta_a = torch.arctan2(torch.tensor(ab, device=device), to
rch.tensor(aa - ac, device=device)) / 2




#             params = torch.tensor(p0, requires_grad=True, device=devic
e)
            params = p0.clone().detach().requires_grad_(True).to(devic
e)
            optimizer = LBFGS([params])
            optimizer.step(lambda: closure(optimizer, x_, y_, params))
# xx or xa
            optimized_params = params.detach()
            a,b,c,d,e,f = optimized_params.tolist() # aa bb cc ..
#             theta_ = torch.arctan2(ab, aa - ac) / 2
            theta_ = torch.arctan2(torch.tensor(b, device=device), torc
h.tensor(a - c, device=device)) / 2
            cos_theta_, sin_theta_ = torch.cos(theta_), torch.sin(theta
_)




#             arxy = calculate_eccentricity_fx(a, b, c, np.mean(s_), np.
mean(a_), np.mean(dis_), np.mean(o_), np.mean(dir_), theta_) #aspect rat
io
```

```
            # if player_data['club'].iloc[0] == offensiveTeamAbbr:
            if player_data['player_side'].iloc[0] == "Offense":
#                 no_transforms[gameId][playId]['offense'][nflId] = {'ec
centricity': eccentricity, 'disp':disp, 'theta': theta_, 'cuv':curvatur
e, 'cuvx': avg_cuvx, 'cuvy': avg_cuvy, 'disc':discriminant, 'major':al,
'minor':bl, 'arxy':arxy, 'ars':ars, 'ara': ara, 'theta_s': theta_s, 'the
ta_a':theta_a, 'xl':xl, 'yl':yl, 'x_mean':x_mean, 'y_mean':y_mean, 'game
id': group_data['meta_data']['gameId'], 'playid': group_data['meta_dat
a']['playId'], 'a'=a,'b'= b,'c'=c,'d'=d,'e'=e,'f'=f, 'sa'=sa, 'sb'=sb,
'sc'=sc, s'sd'=d, 'se'=se, 'sf'sf, 'aa'=aa, 'ab'=ab, 'ac'= ac, 'ad'=ad,
'ae'=ae, 'af'=af}
#                 no_transforms[gameId][playId]['offense'][nflId] =
{'a'=a,'b'= b,'c'=c,'d'=d,'e'=e,'f'=f, 'sa'=sa, 'sb'=sb, 'sc'=sc, s'sd'=
d, 'se'=se, 'sf'sf, 'aa'=aa, 'ab'=ab, 'ac'= ac, 'ad'=ad, 'ae'=ae, 'af'=a
f}
                no_transforms[gameId][playId]['offense'][nflId] = {'a':
a,'b': b,'c': c,'d': d,'e': e,'f': f, 'sa': sa, 'sb': sb, 'sc':sc, 's
d': sd, 'se':se, 'sf': sf, 'aa': aa, 'ab': ab, 'ac': ac, 'ad': ad, 'a
e': ae, 'af': af, 'xl':xl, 'yl':yl, 'x_mean':x_mean, 'y_mean':y_mean,
'theta_':theta_, 'theta_s':theta_s, 'theta_a':theta_a, 'gameid': group_
data['meta_data']['gameId'], 'playid': group_data['meta_data']['playI
d']} #, 'last_event'=last_event}
                # 'disp':disp,

            # elif player_data['club'].iloc[0] == defensiveTeamAbbr:
            elif player_data['player_side'].iloc[0] == "Defense":
#                 no_transforms[gameId][playId]['defense'][nflId] = {'ec
centricity': eccentricity, 'disp':disp, 'theta': theta_, 'cuv':curvatur
e,  'cuvx': avg_cuvx, 'cuvy': avg_cuvy, 'disc':discriminant, 'major':al,
'minor':bl, 'arxy':arxy, 'ars':ars, 'ara': ara, 'theta_s': theta_s, 'the
ta_a':theta_a, 'xl':xl, 'yl':yl, 'x_mean':x_mean, 'y_mean':y_mean, 'game
id': group_data['meta_data']['gameId'], 'playid': group_data['meta_dat
a']['playId']}
                no_transforms[gameId][playId]['defense'][nflId] = {'a':
a,'b': b,'c': c,'d': d,'e': e,'f': f, 'sa': sa, 'sb': sb, 'sc':sc, 's
d': sd, 'se':se, 'sf': sf, 'aa': aa, 'ab': ab, 'ac': ac, 'ad': ad, 'a
e': ae, 'af': af, 'xl':xl, 'yl':yl, 'x_mean':x_mean, 'y_mean':y_mean,
'theta_':theta_, 'theta_s':theta_s, 'theta_a':theta_a, 'gameid': group_
data['meta_data']['gameId'], 'playid': group_data['meta_data']['playI
d']} #, 'last_event'=last_event}
                # 'disp':disp,

        else:
            play_count_without_snap_event += 1
```

```
            # events_without_snap[(gameId, playId)] = player_data['even
t'].unique()  # Store unique events

# Final summary
print(f"Total plays processed: {len(tracking_data_dict_play_subset)}")
print(f"Plays with 'ball_snap' or 'snap_direct' event: {play_count_with
_snap_event}")
print(f"Plays without 'ball_snap' or 'snap_direct' event: {play_count_w
ithout_snap_event}")



# torch.save(no_transforms, 'no_transforms[all].pth')
```

In [50]:

```
# tracking_data_dict_play_subset[2023120306][1620]
# tracking_data_dict_play_subset.keys()
# tracking_data_dict_play_subset[(2023120306,1620)]
key_ = (2023120306,1620)
data_ = tracking_data_dict_play_subset[(2023120306,1620)]
# dict_gp = dict(key_:data_)
dict_gp = {key_: data_}

tracking_data_dict_play_subset[(2023120306,1620)] == dict_gp
dict_gp[(2023120306,1620)] is tracking_data_dict_play_subset[(202312030
6,1620)]
```

Out[50]:

```
True
```

In [ ]:

```
import pandas as pd
import numpy as np
from tqdm import tqdm
from scipy.optimize import minimize


# Objective function to minimize
def objective_fn(params, x, y):
    return torch.sum(conics(params, x, y)**2)


import torch
from torch.optim import LBFGS

# # Define the closure function for optimization


# def closure():
def closure(optimiser, sx ,sy, params ):
    optimizer.zero_grad()
    # Split params into sx and sy (assuming sx and sy are of the same si
ze)


    # Now call the objective function with sx and sy
    loss = objective_fn(params, sx, sy)
    loss.backward()
    return loss



# Define conic equation
def conics(params, x, y):
    a, b, c, d, e, f = params
    return a * x**2 + b * x * y + c * y**2 + d * x + e * y + f


def curvature_conic(A, B, C, D, E, x, y):
    # First derivative (dy/dx)
    numerator_first = -2*A*x - D
    denominator_first = B*x + 2*C*y + E
    dy_dx = numerator_first / denominator_first
```

```
    # Second derivative (d^2y/dx^2)
    # Derivatives of the numerator and denominator
    d_numerator_first_dx = -2*A
    d_denominator_first_dx = B

    # Using the quotient rule for the second derivative
    numerator_second = (d_numerator_first_dx * denominator_first) - (nu
merator_first * d_denominator_first_dx)
    denominator_second = denominator_first**2

    d2y_dx2 = numerator_second / denominator_second

    # Curvature calculation
    curvature = d2y_dx2 / (1 + dy_dx**2)**(3/2)

    return curvature



def average_curvature(A, B, C, D, E, x_start, x_end, num_points=100):
    # Create a tensor of x values, using torch.linspace
    x_values = torch.linspace(x_start, x_end, num_points, device='cuda'
if torch.cuda.is_available() else 'cpu')

    # Initialize the curvature_values tensor
    curvature_values = torch.zeros_like(x_values)

    # Compute the curvature for each x value
    for i, x in enumerate(x_values):
        y = 0  # Fixed y value for the calculation (this can be modified
based on specific requirements)
        curvature_values[i] = curvature_conic(A, B, C, D, E, x, y)

    # Calculate the average curvature using torch.mean
    average_curv = torch.mean(curvature_values)

    return average_curv







# #hyperbolic only
```

```
# def calculate_eccentricity(A, B, C, s, acc, dis, o, dir_) :
def calculate_eccentricity_fx(A, B, C, s, acc, dis, o, dir_, theta_) :
    eig_val1 = 0.5 * (A + C + np.sqrt((A - C)**2 + B**2))
    eig_val2 = 0.5 * (A + C - np.sqrt((A - C)**2 + B**2))
    a = np.sqrt(1 / np.abs(eig_val1))
    b = np.sqrt(1 / np.abs(eig_val2))

    a, b = sorted([a, b], reverse=True)


    return (a**2/b**2)



# Initial guess for the conic parameters
# p0 = [1, 1, 1, 1, 1, 1]
p0 = torch.tensor([1, 1, 1, 1, 1, 1], dtype=torch.float32, requires_gra
d=True, device=device)


# Initialize the main dictionary to store no transforms
no_transforms = {}

# Summary stats for tracking processing
play_count_with_snap_event = 0
play_count_without_snap_event = 0
events_without_snap = {}

n = 200 #50 #500  # Subset for testing
# tracking_data_dict_play_subset = dict(list(tracking_data_dict_play.ite
ms())[:n])
tracking_data_dict_play_subset = dict(list(tracking_data_dict_play.item
s()))

###NEWLY
tracking_data_dict_play_subset = dict_gp



# Device setup: Use GPU if available
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")


for (gameId, playId), group_data in tqdm(tracking_data_dict_play_subse
t.items()) :
```

```
    # Initialize gameId in no_transforms if not already present
    if gameId not in no_transforms:
        no_transforms[gameId] = {}

    # Initialize playId within the gameId if not already present
    if playId not in no_transforms[gameId]:
        no_transforms[gameId][playId] = {
            'offense': {},
            'defense': {},
        }


    # offensiveTeamAbbr = plays[(plays['gameId'] == gameId) & (plays['pl
ayId'] == playId)].iloc[0]['possessionTeam']
    # defensiveTeamAbbr = plays[(plays['gameId'] == gameId) & (plays['pl
ayId'] == playId)].iloc[0]['defensiveTeam']
    offensiveTeamAbbr = sup_data[(sup_data['game_id'] == gameId) & (sup
_data['play_id'] == playId)].iloc[0]['possession_team']
    defensiveTeamAbbr = sup_data[(sup_data['game_id'] == gameId) & (sup
_data['play_id'] == playId)].iloc[0]['defensive_team']

    # Retrieve tracking data for the current game/play
    # tracking_data = group_data['tracking_data']
    tracking_data = group_data['input']
    tracking_data_op = group_data['output']



    # Process tracking data for each player (nflId)
    for nflId, player_data in tracking_data.groupby('nfl_id'):
        if pd.isna(nflId):  # Skip if it's NaN (ball)
            continue



        snap_event_idx = 30
        # player_data_truncated = player_data

        if pd.notna(snap_event_idx):
            # player_data_truncated = player_data.loc[:snap_event_idx]
            player_data_truncated = player_data
```

```
#             x_ = player_data_truncated['x'].values
            x_ = torch.tensor(player_data_truncated['x'].values, device
=device)
            y_ = torch.tensor(player_data_truncated['y'].values, device
=device)



            ################

            # s_ = torch.tensor(player_data_truncated['s'].values, devic
e=device)
            # a_ = torch.tensor(player_data_truncated['a'].values, devic
e=device)
            # # dis_ = torch.tensor(player_data_truncated['dis'].values,
device=device)
            # dir_ = torch.tensor(player_data_truncated['dir'].values, d
evice=device)

            # o_ = torch.tensor(player_data_truncated['o'].values, devic
e=device)

            ################



            #
# #             disp = np.sum(dis_) #$disall = np.sum(dis_)
#             disp = player_data_truncated['dis'].sum()  # Instead of n
p.sum
#             xl = x_[-1]
#             yl = y_[-1]
# #             x_mean = np.mean(x_)
# #             y_mean = np.mean(y_)
#             x_mean = player_data_truncated['x'].mean()
#             y_mean = player_data_truncated['y'].mean()

#             # Sum of 'dis' (already in CUDA if player_data_truncated
['dis'] is transferred as tensor)
#             disp = dis_.sum()  # PyTorch automatically performs the su
m on GPU
```

```
            #
#             # Last elements of x_ and y_ (directly available in GPU te
nsors)
#             xl = x_[-1]
#             yl = y_[-1]

#             # Mean calculations using PyTorch
#             x_mean = x_.mean()  # Mean calculated on GPU
#             y_mean = y_.mean()

            #
            #optimized version
            # disp = dis_.sum()  # Sum of 'dis' directly on GPU
            xl = x_[-1]        # Last x-coordinate
            yl = y_[-1]        # Last y-coordinate
            x_mean = x_.mean() # Mean of x-coordinates on GPU
            y_mean = y_.mean() # Mean of y-coordinates on GPU



            #
#             dir_rad = np.radians(dir_)
#             sx = s_*np.sin(dir_rad) #np.mean(s_*np.sin()
#             sy = s_*np.cos(dir_rad)
#             ax = a_*np.sin(dir_rad)
#             ay = a_*np.cos(dir_rad)
#             param_s = minimize(objective_fn, p0, args=(sx, sy), method
='L-BFGS-B') #xs
#             param_a = minimize(objective_fn, p0, args=(sy, ay), method
='L-BFGS-B')
#             sa,sb,sc,sd,se,sf = param_s.x #$params_s.
#             aa,ab,ac,ad,ae,af = param_a.x
#             theta_s = np.arctan2(sb,sa-sc)/2
#             theta_a = np.arctan2(ab,aa-ac)/2

            #
#             # Convert dir_ to radians
#             dir_rad = torch.radians(dir_)

#             # Trigonometric operations (PyTorch versions)
#             sx = s_ * torch.sin(dir_rad)  # PyTorch handles this on GP
U
#             sy = s_ * torch.cos(dir_rad)
```

```
#             ax = a_ * torch.sin(dir_rad)
#             ay = a_ * torch.cos(dir_rad)
#             param_s = minimize(objective_fn, p0, args=(sx.cpu().numpy
(), sy.cpu().numpy()), method='L-BFGS-B')
#             param_a = minimize(objective_fn, p0, args=(sy.cpu().numpy
(), ay.cpu().numpy()), method='L-BFGS-B')

#             # Extract parameters
#             sa, sb, sc, sd, se, sf = param_s.x
#             aa, ab, ac, ad, ae, af = param_a.x

#             # Trigonometric calculations
#             theta_s = torch.arctan2(torch.tensor(sb), torch.tensor(sa
- sc)) / 2
#             theta_a = torch.arctan2(torch.tensor(ab), torch.tensor(aa
- ac)) / 2


#####################################

            dir_rad = torch.deg2rad(dir_) #not radians
            sx = s_ * torch.sin(dir_rad)  # PyTorch handles this on GPU
            sy = s_ * torch.cos(dir_rad)
            ax = a_ * torch.sin(dir_rad)
            ay = a_ * torch.cos(dir_rad)

            # Define initial parameters for optimization
#             params = torch.tensor(p0, requires_grad=True, device=devic
e)
            params = p0.clone().detach().requires_grad_(True).to(devic
e)



            # Create LBFGS optimizer
            optimizer = LBFGS([params])

#             # Step through the optimizer
#             optimizer.step(closure)

            # Use closure for (sx, sy)
            optimizer.step(lambda: closure(optimizer, sx, sy, params))
#             optimizer.step(lambda: closure(optimizer, sx, sy))

            # After optimization, the parameters are updated. You can no
```

```
w access the optimized values:
            optimized_params = params.detach()  # Detach the tensor fro
m the computation graph

            # Extract the optimized parameters (sa, sb, sc, sd, se, sf)
for further use
            sa, sb, sc, sd, se, sf = optimized_params.tolist()  # Conve
rt tensor to list for easy usage

            # Perform any further calculations or use these parameters i
n your code
#             theta_s = torch.arctan2(sb, sa - sc) / 2
            theta_s = torch.arctan2(torch.tensor(sb, device=device), to
rch.tensor(sa - sc, device=device)) / 2




#             params = torch.tensor(p0, requires_grad=True, device=devic
e)
            params = p0.clone().detach().requires_grad_(True).to(devic
e)
            optimizer = LBFGS([params])
            optimizer.step(lambda: closure(optimizer, ax, ay, params))
            optimized_params = params.detach()
            aa, ab, ac, ad, ae, af = optimized_params.tolist()
#             theta_a = torch.arctan2(ab, aa - ac) / 2
            theta_a = torch.arctan2(torch.tensor(ab, device=device), to
rch.tensor(aa - ac, device=device)) / 2




#             params = torch.tensor(p0, requires_grad=True, device=devic
e)
            params = p0.clone().detach().requires_grad_(True).to(devic
e)
            optimizer = LBFGS([params])
            optimizer.step(lambda: closure(optimizer, x_, y_, params))
# xx or xa
            optimized_params = params.detach()
            a,b,c,d,e,f = optimized_params.tolist() # aa bb cc ..
#             theta_ = torch.arctan2(ab, aa - ac) / 2
            theta_ = torch.arctan2(torch.tensor(b, device=device), torc
```

```
h.tensor(a - c, device=device)) / 2
            cos_theta_, sin_theta_ = torch.cos(theta_), torch.sin(theta
_)

##################################


#             arxy = calculate_eccentricity_fx(a, b, c, np.mean(s_), np.
mean(a_), np.mean(dis_), np.mean(o_), np.mean(dir_), theta_) #aspect rat
io

            # if player_data['club'].iloc[0] == offensiveTeamAbbr:
            if player_data['player_side'].iloc[0] == "Offense":
#                 no_transforms[gameId][playId]['offense'][nflId] = {'ec
centricity': eccentricity, 'disp':disp, 'theta': theta_, 'cuv':curvatur
e, 'cuvx': avg_cuvx, 'cuvy': avg_cuvy, 'disc':discriminant, 'major':al,
'minor':bl, 'arxy':arxy, 'ars':ars, 'ara': ara, 'theta_s': theta_s, 'the
ta_a':theta_a, 'xl':xl, 'yl':yl, 'x_mean':x_mean, 'y_mean':y_mean, 'game
id': group_data['meta_data']['gameId'], 'playid': group_data['meta_dat
a']['playId'], 'a'=a,'b'= b,'c'=c,'d'=d,'e'=e,'f'=f, 'sa'=sa, 'sb'=sb,
'sc'=sc, s'sd'=d, 'se'=se, 'sf'sf, 'aa'=aa, 'ab'=ab, 'ac'= ac, 'ad'=ad,
'ae'=ae, 'af'=af}
#                 no_transforms[gameId][playId]['offense'][nflId] =
{'a'=a,'b'= b,'c'=c,'d'=d,'e'=e,'f'=f, 'sa'=sa, 'sb'=sb, 'sc'=sc, s'sd'=
d, 'se'=se, 'sf'sf, 'aa'=aa, 'ab'=ab, 'ac'= ac, 'ad'=ad, 'ae'=ae, 'af'=a
f}
                no_transforms[gameId][playId]['offense'][nflId] = {'a':
a,'b': b,'c': c,'d': d,'e': e,'f': f, 'sa': sa, 'sb': sb, 'sc':sc, 's
d': sd, 'se':se, 'sf': sf, 'aa': aa, 'ab': ab, 'ac': ac, 'ad': ad, 'a
e': ae, 'af': af, 'xl':xl, 'yl':yl, 'x_mean':x_mean, 'y_mean':y_mean,
'theta_':theta_, 'theta_s':theta_s, 'theta_a':theta_a, 'gameid': group_
data['meta_data']['gameId'], 'playid': group_data['meta_data']['playI
d']} #, 'last_event'=last_event}
                # 'disp':disp,

            # elif player_data['club'].iloc[0] == defensiveTeamAbbr:
            elif player_data['player_side'].iloc[0] == "Defense":
#                 no_transforms[gameId][playId]['defense'][nflId] = {'ec
centricity': eccentricity, 'disp':disp, 'theta': theta_, 'cuv':curvatur
e,  'cuvx': avg_cuvx, 'cuvy': avg_cuvy, 'disc':discriminant, 'major':al,
'minor':bl, 'arxy':arxy, 'ars':ars, 'ara': ara, 'theta_s': theta_s, 'the
ta_a':theta_a, 'xl':xl, 'yl':yl, 'x_mean':x_mean, 'y_mean':y_mean, 'game
id': group_data['meta_data']['gameId'], 'playid': group_data['meta_dat
a']['playId']}
```

```
                no_transforms[gameId][playId]['defense'][nflId] = {'a':
a,'b': b,'c': c,'d': d,'e': e,'f': f, 'sa': sa, 'sb': sb, 'sc':sc, 's
d': sd, 'se':se, 'sf': sf, 'aa': aa, 'ab': ab, 'ac': ac, 'ad': ad, 'a
e': ae, 'af': af, 'xl':xl, 'yl':yl, 'x_mean':x_mean, 'y_mean':y_mean,
'theta_':theta_, 'theta_s':theta_s, 'theta_a':theta_a, 'gameid': group_
data['meta_data']['gameId'], 'playid': group_data['meta_data']['playI
d']} #, 'last_event'=last_event}
                # 'disp':disp,

        else:
            play_count_without_snap_event += 1
            # events_without_snap[(gameId, playId)] = player_data['even
t'].unique()  # Store unique events

# Final summary
print(f"Total plays processed: {len(tracking_data_dict_play_subset)}")
print(f"Plays with 'ball_snap' or 'snap_direct' event: {play_count_with
_snap_event}")
print(f"Plays without 'ball_snap' or 'snap_direct' event: {play_count_w
ithout_snap_event}")


# torch.save(no_transforms, 'no_transforms[all].pth')
```

In [ ]:

```
import pandas as pd
import numpy as np
from tqdm import tqdm
from scipy.optimize import minimize


# Objective function to minimize
def objective_fn(params, x, y):
    return torch.sum(conics(params, x, y)**2)


import torch
from torch.optim import LBFGS

# # Define the closure function for optimization


# def closure():
def closure(optimiser, sx ,sy, params ):
    optimizer.zero_grad()
    # Split params into sx and sy (assuming sx and sy are of the same si
ze)


    # Now call the objective function with sx and sy
    loss = objective_fn(params, sx, sy)
    loss.backward()
    return loss



# Define conic equation
def conics(params, x, y):
    a, b, c, d, e, f = params
    return a * x**2 + b * x * y + c * y**2 + d * x + e * y + f



# #hyperbolic only
# def calculate_eccentricity(A, B, C, s, acc, dis, o, dir_) :
def calculate_eccentricity_fx(A, B, C, s, acc, dis, o, dir_, theta_) :
    eig_val1 = 0.5 * (A + C + np.sqrt((A - C)**2 + B**2))
    eig_val2 = 0.5 * (A + C - np.sqrt((A - C)**2 + B**2))
```

```
    a = np.sqrt(1 / np.abs(eig_val1))
    b = np.sqrt(1 / np.abs(eig_val2))

    a, b = sorted([a, b], reverse=True)


    return (a**2/b**2)



# Initial guess for the conic parameters
# p0 = [1, 1, 1, 1, 1, 1]
p0 = torch.tensor([1, 1, 1, 1, 1, 1], dtype=torch.float32, requires_gra
d=True, device=device)


# Initialize the main dictionary to store no transforms
no_transforms = {}

# Summary stats for tracking processing
play_count_with_snap_event = 0
play_count_without_snap_event = 0
events_without_snap = {}

n = 200 #50 #500  # Subset for testing
# tracking_data_dict_play_subset = dict(list(tracking_data_dict_play.ite
ms())[:n])
tracking_data_dict_play_subset = dict(list(tracking_data_dict_play.item
s()))

###NEWLY
tracking_data_dict_play_subset = dict_gp


# Device setup: Use GPU if available
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")


for (gameId, playId), group_data in tqdm(tracking_data_dict_play_subse
t.items()) :

    # Initialize gameId in no_transforms if not already present
    if gameId not in no_transforms:
        no_transforms[gameId] = {}
```

```
    # Initialize playId within the gameId if not already present
    if playId not in no_transforms[gameId]:
        no_transforms[gameId][playId] = {
            'offense': {},
            'defense': {},
        }


    # offensiveTeamAbbr = plays[(plays['gameId'] == gameId) & (plays['pl
ayId'] == playId)].iloc[0]['possessionTeam']
    # defensiveTeamAbbr = plays[(plays['gameId'] == gameId) & (plays['pl
ayId'] == playId)].iloc[0]['defensiveTeam']
    offensiveTeamAbbr = sup_data[(sup_data['game_id'] == gameId) & (sup
_data['play_id'] == playId)].iloc[0]['possession_team']
    defensiveTeamAbbr = sup_data[(sup_data['game_id'] == gameId) & (sup
_data['play_id'] == playId)].iloc[0]['defensive_team']

    # Retrieve tracking data for the current game/play
    # tracking_data = group_data['tracking_data']
    tracking_data = group_data['input']
    tracking_data_op = group_data['output']



    # Process tracking data for each player (nflId)
    for nflId, player_data in tracking_data.groupby('nfl_id'):
        if pd.isna(nflId):  # Skip if it's NaN (ball)
            continue



        snap_event_idx = 30
        # player_data_truncated = player_data

        if pd.notna(snap_event_idx):
            # player_data_truncated = player_data.loc[:snap_event_idx]
            player_data_truncated = player_data
```

```
#             x_ = player_data_truncated['x'].values
            x_ = torch.tensor(player_data_truncated['x'].values, device
=device)
            y_ = torch.tensor(player_data_truncated['y'].values, device
=device)
            s_ = torch.tensor(player_data_truncated['s'].values, device
=device)
            a_ = torch.tensor(player_data_truncated['a'].values, device
=device)
            # dis_ = torch.tensor(player_data_truncated['dis'].values, d
evice=device)
            dir_ = torch.tensor(player_data_truncated['dir'].values, de
vice=device)

            o_ = torch.tensor(player_data_truncated['o'].values, device
=device)




            #
            #optimized version
            # disp = dis_.sum()  # Sum of 'dis' directly on GPU
            xl = x_[-1]        # Last x-coordinate
            yl = y_[-1]        # Last y-coordinate
            x_mean = x_.mean() # Mean of x-coordinates on GPU
            y_mean = y_.mean() # Mean of y-coordinates on GPU







            # print('x_',x_)
            # print('y_',y_)
            # print(len(x_), len(y_))

            p0 = torch.tensor([1, 1, 1, 1, 1, 1], dtype=torch.float32,
requires_grad=True, device=device)
#             params = torch.tensor(p0, requires_grad=True, device=devic
e)
            params = p0.clone().detach().requires_grad_(True).to(devic
e)
            # print(params)
            optimizer = LBFGS([params])
            # print(optimizer)
```

```
            optimizer.step(lambda: closure(optimizer, x_, y_, params))
# xx or xa
            optimized_params = params.detach()
            a,b,c,d,e,f = optimized_params.tolist() # aa bb cc ..
            print(nflId, a,b,c,d,e,f)
#             theta_ = torch.arctan2(ab, aa - ac) / 2
            theta_ = torch.arctan2(torch.tensor(b, device=device), torc
h.tensor(a - c, device=device)) / 2
            cos_theta_, sin_theta_ = torch.cos(theta_), torch.sin(theta
_)

##################################


#             arxy = calculate_eccentricity_fx(a, b, c, np.mean(s_), np.
mean(a_), np.mean(dis_), np.mean(o_), np.mean(dir_), theta_) #aspect rat
io

            # if player_data['club'].iloc[0] == offensiveTeamAbbr:
            if player_data['player_side'].iloc[0] == "Offense":
#                 no_transforms[gameId][playId]['offense'][nflId] = {'ec
centricity': eccentricity, 'disp':disp, 'theta': theta_, 'cuv':curvatur
e, 'cuvx': avg_cuvx, 'cuvy': avg_cuvy, 'disc':discriminant, 'major':al,
'minor':bl, 'arxy':arxy, 'ars':ars, 'ara': ara, 'theta_s': theta_s, 'the
ta_a':theta_a, 'xl':xl, 'yl':yl, 'x_mean':x_mean, 'y_mean':y_mean, 'game
id': group_data['meta_data']['gameId'], 'playid': group_data['meta_dat
a']['playId'], 'a'=a,'b'= b,'c'=c,'d'=d,'e'=e,'f'=f, 'sa'=sa, 'sb'=sb,
'sc'=sc, s'sd'=d, 'se'=se, 'sf'sf, 'aa'=aa, 'ab'=ab, 'ac'= ac, 'ad'=ad,
'ae'=ae, 'af'=af}
#                 no_transforms[gameId][playId]['offense'][nflId] =
{'a'=a,'b'= b,'c'=c,'d'=d,'e'=e,'f'=f, 'sa'=sa, 'sb'=sb, 'sc'=sc, s'sd'=
d, 'se'=se, 'sf'sf, 'aa'=aa, 'ab'=ab, 'ac'= ac, 'ad'=ad, 'ae'=ae, 'af'=a
f}
                no_transforms[gameId][playId]['offense'][nflId] = {'a':
a,'b': b,'c': c,'d': d,'e': e,'f': f, 'sa': sa, 'sb': sb, 'sc':sc, 's
d': sd, 'se':se, 'sf': sf, 'aa': aa, 'ab': ab, 'ac': ac, 'ad': ad, 'a
e': ae, 'af': af, 'xl':xl, 'yl':yl, 'x_mean':x_mean, 'y_mean':y_mean,
'theta_':theta_, 'theta_s':theta_s, 'theta_a':theta_a, 'gameid': group_
data['meta_data']['gameId'], 'playid': group_data['meta_data']['playI
d']} #, 'last_event'=last_event}
                # 'disp':disp,

            # elif player_data['club'].iloc[0] == defensiveTeamAbbr:
            elif player_data['player_side'].iloc[0] == "Defense":
```

```
#                 no_transforms[gameId][playId]['defense'][nflId] = {'ec
centricity': eccentricity, 'disp':disp, 'theta': theta_, 'cuv':curvatur
e,  'cuvx': avg_cuvx, 'cuvy': avg_cuvy, 'disc':discriminant, 'major':al,
'minor':bl, 'arxy':arxy, 'ars':ars, 'ara': ara, 'theta_s': theta_s, 'the
ta_a':theta_a, 'xl':xl, 'yl':yl, 'x_mean':x_mean, 'y_mean':y_mean, 'game
id': group_data['meta_data']['gameId'], 'playid': group_data['meta_dat
a']['playId']}
                no_transforms[gameId][playId]['defense'][nflId] = {'a':
a,'b': b,'c': c,'d': d,'e': e,'f': f, 'sa': sa, 'sb': sb, 'sc':sc, 's
d': sd, 'se':se, 'sf': sf, 'aa': aa, 'ab': ab, 'ac': ac, 'ad': ad, 'a
e': ae, 'af': af, 'xl':xl, 'yl':yl, 'x_mean':x_mean, 'y_mean':y_mean,
'theta_':theta_, 'theta_s':theta_s, 'theta_a':theta_a, 'gameid': group_
data['meta_data']['gameId'], 'playid': group_data['meta_data']['playI
d']} #, 'last_event'=last_event}
                # 'disp':disp,

        else:
            play_count_without_snap_event += 1
            # events_without_snap[(gameId, playId)] = player_data['even
t'].unique()  # Store unique events

# Final summary
print(f"Total plays processed: {len(tracking_data_dict_play_subset)}")
print(f"Plays with 'ball_snap' or 'snap_direct' event: {play_count_with
_snap_event}")
print(f"Plays without 'ball_snap' or 'snap_direct' event: {play_count_w
ithout_snap_event}")


# torch.save(no_transforms, 'no_transforms[all].pth')
```

In [ ]:

```
tracking_data_dict_play_subset = dict_gp

for (gameId, playId), group_data in tqdm(tracking_data_dict_play_subset.items()) :

    tracking_data = group_data['input']

    for nflId, player_data in tracking_data.groupby('nfl_id'):

        player_data_truncated = player_data

        p0 = torch.tensor([1, 1, 1, 1, 1, 1], dtype=torch.float32, requires_grad=True, device=device)



        x_ = torch.tensor(player_data_truncated['x'].values, device=device)
        y_ = torch.tensor(player_data_truncated['y'].values, device=device)

        # x_ = x_t
        # y_ = y_t
        # x_ = torch.tensor(x_t)
        # y_ = torch.tensor(y_t)

        params = p0.clone().detach().requires_grad_(True).to(device)
        optimizer = LBFGS([params])
        optimizer.step(lambda: closure(optimizer, x_, y_, params))   # xx or xa
        optimized_params = params.detach()
        a,b,c,d,e,f = optimized_params.tolist() # aa bb cc ..
        print(nflId, a,b,c,d,e,f)
```

In [ ]:

```
tracking_data_dict_play_subset = dict_gp

for (gameId, playId), group_data in tqdm(tracking_data_dict_play_subset.items()) :

    tracking_data = group_data['input']

    for nflId, player_data in tracking_data.groupby('nfl_id'):

        if nflId == 54481 :

            player_data_truncated = player_data

            p0 = torch.tensor([1, 1, 1, 1, 1, 1], dtype=torch.float32, requires_grad=True, device=device)


            x_ = torch.tensor(player_data_truncated['x'].values, device=device)
            y_ = torch.tensor(player_data_truncated['y'].values, device=device)

            print(x_)
            print(y_)
            print(len(x_))
            # x_ = x_t
            # y_ = y_t
            x_ = torch.tensor(x_t)
            y_ = torch.tensor(y_t)
            print()
            # print(x_==torch.tensor(x_t))
            print(y_==torch.tensor(y_t))
            print(x_)
            print(y_)
            print(len(x_))

            params = p0.clone().detach().requires_grad_(True).to(device)
            optimizer = LBFGS([params])
            optimizer.step(lambda: closure(optimizer, x_, y_, params)) # xx or xa
            optimized_params = params.detach()
```

```
        a,b,c,d,e,f = optimized_params.tolist() # aa bb cc ..
        print(nflId, a,b,c,d,e,f)
```

In [54]:

```
print(device)
```

```
cpu
```

In [ ]:

```
x_t = tracking_data_ip[(tracking_data_ip['game_id'] == 2023120306) & (t
racking_data_ip['play_id'] == 1620) & (tracking_data_ip['nfl_id'] == 54
481)]['x'].to_list()
y_t = tracking_data_ip[(tracking_data_ip['game_id'] == 2023120306) & (t
racking_data_ip['play_id'] == 1620) & (tracking_data_ip['nfl_id'] == 54
481)]['y'].to_list()

tracking_data_dict_play_subset = dict_gp

for (gameId, playId), group_data in tqdm(tracking_data_dict_play_subse
t.items()) :

    tracking_data = group_data['input']

    for nflId, player_data in tracking_data.groupby('nfl_id'):

        if nflId == 54481 :

            player_data_truncated = player_data

            p0 = torch.tensor([1, 1, 1, 1, 1, 1], dtype=torch.float32,
requires_grad=True, device=device)


            x_ = torch.tensor(player_data_truncated['x'].values, device
=device)
            y_ = torch.tensor(player_data_truncated['y'].values, device
=device)


            # x_ = torch.tensor(player_data_truncated['x'].numpy(), devi
ce=device)
            # y_ = torch.tensor(player_data_truncated['y'].numpy(), devi
ce=device)

            # x_ = torch.tensor(list(player_data_truncated['x'].values),
device=device)
            # y_ = torch.tensor(list(player_data_truncated['y'].values),
device=device)

            x_ = torch.tensor(player_data_truncated['x'].to_list(), dev
ice=device)
            y_ = torch.tensor(player_data_truncated['y'].to_list(), dev
```

```
ice=device)

            print(x_)
            print(y_)

            params = p0.clone().detach().requires_grad_(True).to(devic
e)
            optimizer = LBFGS([params])
            optimizer.step(lambda: closure(optimizer, x_, y_, params))
# xx or xa
            optimized_params = params.detach()
            a,b,c,d,e,f = optimized_params.tolist() # aa bb cc ..
            print(nflId, a,b,c,d,e,f)



            print(type(x_t))

            # x_t = pd.Series(x_t)
            # y_t = pd.Series(y_t)
            x_ = torch.tensor(x_t)
            y_ = torch.tensor(y_t)
            x_ = torch.tensor(x_t, device=device)
            y_ = torch.tensor(y_t, device=device)

            print(x_)
            print(y_)

            params = p0.clone().detach().requires_grad_(True).to(devic
e)
            optimizer = LBFGS([params])
            optimizer.step(lambda: closure(optimizer, x_, y_, params))
# xx or xa
            optimized_params = params.detach()
            a,b,c,d,e,f = optimized_params.tolist() # aa bb cc ..
            print(nflId, a,b,c,d,e,f)
```

In [ ]:

```
DTYPE = torch.float32
DEVICE = device  # "cpu" if CPU-only

def to_t(x):
    # list/np/pandas/tensor -> torch tensor on DEVICE with DTYPE
    if isinstance(x, torch.Tensor):
        return x.to(device=DEVICE, dtype=DTYPE)
    return torch.as_tensor(x, device=DEVICE, dtype=DTYPE)

tracking_data_dict_play_subset = dict_gp

for (gameId, playId), group_data in tqdm(tracking_data_dict_play_subse
t.items()):
    tracking_data = group_data['input']

    for nflId, player_data in tracking_data.groupby('nfl_id'):
        if nflId != 54481:
            continue

        p0 = torch.tensor([1,1,1,1,1,1], dtype=DTYPE, device=DEVICE, re
quires_grad=True)

        # --- Fit 1: DataFrame values (force dtype/device) ---
        x_ = to_t(player_data['x'].to_numpy())
        y_ = to_t(player_data['y'].to_numpy())
        print(x_); print(y_)

        params = p0.detach().clone().requires_grad_(True)
        opt = LBFGS([params])

        def closure():
            opt.zero_grad(set_to_none=True)
            loss = objective_fn(params, x_, y_)  # must return scalar t
ensor
            loss.backward()
            return loss

        opt.step(closure)
        a,b,c,d,e,f = params.unbind()
        print(nflId, *(t.item() for t in (a,b,c,d,e,f)))

        # --- Fit 2: lists x_t / y_t (force dtype/device once) ---
        x_ = to_t(x_t)
```

```
        y_ = to_t(y_t)
        print(x_); print(y_)

        params = p0.detach().clone().requires_grad_(True)
        opt = LBFGS([params])

        def closure2():
            opt.zero_grad(set_to_none=True)
            loss = objective_fn(params, x_, y_)
            loss.backward()
            return loss

        opt.step(closure2)
        a,b,c,d,e,f = params.unbind()
        print(nflId, *(t.item() for t in (a,b,c,d,e,f)))
```

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:
```
len(no_transforms)
```

In [57]:
```
no_transforms[2023120306][1620]['offense'][54481]
```

Out[57]:
```
{'a': -0.003792740870267153,
 'b': 0.004493914544582367,
 'c': -0.008404567837715149,
 'd': -0.027560919523239136,
 'e': 0.3873665928840637,
 'f': 0.9618589878082275,
 'sa': 0.3807755708694458,
 'sb': -0.0560823455452919,
 'sc': -0.013180343434214592,
 'sd': -0.0820808932185173,
 'se': -0.01174800843000412,
 'sf': -0.002267630770802498,
 'aa': 1.083538336388301e-06,
 'ab': -7.991233360371552e-06,
 'ac': 2.6366751626483165e-07,
 'ad': -5.041834356234176e-06,
 'ae': -1.1503550467750756e-06,
 'af': 1.3299326383275911e-05,
 'xl': tensor(48.7800, dtype=torch.float64),
 'yl': tensor(21.0300, dtype=torch.float64),
 'x_mean': tensor(51.8264, dtype=torch.float64),
 'y_mean': tensor(28.7071, dtype=torch.float64),
 'theta_': tensor(0.3862),
 'theta_s': tensor(-0.0707),
 'theta_a': tensor(-0.7343),
 'gameid': 2023120306,
 'playid': 1620}
```

In [58]:
```
# optimizer = LBFGS([params], lr=1.0, max_iter=20, line_search_fn='stron
g_wolfe')
```

In [ ]:
```
x_ = torch.tensor(x_t)
y_ = torch.tensor(y_t)
p0 = torch.tensor([1, 1, 1, 1, 1, 1], dtype=torch.float32, requires_gra
d=True, device=device)


params = p0.clone().detach().requires_grad_(True).to(device)
print(params)
optimizer = LBFGS([params])
# optimizer = LBFGS([params], lr=1.0, max_iter=20, line_search_fn='stron
g_wolfe')

print(optimizer)
optimizer.step(lambda: closure(optimizer, x_, y_, params))   # xx or xa

optimized_params = params.detach()
a,b,c,d,e,f = optimized_params.tolist() # aa bb cc ..
print( a,b,c,d,e,f)
```

In [ ]:
```

```

In [ ]:
```

```

In [60]:
```
x_t = tracking_data_ip[(tracking_data_ip['game_id'] == 2023120306) & (t
racking_data_ip['play_id'] == 1620) & (tracking_data_ip['nfl_id'] == 54
481)]['x'].to_list()
y_t = tracking_data_ip[(tracking_data_ip['game_id'] == 2023120306) & (t
racking_data_ip['play_id'] == 1620) & (tracking_data_ip['nfl_id'] == 54
481)]['y'].to_list()

# x_t = tracking_data_ip[(tracking_data_ip['game_id'] == 2023120306) &
(tracking_data_ip['play_id'] == 1620) & (tracking_data_ip['nfl_id'] == 4
1349 )]['x'].to_list()
# y_t = tracking_data_ip[(tracking_data_ip['game_id'] == 2023120306) &
(tracking_data_ip['play_id'] == 1620) & (tracking_data_ip['nfl_id'] == 4
1349)]['y'].to_list()



# x_t = tracking_data_ip[(tracking_data_ip['game_id'] == 2023120306) &
(tracking_data_ip['play_id'] == 1620) & (tracking_data_ip['nfl_id'] == 4
4887)]['x'].to_list()
# y_t = tracking_data_ip[(tracking_data_ip['game_id'] == 2023120306) &
(tracking_data_ip['play_id'] == 1620) & (tracking_data_ip['nfl_id'] == 4
4887)]['y'].to_list()

print(x_t)
print()
print(y_t)
# y_t
# y_t.shape, x_t,shape  #yt xt
len(y_t), len(x_t)
```

```
[54.93, 54.91, 54.86, 54.76, 54.61, 54.41, 54.16, 53.87, 53.54, 53.
19, 52.83, 52.47, 52.11, 51.76, 51.42, 51.11, 50.82, 50.55, 50.3, 5
0.07, 49.87, 49.69, 49.52, 49.37, 49.22, 49.08, 48.93, 48.78]

[32.26, 32.26, 32.25, 32.24, 32.21, 32.15, 32.06, 31.95, 31.79, 31.
59, 31.34, 31.03, 30.67, 30.26, 29.8, 29.29, 28.75, 28.17, 27.55, 2
6.9, 26.22, 25.52, 24.79, 24.06, 23.31, 22.55, 21.8, 21.03]
```

Out[60]:
```
(28, 28)
```

In [61]:

```
#       -0.003793       0.004494        -0.008405       -0.027561
0.387367        0.961859

print(' -0.003793       0.004494        -0.008405       -0.027561
0.387367        0.961859')


# dir_ = torch.tensor([0.0,0.0])
dis_ = np.array([0.0,0.0])
s_ = np.array([0.0,0.0])
a_ = np.array([0.0,0.0])
o_ = np.array([0.0,0.0])
dir_ = np.array([0.0,0.0])




# #hyperbolic only
# def calculate_eccentricity(A, B, C, s, acc, dis, o, dir_) :
def calculate_eccentricity_fx(A, B, C, s, acc, dis, o, dir_, theta_) :
    eig_val1 = 0.5 * (A + C + np.sqrt((A - C)**2 + B**2))
    eig_val2 = 0.5 * (A + C - np.sqrt((A - C)**2 + B**2))
    a = np.sqrt(1 / np.abs(eig_val1))
    b = np.sqrt(1 / np.abs(eig_val2))
#     a, b = sorted([a, b], key=abs)
#     a, b = sorted([a, b], key=abs, reverse=True)
#     a, b = sorted([a, b])
    a, b = sorted([a, b], reverse=True)

#         eccentricity = np.sqrt(1 - (b**2 / a**2))
#     eccentricity = np.sqrt(1 + (b**2 / a**2))
#     return np.sqrt(1 + (b**2 / a**2)) #
#     return (b**2 / a**2)
#     return np.log(b**2 / a**2)
#     return (a / b) #asect ratio
    return (a**2/b**2) # not ^
#     return 2*b**2/a  #latus rectum lr
#     return np.abs(theta_)
#     return B**2 - 4 * A * C #dis
#     return (b**2 / a**2) #asect ratio
#     return np.sqrt((b**2 / a**2))
#else return np.nan
```

```
import pandas as pd
import numpy as np
from tqdm import tqdm
from scipy.optimize import minimize


# # Objective function to minimize
# def objective_fn(params, x, y):
#     return np.sum(conics(params, x, y)**2)

# Objective function to minimize
def objective_fn(params, x, y):
    return torch.sum(conics(params, x, y)**2)


import torch
from torch.optim import LBFGS

# # Define the closure function for optimization
# def closure():
#     optimizer.zero_grad()  # Clear previous gradients
#     loss = objective_fn(params, sx, sy)  # Compute loss using objectiv
e_fn
#     loss.backward()  # Backpropagate the gradients
#     return loss


# def closure():
def closure(optimiser, sx ,sy, params ):
    optimizer.zero_grad()
    # Split params into sx and sy (assuming sx and sy are of the same si
ze)
#     sx = params[:len(sx)]  # First part of params for sx
#     sy = params[len(sx):]  # Second part of params for sy

    # Now call the objective function with sx and sy
    loss = objective_fn(params, sx, sy)
    loss.backward()
    return loss
```

```
# Define conic equation
def conics(params, x, y):
    a, b, c, d, e, f = params
    return a * x**2 + b * x * y + c * y**2 + d * x + e * y + f

# # Objective function to minimize
# def objective_fn(params, x, y):
#     return np.sum(conics(params, x



# Initial guess for the conic parameters
# p0 = [1, 1, 1, 1, 1, 1]
p0 = torch.tensor([1, 1, 1, 1, 1, 1], dtype=torch.float32, requires_gra
d=True, device=device)



# x_ = x_t
# y_ = y_t
x_ = torch.tensor(x_t)
y_ = torch.tensor(y_t)

params = p0.clone().detach().requires_grad_(True).to(device)
optimizer = LBFGS([params])
optimizer.step(lambda: closure(optimizer, x_, y_, params))   # xx or xa
optimized_params = params.detach()
a,b,c,d,e,f = optimized_params.tolist() # aa bb cc ..
print(a,b,c,d,e,f)
#             theta_ = torch.arctan2(ab, aa - ac) / 2
theta_ = torch.arctan2(torch.tensor(b, device=device), torch.tensor(a -
c, device=device)) / 2
cos_theta_, sin_theta_ = torch.cos(theta_), torch.sin(theta_)
```

```
        -0.003793       0.004494        -0.008405       -0.027561
0.387367        0.961859
0.017734315246343613 -0.16258327662944794 0.1559484302997589 0.7624
068260192871 0.8475053906440735 0.9914431571960449
```

In [ ]:

In [62]:

```
x_t = tracking_data_ip[(tracking_data_ip['game_id'] == 2023111910) & (t
racking_data_ip['play_id'] == 3746) & (tracking_data_ip['nfl_id'] == 54
054)]['x'].to_list()
y_t = tracking_data_ip[(tracking_data_ip['game_id'] == 2023111910) & (t
racking_data_ip['play_id'] == 3746) & (tracking_data_ip['nfl_id'] == 54
054)]['y'].to_list()

# x_t = tracking_data_ip[(tracking_data_ip['game_id'] == 2023120306) &
(tracking_data_ip['play_id'] == 1620) & (tracking_data_ip['nfl_id'] == 4
1349 )]['x'].to_list()
# y_t = tracking_data_ip[(tracking_data_ip['game_id'] == 2023120306) &
(tracking_data_ip['play_id'] == 1620) & (tracking_data_ip['nfl_id'] == 4
1349)]['y'].to_list()



# x_t = tracking_data_ip[(tracking_data_ip['game_id'] == 2023120306) &
(tracking_data_ip['play_id'] == 1620) & (tracking_data_ip['nfl_id'] == 4
4887)]['x'].to_list()
# y_t = tracking_data_ip[(tracking_data_ip['game_id'] == 2023120306) &
(tracking_data_ip['play_id'] == 1620) & (tracking_data_ip['nfl_id'] == 4
4887)]['y'].to_list()

print(x_t)
print()
print(y_t)
# y_t
# y_t.shape, x_t,shape  #yt xt
len(y_t), len(x_t)









#       -0.003793       0.004494        -0.008405       -0.027561
0.387367        0.961859

print(' -0.003793       0.004494        -0.008405       -0.027561
0.387367        0.961859')



# dir_ = torch.tensor([0.0,0.0])
dis_ = np.array([0.0,0.0])
s_ = np.array([0.0,0.0])
```

```
a_ = np.array([0.0,0.0])
o_ = np.array([0.0,0.0])
dir_ = np.array([0.0,0.0])




# #hyperbolic only
# def calculate_eccentricity(A, B, C, s, acc, dis, o, dir_) :
def calculate_eccentricity_fx(A, B, C, s, acc, dis, o, dir_, theta_) :
    eig_val1 = 0.5 * (A + C + np.sqrt((A - C)**2 + B**2))
    eig_val2 = 0.5 * (A + C - np.sqrt((A - C)**2 + B**2))
    a = np.sqrt(1 / np.abs(eig_val1))
    b = np.sqrt(1 / np.abs(eig_val2))
#     a, b = sorted([a, b], key=abs)
#     a, b = sorted([a, b], key=abs, reverse=True)
#     a, b = sorted([a, b])
    a, b = sorted([a, b], reverse=True)

#         eccentricity = np.sqrt(1 - (b**2 / a**2))
#     eccentricity = np.sqrt(1 + (b**2 / a**2))
#     return np.sqrt(1 + (b**2 / a**2)) #
#     return (b**2 / a**2)
#     return np.log(b**2 / a**2)
#     return (a / b) #asect ratio
    return (a**2/b**2) # not ^
#     return 2*b**2/a  #latus rectum lr
#     return np.abs(theta_)
#     return B**2 - 4 * A * C #dis
#     return (b**2 / a**2) #asect ratio
#     return np.sqrt((b**2 / a**2))
#else return np.nan



import pandas as pd
import numpy as np
from tqdm import tqdm
from scipy.optimize import minimize


# # Objective function to minimize
# def objective_fn(params, x, y):
#     return np.sum(conics(params, x, y)**2)
```

```
# Objective function to minimize
def objective_fn(params, x, y):
    return torch.sum(conics(params, x, y)**2)


import torch
from torch.optim import LBFGS

# # Define the closure function for optimization
# def closure():
#     optimizer.zero_grad()  # Clear previous gradients
#     loss = objective_fn(params, sx, sy)  # Compute loss using objectiv
e_fn
#     loss.backward()  # Backpropagate the gradients
#     return loss


# def closure():
def closure(optimiser, sx ,sy, params ):
    optimizer.zero_grad()
    # Split params into sx and sy (assuming sx and sy are of the same si
ze)
#     sx = params[:len(sx)]  # First part of params for sx
#     sy = params[len(sx):]  # Second part of params for sy

    # Now call the objective function with sx and sy
    loss = objective_fn(params, sx, sy)
    loss.backward()
    return loss



# Define conic equation
def conics(params, x, y):
    a, b, c, d, e, f = params
    return a * x**2 + b * x * y + c * y**2 + d * x + e * y + f

# # Objective function to minimize
# def objective_fn(params, x, y):
#     return np.sum(conics(params, x
```

```
# Initial guess for the conic parameters
# p0 = [1, 1, 1, 1, 1, 1]
p0 = torch.tensor([1, 1, 1, 1, 1, 1], dtype=torch.float32, requires_gra
d=True, device=device)



# x_ = x_t
# y_ = y_t
x_ = torch.tensor(x_t)
y_ = torch.tensor(y_t)

params = p0.clone().detach().requires_grad_(True).to(device)
optimizer = LBFGS([params])
optimizer.step(lambda: closure(optimizer, x_, y_, params))   # xx or xa
optimized_params = params.detach()
a,b,c,d,e,f = optimized_params.tolist() # aa bb cc ..
print(a,b,c,d,e,f)
#             theta_ = torch.arctan2(ab, aa - ac) / 2
theta_ = torch.arctan2(torch.tensor(b, device=device), torch.tensor(a -
c, device=device)) / 2
cos_theta_, sin_theta_ = torch.cos(theta_), torch.sin(theta_)
```

```
[59.55, 59.55, 59.55, 59.56, 59.57, 59.6, 59.63, 59.68, 59.74, 59.
8, 59.86, 59.92, 59.97, 59.99, 59.98, 59.95, 59.88, 59.78, 59.64, 5
9.49, 59.32, 59.13, 58.94]

[24.24, 24.24, 24.26, 24.28, 24.31, 24.38, 24.5, 24.71, 24.97, 25.2
9, 25.68, 26.11, 26.59, 27.1, 27.64, 28.2, 28.78, 29.39, 30.02, 30.
65, 31.3, 31.96, 32.64]
        -0.003793       0.004494        -0.008405       -0.027561
0.387367        0.961859
-0.027767855674028397 0.033033911138772964 -0.053708042949438095 0.
968994677066803 0.9769294261932373 0.9992486238479614
```

In [63]:
```
# calculate_eccentricity_fx(
#             -0.027767855674028397, 0.033033911138772964,  -0.053708042
949438095,
#             0.0, 0.0, 0.0, 0.0, 0.0,
#             0.0
#         )



calculate_eccentricity_fx(
            a, b, c,
            0.0, 0.0, 0.0, 0.0, 0.0,
            0.0
        )
```

Out[63]:
```
3.128045700576517
```

In [ ]:
```

```

In [ ]:
```

```

In [ ]:
```

```

In [64]:
```
len(tracking_data_dict_play)
```

Out[64]:
```
18009
```

In [66]:

```
# Initial guess for the conic parameters
# p0 = [1, 1, 1, 1, 1, 1]
p0 = torch.tensor([1, 1, 1, 1, 1, 1], dtype=torch.float32, requires_gra
d=True, device=device)


# Initialize the main dictionary to store no transforms
no_transforms = {}

# Summary stats for tracking processing
play_count_with_snap_event = 0
play_count_without_snap_event = 0
events_without_snap = {}

n = 200 #50 #500  # Subset for testing
# tracking_data_dict_play_subset = dict(list(tracking_data_dict_play.ite
ms())[:n])
tracking_data_dict_play_subset = dict(list(tracking_data_dict_play.item
s()))


# Device setup: Use GPU if available
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")


# for (gameId, playId), group_data in tqdm(tracking_data_dict_play_subse
t.items()) :
items = list(tracking_data_dict_play_subset.items())
# for (gameId, playId), group_data in tqdm(items[13000:], total=len(item
s)-14000, initial=14000):
z = 14108

# for (gameId, playId), group_data in tqdm(items[z:], total=len(items)-
z, initial=z):
for (gameId, playId), group_data in tqdm(items[z:]): #, total=len(item
s)-z, initial=z):

# for (gameId, playId), group_data in tqdm(items[14200:], total=len(item
s)-14200, initial=14200):

    # print(group_data['nfl_id'])
    # print(tracking_data['nfl_id'])
    # process
```

```
    ...


    # Initialize gameId in no_transforms if not already present
    if gameId not in no_transforms:
        no_transforms[gameId] = {}

    # Initialize playId within the gameId if not already present
    if playId not in no_transforms[gameId]:
        no_transforms[gameId][playId] = {
            'offense': {},
            'defense': {},
        }


    # offensiveTeamAbbr = plays[(plays['gameId'] == gameId) & (plays['pl
ayId'] == playId)].iloc[0]['possessionTeam']
    # defensiveTeamAbbr = plays[(plays['gameId'] == gameId) & (plays['pl
ayId'] == playId)].iloc[0]['defensiveTeam']
    offensiveTeamAbbr = sup_data[(sup_data['game_id'] == gameId) & (sup
_data['play_id'] == playId)].iloc[0]['possession_team']
    defensiveTeamAbbr = sup_data[(sup_data['game_id'] == gameId) & (sup
_data['play_id'] == playId)].iloc[0]['defensive_team']

    # Retrieve tracking data for the current game/play
    # tracking_data = group_data['tracking_data']
    tracking_data = group_data['input']
    tracking_data_op = group_data['output']


    # print(tracking_data, tracking_data_op)
    # # # Process tracking data for each player (nflId)
    # for nflId, player_data in tracking_data.groupby('nfl_id'):
    #     if pd.isna(nflId):  # Skip if it's NaN (ball)
    #         continue


    #     snap_event_idx = 30
    #     # player_data_truncated = player_data

    #     if pd.notna(snap_event_idx):
    #         # player_data_truncated = player_data.loc[:snap_event_idx]
    #         player_data_truncated = player_data
```

```
# Final summary
print(f"Total plays processed: {len(tracking_data_dict_play_subset)}")
print(f"Plays with 'ball_snap' or 'snap_direct' event: {play_count_with
_snap_event}")
print(f"Plays without 'ball_snap' or 'snap_direct' event: {play_count_w
ithout_snap_event}")




# None None
# None None
# None None
# None None
# None None
# None None
```

```
100%|██████████| 3901/3901 [00:05<00:00, 754.54it/s]


Total plays processed: 18009
Plays with 'ball_snap' or 'snap_direct' event: 0
Plays without 'ball_snap' or 'snap_direct' event: 0
```

In [69]:

```
items = list(tracking_data_dict_play_subset.items())
# row_key, row_value = items[14000]
# row_key, row_value = items[14108]
row_key, row_value = items[14107]

# print("GameId, PlayId:", row_key)
# print("Group Data:", row_value)
```

In [ ]:

```

```

In [ ]:

```
len(no_transforms)
```

In [ ]:
```
# torch.save(no_transforms, 'no_transforms[all].pth')
```

In [75]:
```
no_transforms = torch.load("no_transforms[all].pth", weights_only=Fals
e)
```

In [76]:
```
type(no_transforms)
```

Out[76]:
```
dict
```

In [77]:
```
len(no_transforms)
```

Out[77]:
```
273
```

In [ ]:
```

```

In [78]:
```
no_transforms[2023120306][1620]['offense'][54481]
```

Out[78]:
```
{'a': -0.003792740870267153,
 'b': 0.004493914544582367,
 'c': -0.008404567837715149,
 'd': -0.027560919523239136,
 'e': 0.3873665928840637,
 'f': 0.9618589878082275,
 'sa': 0.11079810559749603,
 'sb': 0.17107662558555603,
 'sc': 0.11496926844120026,
 'sd': 0.3724491596221924,
 'se': 1.1378397941589355,
 'sf': 0.35767635703086853,
 'aa': 0.057990431785583496,
 'ab': 0.08438724279403687,
 'ac': 0.1850249022245407,
 'ad': 0.3267730474472046,
 'ae': 0.702515184879303,
 'af': 0.5347175002098083,
 'xl': tensor(48.7800, dtype=torch.float64),
 'yl': tensor(21.0300, dtype=torch.float64),
 'x_mean': tensor(51.8264, dtype=torch.float64),
 'y_mean': tensor(28.7071, dtype=torch.float64),
 'theta_': tensor(0.3862),
 'theta_s': tensor(0.7976),
 'theta_a': tensor(1.2776),
 'gameid': 2023120306,
 'playid': 1620}
```

In [ ]:

```
data_list = []

for gameId in tqdm(no_transforms, desc="Games", unit="game"):
#     print('YG')
# for gameId in no_transforms:
    for playId in no_transforms[gameId]: #2022101607 1127

        try :

            route_ran = sup_data.loc[(sup_data['game_id'] == gameId) &
(sup_data['play_id'] == playId), 'route_of_targeted_receiver'].values
[0]

            pass_result = sup_data.loc[(sup_data['game_id'] == gameId)
& (sup_data['play_id'] == playId) , 'pass_result'].values[0]

            pass_length = sup_data.loc[(sup_data['game_id'] == gameId)
& (sup_data['play_id'] == playId) , 'pass_length'].values[0]

            offense_formation = sup_data.loc[(sup_data['game_id'] == ga
meId) & (sup_data['play_id'] == playId) , 'offense_formation'].values
[0]

            receiver_alignment = sup_data.loc[(sup_data['game_id'] == g
ameId) & (sup_data['play_id'] == playId) , 'receiver_alignment'].values
[0]

            play_action = sup_data.loc[(sup_data['game_id'] == gameId)
& (sup_data['play_id'] == playId) , 'play_action'].values[0]

            dropback_type = sup_data.loc[(sup_data['game_id'] == gameI
d) & (sup_data['play_id'] == playId) , 'dropback_type'].values[0]

            dropback_distance = sup_data.loc[(sup_data['game_id'] == ga
meId) & (sup_data['play_id'] == playId) , 'dropback_distance'].values
[0]

            pass_location_type = sup_data.loc[(sup_data['game_id'] == g
ameId) & (sup_data['play_id'] == playId) , 'pass_location_type'].values
[0]

            defenders_in_the_box = sup_data.loc[(sup_data['game_id'] ==
gameId) & (sup_data['play_id'] == playId) , 'defenders_in_the_box'].val
```

```
ues[0]

            team_coverage_man_zone = sup_data.loc[(sup_data['game_id']
== gameId) & (sup_data['play_id'] == playId) , 'team_coverage_man_zon
e'].values[0]

            team_coverage_type = sup_data.loc[(sup_data['game_id'] == g
ameId) & (sup_data['play_id'] == playId) , 'team_coverage_type'].values
[0]

            penalty_yards = sup_data.loc[(sup_data['game_id'] == gameI
d) & (sup_data['play_id'] == playId) , 'penalty_yards'].values[0]

            pre_penalty_yards_gained = sup_data.loc[(sup_data['game_i
d'] == gameId) & (sup_data['play_id'] == playId) , 'pre_penalty_yards_g
ained'].values[0]

            yards_gained = sup_data.loc[(sup_data['game_id'] == gameId)
& (sup_data['play_id'] == playId) , 'yards_gained'].values[0]

            w = sup_data.loc[sup_data['game_id'] == gameId, 'week'].val
ues[0]#,

            q = sup_data.loc[(sup_data['game_id'] == gameId) &
                                        (sup_data['play_id'] == playI
d), 'quarter'].values[0]#,

            d = sup_data.loc[(sup_data['game_id'] == gameId) &
                                        (sup_data['play_id'] == playI
d), 'down'].values[0]

            possession_team = sup_data.loc[(sup_data['game_id'] == game
Id) &
                                        (sup_data['play_id'] == playI
d), 'possession_team'].values[0]#,

            defensive_team = sup_data.loc[(sup_data['game_id'] == gameI
d) &
                                        (sup_data['play_id'] == playI
d), 'defensive_team'].values[0]

            home_team = sup_data.loc[(sup_data['game_id'] == gameId) &
                                        (sup_data['play_id'] == playI
d), 'home_team_abbr'].values[0]#,
```

```
            visitor_team = sup_data.loc[(sup_data['game_id'] == gameId)
&
                                        (sup_data['play_id'] == playI
d), 'visitor_team_abbr'].values[0]









        except IndexError:
            print('x', gameId, playId)
            route_ran = None  # In case routeRan is missing

            hadRushAttempt_ = None
            rushingYards_ = None

            pos = None

            w = None
            q = None
            # d = None
            # arxy = None
            # ars = None
            # ara = None




#         print('YP')
#         for team_type in ['home', 'away', 'offense', 'defense']:
        for team_type in ['offense', 'defense']:
#             print('OD')
            for nflId, player_data in no_transforms[gameId][playId][tea
m_type].items():
#                 print(nflId)
#                 if 'eccentricity' in player_data:
                # print(player_data)

                try:
                    # route_ran = sup_data.loc[(player_play['gameId'] ==
gameId) &
                    #                          (player_play['playId']
```

```
== playId) &
                    #                                (player_play['nflId']
== nflId), 'route_of_targeted_receiver'].values[0]

                    # hadRushAttempt_ = player_play.loc[(player_play['ga
meId'] == gameId) &
                    #                                (player_play['playId']
== playId) &
                    #                                (player_play['nflId']
== nflId), 'hadRushAttempt'].values[0]
                    # rushingYards_ = player_play.loc[(player_play['game
Id'] == gameId) &
                    #                                (player_play['playId']
== playId) &
                    #                                (player_play['nflId']
== nflId), 'rushingYards'].values[0]

                    pos = tracking_data_ip.loc[(tracking_data_ip['nfl_i
d'] == nflId) & (tracking_data_ip['game_id'] == gameId) & (tracking_dat
a_ip['play_id'] == playId), 'player_position'].values[0] #, # cant spea
k why becomes lal on coment
                    frame = tracking_data_ip.loc[(tracking_data_ip['nfl
_id'] == nflId) & (tracking_data_ip['game_id'] == gameId) & (tracking_d
ata_ip['play_id'] == playId), 'frame_id'].values[0]
                    # print(frame)
                    # pos = tracking_data_ip.loc[(tracking_data_ip['nfl_
id'] == nflId) & (tracking_data_ip['game_id'] == playId) & (tracking_dat
a_ip['play_id'] == playId), 'position'].values[0]
                    player_side = tracking_data_ip.loc[(tracking_data_i
p['nfl_id'] == nflId) & (tracking_data_ip['game_id'] == gameId) & (trac
king_data_ip['play_id'] == playId), 'player_side'].values[0]
                    player_role = tracking_data_ip.loc[(tracking_data_i
p['nfl_id'] == nflId) & (tracking_data_ip['game_id'] == gameId) & (trac
king_data_ip['play_id'] == playId), 'player_role'].values[0]
                    play_direction = tracking_data_ip.loc[(tracking_dat
a_ip['nfl_id'] == nflId) & (tracking_data_ip['game_id'] == gameId) & (t
racking_data_ip['play_id'] == playId), 'play_direction'].values[0]

                    # eccentricity = calculate_eccentricity(a, b, c, tor
ch.mean(s_), torch.mean(a_), torch.mean(dis_), torch.mean(o_), torch.mea
n(dir_), theta_)
#                     eccentricity_s =
#                     eccentricity_a =
```

```
                    #week quarter

                    # w = games.loc[games['gameId'] == gameId, 'week'].v
alues[0]#,

                    # q = plays.loc[(plays['gameId'] == gameId) &
                    #                             (plays['playId'] == pl
ayId), 'quarter'].values[0]#,

                    # d = plays.loc[(plays['gameId'] == gameId) &
                    #                             (plays['playId'] == pl
ayId), 'down'].values[0]
                    # dis_ = np.array([0,0])
                    dis_ = torch.tensor([0.0,0.0])
#                     eccentricity
                    # arxy = calculate_eccentricity_fx(player_data['a'],
player_data['b'], player_data['c'], torch.mean(s_), torch.mean(a_), torc
h.mean(dis_), torch.mean(o_), torch.mean(dir_), theta_)
                    # ars = calculate_eccentricity_fx(player_data['sa'],
player_data['sb'], player_data['sc'], torch.mean(s_), torch.mean(a_), to
rch.mean(dis_), torch.mean(o_), torch.mean(dir_), theta_)
                    # ara = calculate_eccentricity_fx(player_data['aa'],
player_data['ab'], player_data['ac'], torch.mean(s_), torch.mean(a_), to
rch.mean(dis_), torch.mean(o_), torch.mean(dir_), theta_)
                    # arxy = calculate_eccentricity_fx(player_data['a'],
player_data['b'], player_data['c'], torch.mean(s_), torch.mean(a_), torc
h.mean(dis_), torch.mean(o_), torch.mean(dir_), theta_)
                    # ars = calculate_eccentricity_fx(player_data['sa'],
player_data['sb'], player_data['sc'], torch.mean(s_), torch.mean(a_), to
rch.mean(dis_), torch.mean(o_), torch.mean(dir_), theta_)
                    # ara = calculate_eccentricity_fx(player_data['aa'],
player_data['ab'], player_data['ac'], torch.mean(s_), torch.mean(a_), to
rch.mean(dis_), torch.mean(o_), torch.mean(dir_), theta_)
                    arxy = calculate_eccentricity_fx(player_data['a'],
player_data['b'], player_data['c'], torch.mean(s_), torch.mean(a_), tor
ch.mean(dis_), torch.mean(o_), torch.mean(dir_), theta_)
                    ars = calculate_eccentricity_fx(player_data['sa'],
player_data['sb'], player_data['sc'], torch.mean(s_), torch.mean(a_), t
orch.mean(dis_), torch.mean(o_), torch.mean(dir_), theta_)
                    ara = calculate_eccentricity_fx(player_data['aa'],
player_data['ab'], player_data['ac'], torch.mean(s_), torch.mean(a_), t
orch.mean(dis_), torch.mean(o_), torch.mean(dir_), theta_)

                    eccentricity = calculate_eccentricity(player_data
```

```
['a'], player_data['b'], player_data['c'], torch.mean(s_), torch.mean(a
_), torch.mean(dis_), torch.mean(o_), torch.mean(dir_), theta_)


                except IndexError:
                    print('x', gameId, playId)
                    route_ran = None  # In case routeRan is missing

                    hadRushAttempt_ = None
                    rushingYards_ = None

                    pos = None

                    w = None
                    q = None
                    d = None
                    arxy = None
                    ars = None
                    ara = None


                data_list.append({
                    'game_id': gameId,
                    'play_id': playId,
                    'nfl_id': nflId,

                    # 'eccentricity': player_data['eccentricity'],
                    # 'routeRan': route_ran,
                    # 'rushingYards': rushingYards_,

                    # 'hadRushAttempt': hadRushAttempt_,

#                         'dis': player_data['dis'].mean()
                    # 'disp': player_data['disp'], #,
                    'week': w,
                    'quarter': q,
                    'down' : d,
                    'arxy': arxy,
                    'ars': ars,
                    'ara': ara,
                    'pos' : pos,

                    'route_ran' : route_ran,
                    'pass_result' : pass_result,
```

```
                    'pass_length' : pass_length,
                    'offense_formation' : offense_formation,
                    'receiver_alignment' : receiver_alignment,
                    'play_action' : play_action,
                    'dropback_type' : dropback_type,
                    'dropback_distance' : dropback_distance,
                    'pass_location_type' : pass_location_type,
                    'defenders_in_the_box' : defenders_in_the_box,
                    'team_coverage_man_zone' : team_coverage_man_zone,
                    'team_coverage_type' : team_coverage_type,
                    'penalty_yards' : penalty_yards,
                    'pre_penalty_yards_gained' : pre_penalty_yards_gain
ed,
                    'yards_gained' : yards_gained,
                    # 'w' = w,
                    # 'q' = q,
                    # 'd' = d,
                    'possession_team' : possession_team,
                    'defensive_team' : defensive_team,
                    'home_team' : home_team,
                    'visitor_team' : visitor_team,

                    'pos' : pos,
                    'frame' : frame,
                    'player_side' : player_side,
                    'player_role' : player_role,
                    'play_direction' : play_direction,

                    'eccentricity' : eccentricity




#                     'quarter' :
#                     #'theta': player_data['theta']
#                     'theta': np.degrees(player_data['theta']),
#                     'theta_abs': np.abs(np.degrees(player_data['thet
a'])),
#                     'curvature' : player_data['cuv'],
#                     'curvaturex' : player_data['cuvx'],
#                     'curvaturey' : player_data['cuvy'],
#                     'curvaturexabs' : np.abs(player_data['cuvx']),
#                     'discriminant' : player_data['disc'],
```

```
#                     #'rushingYards' : rushingYards_, 'disp':disp, 'wee
k': w, 'quarter':q, 'theta': theta_, 'cuv':curvature, 'cuvx': avg_cuvx,
'cuvy': avg_cuvy, 'disc':discriminant,
#                     'major':player_data['major'], 'minor':player_data
['minor'], 'arxy':player_data['arxy'], 'ars':player_data['ars'], 'ara':
player_data['ara'], 'theta_s': np.degrees(player_data['theta_s']), 'thet
a_a':np.degrees(player_data['theta_a']), 'xl':player_data['xl'], 'yl':pl
ayer_data['yl'], 'x_mean':player_data['x_mean'], 'y_mean':player_data['y
_mean']

#                         'dir'
                })


# Create a DataFrame from the list
eccentricity_df = pd.DataFrame(data_list)
eccentricity_df.head()



# Games:  65%|[redacted]    | 88/136 [40:30<21:52, 27.34s/game]
# 2022101607 1127
# Games: 100%|[redacted]| 136/136 [1:02:39<00:00, 27.64s/game]
```

In [ ]:

```
# torch.save(eccentricity_df, 'eccentricity_df.pth')
```

In [79]:
```
import pandas as pd
import pywt
import numpy as np
import json

from tqdm import tqdm


import torch



pd.set_option('display.max_columns', None)

pd.set_option('display.max_colwidth', None)

pd.set_option('display.max_rows', None)
```

In [80]:
```
# eccentricity_df_loaded = torch.load("eccentricity_df.pth")
# eccentricity_df = torch.load("eccentricity_df.pth")
eccentricity_df = torch.load("eccentricity_df.pth", weights_only=False)
```

In [81]:
```
eccentricity_df.shape
```

Out[81]:
```
(173150, 34)
```

In [82]:
```
eccentricity_df.head()
```

```
/usr/local/lib/python3.11/dist-packages/pandas/io/formats/format.p
y:1458: RuntimeWarning: invalid value encountered in greater
  has_large_values = (abs_vals > 1e6).any()
/usr/local/lib/python3.11/dist-packages/pandas/io/formats/format.p
y:1459: RuntimeWarning: invalid value encountered in less
  has_small_values = ((abs_vals < 10 ** (-self.digits)) & (abs_vals
> 0)).any()
/usr/local/lib/python3.11/dist-packages/pandas/io/formats/format.p
y:1459: RuntimeWarning: invalid value encountered in greater
  has_small_values = ((abs_vals < 10 ** (-self.digits)) & (abs_vals
> 0)).any()
```

Out[82]:

| | game_id | play_id | nfl_id | week | quarter | down | arxy | ars | ara | |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 2023090700 | 101 | 43290 | 1 | 1 | 3 | 4.285046 | 17.689530 | 13.986375 | |
| 1 | 2023090700 | 101 | 44930 | 1 | 1 | 3 | 1272.317907 | 29.623264 | 10.836651 | |
| 2 | 2023090700 | 101 | 53541 | 1 | 1 | 3 | 7.995704 | 16.799922 | 5.577154 | |
| 3 | 2023090700 | 101 | 53959 | 1 | 1 | 3 | 15.961717 | 21.050669 | 1.979233 | |
| 4 | 2023090700 | 101 | 46137 | 1 | 1 | 3 | 8.025947 | 24.178478 | 77.014893 | |

In [ ]:
```
```

In [ ]:
```
```

## Adding features

In [83]:

```
data_list2 = []

for gameId in tqdm(no_transforms, desc="Games", unit="game"):
#     print('YG')
# for gameId in no_transforms:
    for playId in no_transforms[gameId]: #2022101607 1127

        try :

            w = sup_data.loc[sup_data['game_id'] == gameId, 'week'].val
ues[0]#,

            q = sup_data.loc[(sup_data['game_id'] == gameId) &
                                        (sup_data['play_id'] == playI
d), 'quarter'].values[0]#,


        except IndexError:
            print('x', gameId, playId)

            # w = None
            # q = None

        for team_type in ['offense', 'defense']:

            for nflId, player_data in no_transforms[gameId][playId][tea
m_type].items():

                try:

                    # pos = tracking_data_ip.loc[(tracking_data_ip['nfl_
id'] == nflId) & (tracking_data_ip['game_id'] == gameId) & (tracking_dat
a_ip['play_id'] == playId), 'player_position'].values[0] #, # cant speak
why becomes lal on coment

                    # eccentricity = calculate_eccentricity(player_data
['a'], player_data['b'], player_data['c'], torch.mean(s_), torch.mean(a
_), torch.mean(dis_), torch.mean(o_), torch.mean(dir_), theta_)


                    a = player_data['a']
                    b = player_data['b']
                    c = player_data['c']
```

```
                    d = player_data['d']
                    e = player_data['e']
                    f = player_data['f']


                except IndexError:


                    a = None
                    b = None
                    c = None
                    d = None
                    e = None
                    f = None

                data_list2.append({
                    'game_id': gameId,
                    'play_id': playId,
                    'nfl_id': nflId,


                    'a': a,
                    'b': b,
                    'c' : c,
                    'd' : d,
                    'e' : e,
                    'f' : f

                })

# Create a DataFrame from the list
eccentricity_df2 = pd.DataFrame(data_list2)
eccentricity_df2.head()


# torch.save(eccentricity_df2, 'eccentricity_df2.pth')
# eccdf.to_parquet('eccentricity_df.parquet', index=False)   # when you
next save
```

```
Games: 100%|██████████| 273/273 [00:11<00:00, 24.25game/s]
```

Out[83]:

| | game_id | play_id | nfl_id | a | b | c | d | e | f |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 2023090700 | 101 | 43290 | 0.001110 | -0.035468 | -0.030741 | 0.965663 | 0.965806 | 0.998 |
| 1 | 2023090700 | 101 | 44930 | 0.035347 | -0.229525 | 0.376368 | -0.223143 | 0.618138 | 0.945 |
| 2 | 2023090700 | 101 | 53541 | -0.036359 | 0.088478 | -0.173440 | 0.887024 | 0.932098 | 0.995 |
| 3 | 2023090700 | 101 | 53959 | -0.115785 | 0.279340 | -0.223708 | 0.888024 | 0.887742 | 0.994 |
| 4 | 2023090700 | 101 | 46137 | -0.028223 | 0.066825 | -0.148180 | 0.903952 | 0.943100 | 0.996 |

In [84]:
```
eccentricity_df2 = torch.load("eccentricity_df2.pth", weights_only=Fals
e)
```

In [85]:
```
# len(eccentricity_df2)
```

In [86]:
```
# Reload your dataframe
# eccdf = torch.load("eccentricity_df.pth")
eccentricity_dfx = eccentricity_df.copy()
eccentricity_dfy = eccentricity_df2.copy()

# Add week and quarter from sup_data
# sup_add = sup_data[['game_id','play_id','week','quarter']].drop_duplic
ates(['game_id','play_id'])

# eccdf = eccdf.merge(sup_add, on=['game_id','play_id'], how='left')
eccentricity_dfz = eccentricity_dfx.merge(eccentricity_dfy, on=['game_i
d','play_id', 'nfl_id'], how='inner')
```

In [ ]:
```

```

In [87]:
```
torch.save(eccentricity_dfz, 'eccentricity_dfz.pth')
```

In [88]:
```
torch.save(eccentricity_dfz, 'eccentricity_dfz.pth')
eccentricity_dfz = torch.load("eccentricity_dfz.pth", weights_only=Fals
e)
eccentricity_dfz.shape
```

Out[88]:
```
(173150, 40)
```

In [ ]:
```

```

In [ ]:
```

```

In [89]:
```
eccentricity_dfz.shape
```

Out[89]:
```
(173150, 40)
```