multiparticles

March 25, 2025

[2300]:
```python
import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl
from scipy.special import binom, factorial
```

0.0.1 Defining diagrams

Let's start tackling the problem of many types of particles, and modifiying the functions from the 1 type.

To define diagram with n types of particle, we will still use the same points array as the 1 type case, but now there will be n arrays for the paths, so as a example in the case of quarks, antiquarks and gluons a simple diagram could be.

[2301]:
```python
points =  np.array([[0, -1], [0, 1], [1, 0], [2, 0]])
paths_q = np.array([[1, 3]])
paths_a = np.array([[2, 3]])
paths_g = np.array([[3, 4]])
```

Joining paths into 1 array But this way of separating the paths is inconvenient for the function that we define, so instead let's add them in a single array

[2302]:
```python
def combine_paths(*paths):
    max_len = max([len(path) for path in paths])
    final_path = np.zeros((len(paths), max_len, 2), dtype=int)
    for i in range(len(paths)):
        if len(paths[i]) < max_len:
            final_path[i] = np.append(paths[i], np.zeros((max_len -␣
 ↪len(paths[i]), 2), dtype=int), axis=0)
        else:
            final_path[i] = paths[i]
    return final_path

paths = combine_paths(paths_q, paths_a, paths_g)
```

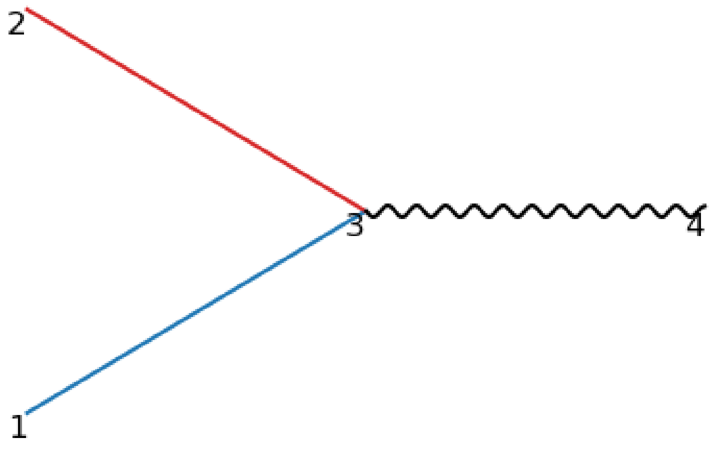
0.0.2 Represent diagram

Similar to the one type case, but now we add a additional loop for all the particle types.

```
[2303]: def find_equal_subarrays(array):
            sorted_subarrays = [np.sort(subarray) for subarray in array]
            unique_subarrays, indices, counts = np.unique(sorted_subarrays, axis=0,␣
         ↪return_index=True, return_counts=True)
            duplicate_positions = [np.where((sorted_subarrays == unique_subarrays[i]).
         ↪all(axis=1))[0] for i in range(len(unique_subarrays)) if counts[i] > 1]
            return duplicate_positions

        def represent_diagram (points, all_paths, index = False, directory = "", colors␣
         ↪= ["tab:blue", "tab:red", "black"], line = ["solid", "solid", "dashed"]):
            fig=plt.figure(figsize=(5,3))
            ax=fig.add_subplot(111)
            ax.axis('off')
            j = 0
            for paths in all_paths:
                loops = find_equal_subarrays(paths)
                for i in range(len(paths)):
                    if (line[j] == "dashed"):
                        with mpl.rc_context({'path.sketch': (3, 15, 1)}):
                            if np.isin(i, loops):
                                middle_point = (points[paths[i, 0]-1] + points[paths[i,␣
         ↪1]-1]) / 2
                                circle = plt.Circle((middle_point[0], middle_point[1]),␣
         ↪np.linalg.norm(points[paths[i, 0]-1]-middle_point), color=colors[j],␣
         ↪fill=False)
                                ax.add_patch(circle)
                            else:
                                ax.plot([points[paths[i, 0]-1, 0], points[paths[i,␣
         ↪1]-1, 0]], [points[paths[i, 0]-1, 1], points[paths[i, 1]-1, 1]],␣
         ↪color=colors[j])
                    else:
                        if np.isin(i, loops):
                            middle_point = (points[paths[i, 0]-1] + points[paths[i,␣
         ↪1]-1]) / 2
                            circle = plt.Circle((middle_point[0], middle_point[1]), np.
         ↪linalg.norm(points[paths[i, 0]-1]-middle_point), color=colors[j],␣
         ↪fill=False, linestyle=line[j])
                            ax.add_patch(circle)
                        else:
                            ax.plot([points[paths[i, 0]-1, 0], points[paths[i, 1]-1,␣
         ↪0]], [points[paths[i, 0]-1, 1], points[paths[i, 1]-1, 1]], color=colors[j],␣
         ↪linestyle=line[j])
                j+=1
            if index:
                for i in range(len(points)):
```

```
            ax.text(points[i, 0], points[i, 1], str(i+1), fontsize=12,␣
↪color="black", ha="right", va="top")
    if directory != "":
        plt.savefig(directory, bbox_inches='tight')
        plt.close()
```

[2304]:
```
represent_diagram(points, paths, index=True)
```



0.0.3 Input/Output particles

Similar to the 1 type case, but with the addition of a new function that return the unique values of the array, needed to identify the input/output particles in `in_out_paths`

[2305]:
```
def unique_values(array):
    unique, counts = np.unique(array, return_counts=True)
    unique_values = unique[counts == 1]
    return unique_values

def trim_zeros_2D(array, axis=1):
    mask = ~(array==0).all(axis=axis)
    inv_mask = mask[::-1]
    start_idx = np.argmax(mask == True)
    end_idx = len(inv_mask) - np.argmax(inv_mask == True)
    if axis:
        return array[start_idx:end_idx,:]
    else:
        return array[:, start_idx:end_idx]

def trim_zeros_3D(array, axis=None):
    if axis is None:
        # Trim along all axes
        mask = ~(array == 0).all(axis=(1, 2))
        trimmed_array = array[mask]

        mask = ~(trimmed_array == 0).all(axis=(0, 2))
        trimmed_array = trimmed_array[:, mask]

        mask = ~(trimmed_array == 0).all(axis=(0, 1))
        trimmed_array = trimmed_array[:, :, mask]
    elif axis == 0:
        # Trim along axis 0
        mask = ~(array == 0).all(axis=(1, 2))
        trimmed_array = array[mask]
    elif axis == 1:
        # Trim along axis 1
        mask = ~(array == 0).all(axis=(0, 2))
        trimmed_array = array[:, mask]
    elif axis == 2:
        # Trim along axis 2
        mask = ~(array == 0).all(axis=(0, 1))
        trimmed_array = array[:, :, mask]
    else:
        raise ValueError("Invalid axis. Axis must be 0, 1, 2, or None.")

    return trimmed_array

def in_out_paths (paths):
    max_len = max([len(path) for path in paths])
    #len(paths) is the number of type of particles
    in_out_paths = np.zeros((len(paths), 2, max_len), dtype=int)
    unique_vals = unique_values(paths.flatten())
    for i in range(len(paths)):
        inp = 0
        out = 0
        for j in range(max_len):
            if paths[i, j, 0] in unique_vals:
                in_out_paths[i, 1, out] = paths[i, j, 0]
                out += 1
            if paths[i, j, 1] in unique_vals:
                in_out_paths[i, 0, inp] = paths[i, j, 1]
                inp += 1
    in_out_paths = trim_zeros_3D(in_out_paths, axis=2)
    return in_out_paths
```

[2306]:
```
print(in_out_paths(paths))
```

```
[[[0]
  [1]]
```

```
 [[0]
  [2]]

 [[4]
  [0]]]
```

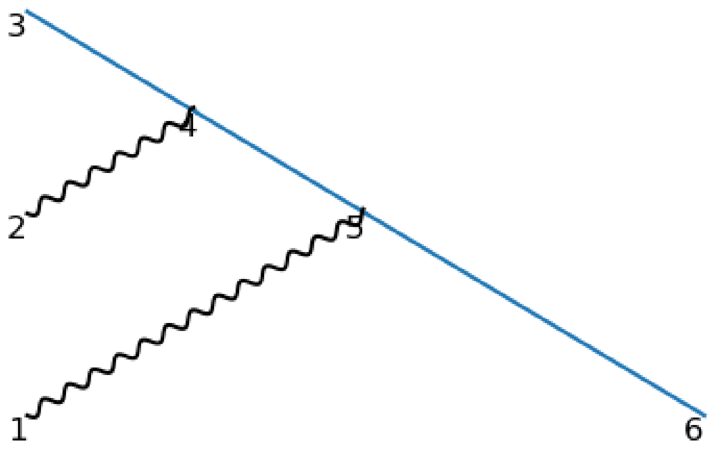
0.0.4 Product of 2 Diagrams (Multiparticle Case)

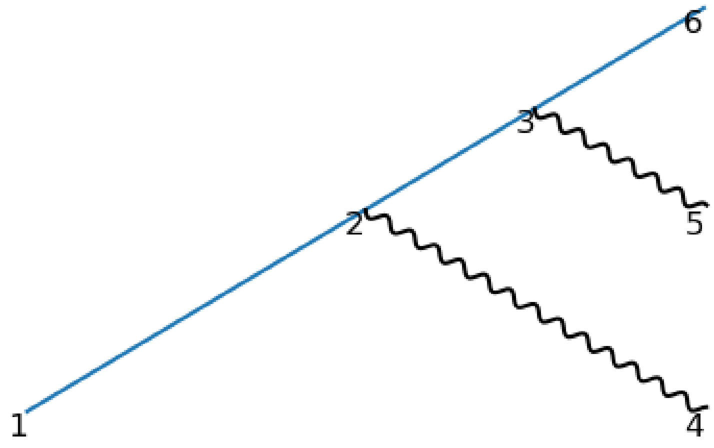
In the case of multiple types of particles, to connect the different points, there will be some differences compared to the 1 type case.

Since now the paths are arrays with one more dimension, this need to be taken account. Let's define some diagrams that make connections possible. (We chose some higher order diagrams to be able to find error in the code)

[2307]:
```
points1 =  np.array([[0, -1], [0, 1], [0, 3], [1, 2], [2, 1],[4, -1]])
paths1_q = np.array([[3, 4], [4, 5], [5, 6]])
paths1_a = np.array([[0, 0]])
paths1_g = np.array([[1, 5], [2, 4]])
paths1 = combine_paths(paths1_q, paths1_a, paths1_g)

points2 =  np.array([[0, -1], [2, 1], [3, 2], [4, -1], [4, 1], [4, 3]])
paths2_q = np.array([[1, 2], [2, 3], [3, 6]])
paths2_a = np.array([[0, 0]])
paths2_g = np.array([[2, 4], [3, 5]])
paths2 = combine_paths(paths2_q, paths2_a, paths2_g)

represent_diagram(points1, paths1, index=True)
represent_diagram(points2, paths2, index=True)

print(in_out_paths(paths1))
```

```
[[[6 0]
  [3 0]]

 [[0 0]
  [0 0]]

 [[0 0]
  [1 2]]]
```





The inputs to the function are, `(points1, paths1, points2, paths2, offset = 0)`. From those we follow the steps, - Calculate the `in_out_paths` for `paths1`and `paths2`. These are used to determine the number of connections, and how they are connected. In this case the `in_out_paths` are arrays of the form. - Index 1: indicate the different types of particles, with `n_types`= `len(in_out_paths1)` = `len(in_out_paths2)`. - Index 2: indicate the if the point is a input or output, with `len(in_out_paths1[0])` indicating the max number of input/output particle, one need to remove 0 to get the number of input/output particle. - Index 3: indicate the i-th input/output particle. - Create the new `points` array, concatenating `points1`, with `points2` and considering a displacement on the horizontal axis equal to the max horizontal value of `points1`, and if `offset`different from 0, add that as a vertical displacement. - Displace the number assign to `in_out_paths2` to match it's new position in `points`

Number of new diagrams Once the framework for the next step is done, we need to consider the different connections, for it we need the following informations, - How many new diagrams are produced. For the case of 1 type of particles we reached the following formula,

$$s_k = \sum_{i_k=1}^{N_k} n_{i_k}, \text{ with, } n_{i_k} = \binom{n_{1_k}}{i}\binom{n_{2_k}}{i} i! \tag{1}$$

n_{1_k} is the number of input particles from diagram 1 of the particle type k, and n_{2_k} is the number of output particles from diagram 2 of the particle type k. N_k is the maximum number of connections possible for the particle type k

$$N_k = min(n_{1_k}, n_{2_k})$$

but now that there are more particles, we need to consider the cases where only one type of particle is connected, 2 types of particles are connected, and so on, this produces more combinations.

This situation is equivalent, to from `n_types` boxes, each with s_k elements, taking any number of elements from 1 to `n_types`, max 1 element per box, how many possible combinations are possible.

$$S = \sum_{r=1}^{n_{\text{types}}} \binom{n_{\text{types}}}{r} \sum_{\substack{I\subseteq\{1,2,\ldots n_{\text{types}}\} \\ |I|=r}} \prod_{k\in I} s_k \tag{2}$$

or equivalent to

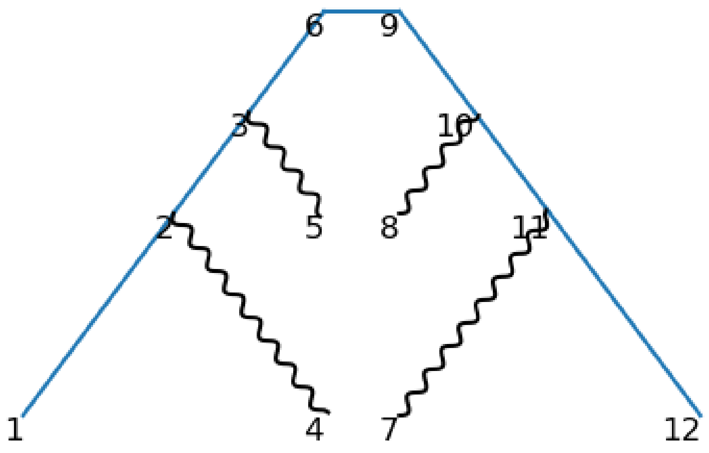
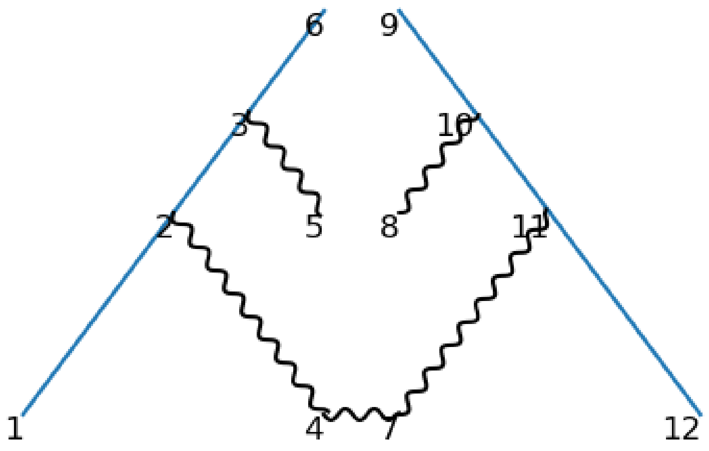
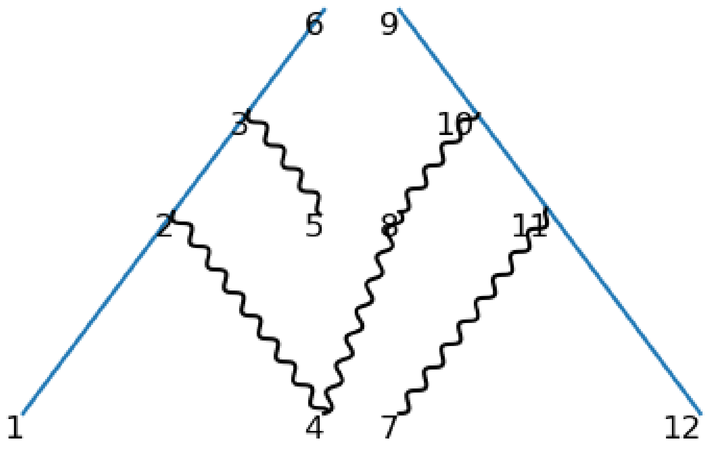
$$S = \sum_{\emptyset\neq I\subseteq\{1,2,\ldots n_{\text{types}}\}} \prod_{k\in I} s_k \tag{3}$$

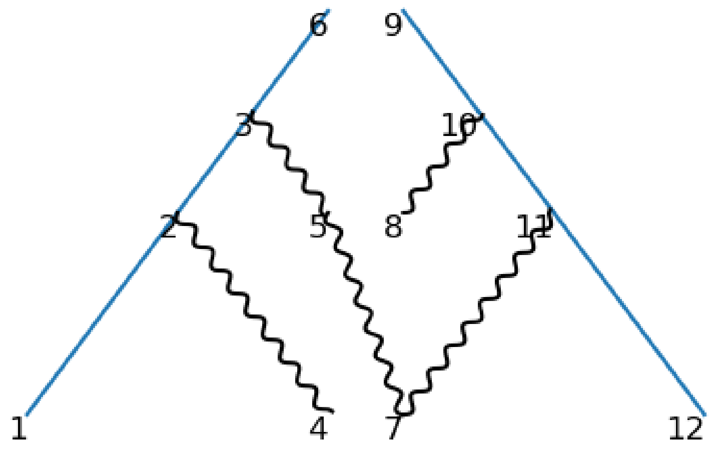
Note: in the code why assign s_k to `n_conenctions` and S to `n_connec`

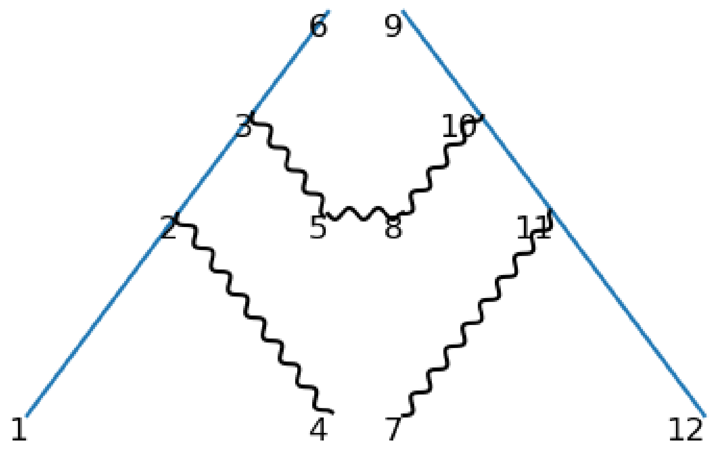
```
[ ]: def how_connected( max_connections, n_connections, n_1, n_2):
         combinations = np.zeros((n_connections, max_connections, 2), dtype=int)
         n = 0
         while n < n_connections:
             for j in range (n_1):
                 for k in range(n_2):
                     combinations[n, 0] = np.array([j+1, k+1])
                     n += 1
                     if n == n_connections:
                         break
                 if n == n_connections:
                     break

         n = n_1*n_2
         if max_connections >1:
             while n < n_connections:
                 diff = False
                 i = 1
```

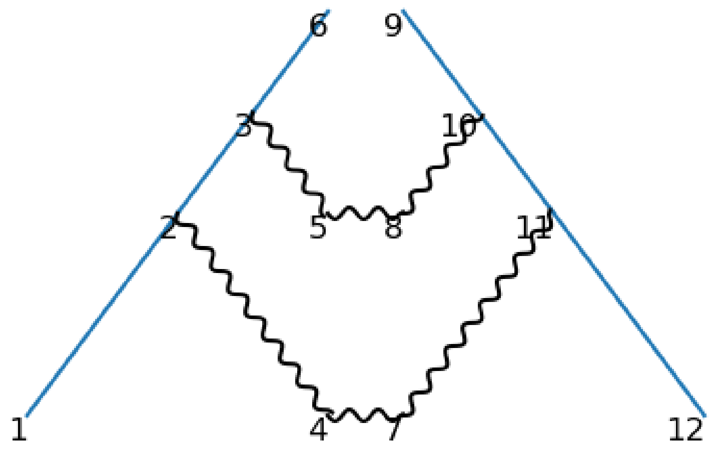
```
            while i < max_connections:
                for j in range (n_1):
                    for k in range(n_2):
                        for l in range(i):
                            if (j+1) != combinations[n, l, 0] and (k+1) !=␣
 ↪combinations[n, l, 1]:
                                diff = True
                            else:
                                diff = False
                        if diff:
                            combinations[n, i] = np.array([j+1, k+1])
                            n +=1
                        if n == n_connections:
                            return combinations
                i += 1
    else:
        return combinations

def connection (points1, paths1, points2, paths2, offset = 0):
    in_out_paths1 = in_out_paths(paths1)
    in_out_paths2 = in_out_paths(paths2)

    n_types = len(in_out_paths1)

    #Create the new points array
    points = np.zeros((len(points1) + len(points2), 2))
    points[:len(points1)] = points1
    points[len(points1):] = points2 + np.array([np.max(points1)+1, offset])

    #Displace the paths of the second diagram to rename the points
    for i in range(n_types):
        for j in range(len(in_out_paths2[0])):
            for k in range(len(in_out_paths2[0, 0])):
                if in_out_paths2[i, j, k] != 0:
                    in_out_paths2[i, j, k] += len(points1)

    #n1 and n2 indicate the number of input for each type of particle and␣
 ↪output for each type of particle
    n1 = np.zeros(n_types, dtype=int)
    n2 = np.zeros(n_types, dtype=int)
    for i in range(n_types):
        n1[i] = len(np.trim_zeros(in_out_paths1[i, 0]))
        n2[i] = len(np.trim_zeros(in_out_paths2[i, 1]))

    #max_connections indicates the maximum number of connections between the␣
 ↪two diagrams for each type of particle
    max_connections = np.zeros(n_types, dtype=int)
    for i in range(n_types):
        max_connections[i] = min(n1[i], n2[i])

    #n_connections indicates the number of connections between the two diagrams␣
↪taking into account the number of types of particles
    n_connections = np.zeros(n_types, dtype=int)
    for j in range(n_types):
        for i in range(int(max_connections[j])):
            n_connections[j] += int(binom(n1[j], i+1)*binom(n2[j], i+1) *␣
↪factorial(i+1))

    #n_connec indicates the total number of connections between the two diagrams
    n_connec = 0
    for subset in range(1, 1 << n_types):
        product = 1
        for i in range(n_types):
            if subset & (1 << i):
                product *= n_connections[i]
        n_connec += product

    dummy_combinations = np.zeros((sum(n_connections), n_types, ␣
↪max(max_connections), 2), dtype=int)
    n = 0
    for i in range(n_types):
        dummy_var = how_connected(max_connections[i], n_connections[i], n1[i],␣
↪n2[i])
        for j in range(n_connections[i]):
            for k in range(max_connections[i]):
                if(dummy_var[j, k, 0] != 0 and dummy_var[j, k, 1] != 0):
                    dummy_combinations[n, i, k] = dummy_var[j, k]
                else:
                    break
            n+=1

    combinations = np.zeros((n_connec, n_types, max(max_connections), 2),␣
↪dtype=int)
    n = 0
    for i in range(n_types):
        if (n_connections[i] == 0):
            continue
        for j in range(np.sum(n_connections[:i]), n_connections[i]+np.
↪sum(n_connections[:i])):
            for k in range(max_connections[i]):
                if(dummy_combinations[j, i, k, 0] != 0 and␣
↪dummy_combinations[j, i, k, 1] != 0):
                    combinations[n, i, k] = dummy_combinations[j, i, k]
                else:
                    break
            n+=1

    n_start = n

    n_prime = 0
    for i in range(n_types-1):
        leng = 0
        for l in range(i+1, n_types):
            leng += n_connections[i]*n_connections[l]
        for n in range(n_start, leng+n_start):
            combinations[n, i] = dummy_combinations[n_prime, i]
            for j in range(i+1, n_types):
                combinations[n, j] =␣
↪dummy_combinations[n-n_start+n_connections[i], j]
        n_prime += 1

    paths = np.zeros((n_connec, n_types, len(paths1) + len(paths2) +␣
↪max(max_connections), 2), dtype=int)
    paths[:n_connec,:n_types,:len(paths1)] = paths1
    for i in range(n_connec):
        for j in range(n_types):
            for k in range(max_connections[j]):
                if (combinations[i,j, k, 0] != 0 and combinations[i,j, k, 1] !=␣
↪0):
                    paths[i,j, len(paths1)+k] = np.array([in_out_paths1[j,0,␣
↪combinations[i, j, k, 0]-1], in_out_paths2[j,1, combinations[i, j, k, 1]-1]])
            if (np.count_nonzero(paths2[j]) != 0):
                paths[i,j, len(paths1)+max(max_connections):] = paths2[j] + np.
↪array([len(points1), len(points1)])

    return points, paths
```

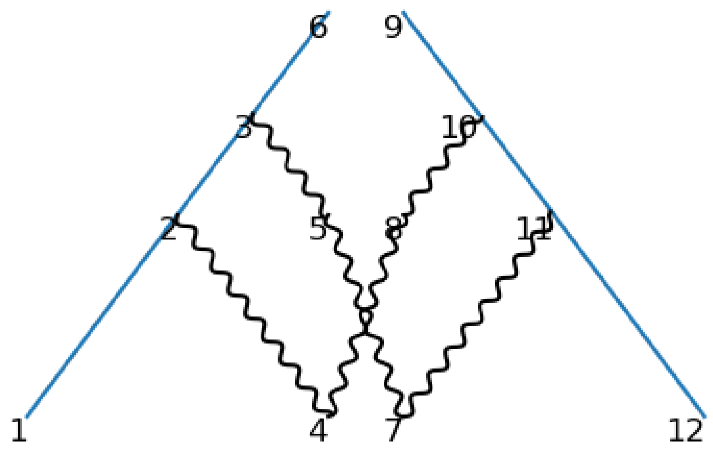
[2310]:
```
points, paths = connection(points2, paths2, points1, paths1)

for i in range(len(paths)):
    represent_diagram(points, paths[i], index=True)
```

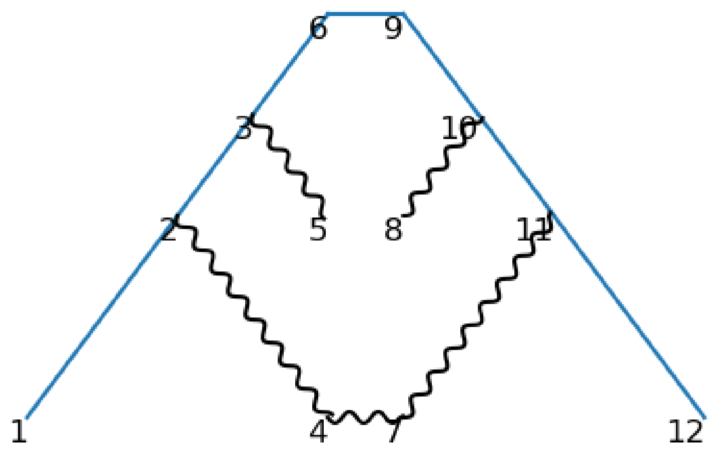

6
9
3
10
2
5
8
11
1
4
7
12

6
9
3
10
2
5
8
11
1
4
7
12

6
9
3
10
2
5
8
11
1
4
7
12

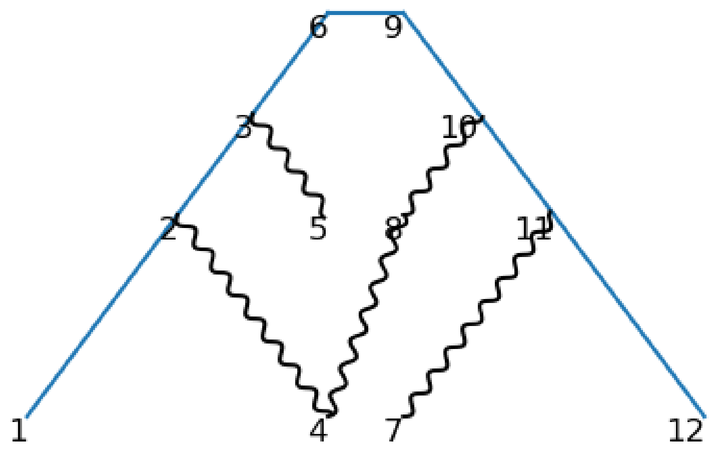

6
9
3
10
2
5
8
11
1
4
7
12

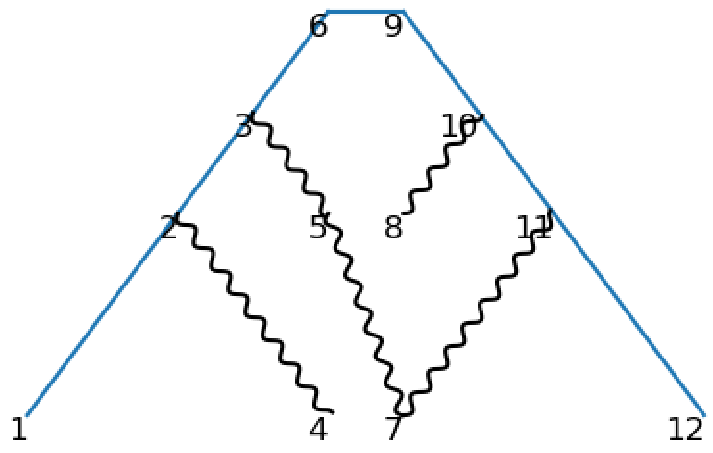

6
9
3
10
2
5
8
11
1
4
7
12

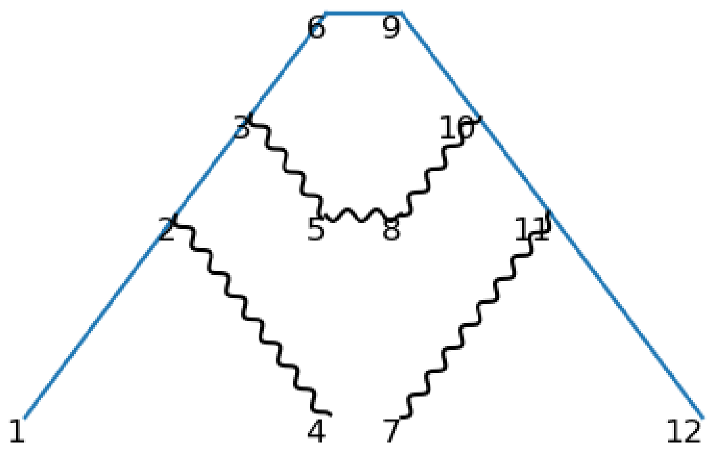

6
9
3
10
2
5
8
11
1
4
7
12

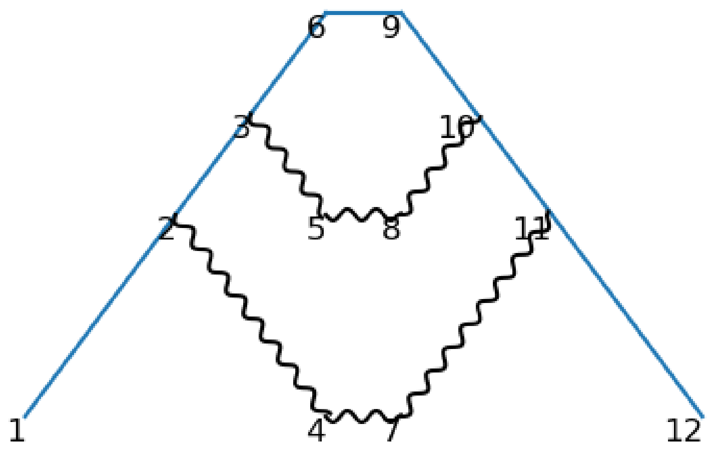

6
9
3
10
2
5
8
11
1
4
7
12

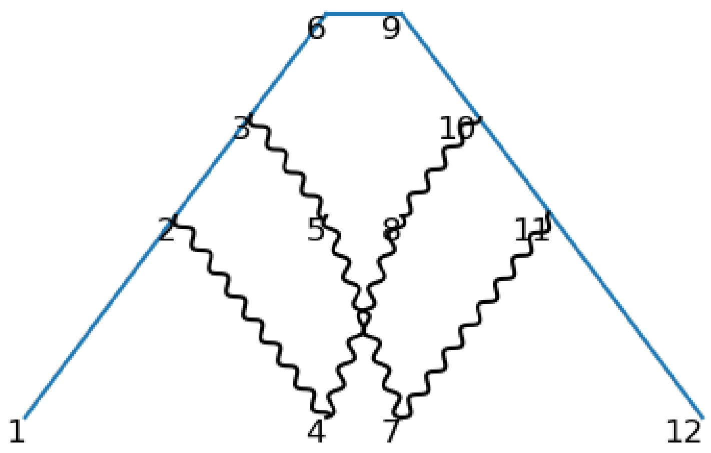

6
9
3
10
2
5
8
11
1
4
7
12


6
9
3
10
2
5
8
11
1
4
7
12


6
9
3
10
2
5
8
11
1
4
7
12


6
9
3
10
2
5
8
11
1
4
7
12


6
9
3
10
2
5
8
11
1
4
7
12


6
9
3
10
2
5
8
11
1
4
7
12