Collaborative Code — DSRI

Python code for representing partitions, Young diagrams, and Maya diagrams. This section is written by Andrew Cameron.

Function takes an array as a partition, and returns its *weight:*

```
def get_weight(array):
    return sum(array)
```

Function takes an array as a partition, and returns its *length:*

```
def get_num_parts(array):
    return len(array)
```

Function takes an index (in range) and array as a partition, and returns the *number of occurrences* of the *value at index i* in the partition:

```
def multiplicity_of_idx(i, array):
    return array.count(array[i])
```

Function takes an array as a partition, and returns its *multiplicity vector representation:*[1]

```
def get_multiplicity_vector(array):
    array.sort(reverse=True)
    lamb = "("

    # Using our above multiplicity function to generate the multiplicity
                                            vector
    for i in range(len(array)):
        if i == 0:
            lamb += str(array[i]) + "^" + str(multiplicity_of_idx(i, array
                                                    ))
        elif array[i] != array[i - 1]:
            if i == len(array) - 1:
                lamb += str(array[i]) + "^" + str(multiplicity_of_idx(i,
                                                        array))
            else:
                lamb += ", " + str(array[i]) + "^" + str(
                                                        multiplicity_of_idx(i
                                                        , array))

    lamb += ")"

    return lamb
```

[1] See p. 1 of MacDonald's *Symmetric Functions and Hall Polynomials* text for a precise definition of this notation.

Function takes an array as a partition, and returns its *Young diagram:*[2]

```
def get_young_diagram(array):
    array.sort(reverse=True)
    yd = ""

    # We use the "#" character to represent a block in the diagram
    for num in array:
        for i in range(num):
            yd += "#"
        yd += "\n"

    return yd
```

Function takes an array as a partition, and returns its *conjugate partition* (equivalent to the *transpose of its Young diagram*):[3]

```
def get_conjugate(array):
    max_value = max(array)
    conjugate = []

    # This determines each part of the conjugate partition
    for i in range(1, max_value + 1):
        count = sum(1 for num in array if num >= i)
        conjugate.append(count)

    return conjugate
```

Function takes $n \in \mathbb{N}$, and recursively returns a list of *all integer partitions for* n:

```
def get_integer_partitions(n):
    full_list = []

    # Using a recursive approach
    def sub_partition(n, k, prefix):
        if n == 0:
            full_list.append(prefix)
            return
        if k > n:
            return
        sub_partition(n, k + 1, prefix)
        sub_partition(n - k, k, prefix + [k])

    sub_partition(n, 1, [])
    return full_list
```

[2]See p. 2 of MacDonald's *Symmetric Functions and Hall Polynomials* text for a precise definition of Young diagrams.

[3]See p. 2 of MacDonald's *Symmetric Functions and Hall Polynomials* text for a precise definition of conjugate partitions.

Function takes a weight ($\in \mathbb{N}$), and *generates a pseudorandom partition of that weight:*

```
from random import randint

def random_part_of_weight(weight):
    remaining_weight = weight
    lamb = []

    # The simple approach to generating a random partition
    while remaining_weight > 0:
        rand = randint(1, remaining_weight)
        remaining_weight -= rand
        lamb.append(rand)

    lamb.sort(reverse=True)
    return lamb
```

Function takes a length ($\in \mathbb{N}$) and maximum value ($\in \mathbb{N}$), and generates a *pseudorandom partition of the specified length with each part less than or equal to the maximum value:*

```
from random import randint

def random_part_of_length(len, max):
    lamb = []

    # Generating random parts for the partition
    for i in range(1, len + 1):
        lamb.append(randint(1, max))

    lamb.sort(reverse=True)
    return lamb
```

Function takes a list of partitions (also a list), and outputs the *relative frequency chart for the lengths of the partitions.* This function is best used in conjunction with the get_integer_partitions(n) function to generate a list of all partitions for a given weight:

```
# Function requires the use of the Matplotlib and NumPy libraries
import matplotlib.pyplot as plt
import numpy as np

def len_rel_freq_chart(partitions):
    # Array to store the various lengths of the partitions
    lengths = [len(lamb) for lamb in partitions]
    unique_lengths, counts = np.unique(lengths, return_counts=True)
    # Adjusting for relative frequency
    relative_freq = counts / len(partitions)
    # matplotlib chart setup
    plt.bar(unique_lengths, relative_freq)
    plt.xlabel("Length of Lambda")
    plt.ylabel("Relative Frequency")
    plt.title("Relative Frequency Chart of Partition Lengths")
    plt.show()
```

Function takes a list of partitions (also a list), and outputs the *relative frequency chart for the multiplicities of each part.* Optionally, one can input a length, such that *only partitions of that length are included in the chart.* This function is best used in conjunction with the get_integer_partitions(n) function to generate a list of all partitions for a given weight:

```
def mult_rel_freq_chart(partitions, length=-1):
    # Redefine the partitions list to only contain partitions of the
                                        specified length (-1 default
                                        value skips this)
    if length != -1:
        partitions = [sublist for sublist in partitions if len(sublist) ==
                                             length]

    # Using a dictionary for the multiplicities (keys -> number, values ->
                                         multiplicity)
    multiplicities = {}
    total_numbers = 0

    # Count the multiplicities of numbers in the lists
    for lst in partitions:
        for num in lst:
            multiplicities[num] = multiplicities.get(num, 0) + 1
            total_numbers += 1

    # Convert the dictionary values to lists
    numbers = list(multiplicities.keys())
    counts = list(multiplicities.values())
    # Adjusting for relative frequency
    relative_freq = np.array(counts) / total_numbers

    # matplotlib chart setup
    plt.bar(numbers, relative_freq)
    plt.xlabel("Number")
    plt.ylabel("Relative Frequency")
    plt.title("Relative Frequency Bar Chart of Number Multiplicities")
    plt.show()
```

Function takes a partition (as an array) and a charge ($\in \mathbb{Z}$), and returns the *Maya diagram for such a partition and charge.*[4] The numerical elements of the array returned is the numerical Maya diagram, and the visual elements appended to the end of the array is the visual Maya diagram:

[4]See Nathan Grieve's paper for more on Maya digrams, partitions, charges, and their connections. https://arxiv.org/pdf/1606.09175.pdf

```python
def maya_diagram(partition, charge):
    partition.sort(reverse=True)
    # Various sets defined in Nathan's paper
    s_plus = []
    s_minus = []
    maya_output = []

    # Numbers to go in s_plus
    for i in range(len(partition)):
        if partition[i] > i:
            s_plus.append(partition[i] - i)
        else:
            break

    # Numbers to go into s_minus
    for i in range(1, len(partition)):
        if sum(num >= i for num in partition) - i >= 1:
            s_minus.append((sum(num >= i for num in partition) - i) * -1)
        else:
            break

    # Numbers to go into s_plus
    for num in s_plus:
        maya_output.append(num)

    # Adding 0 into the final output
    maya_output.append(0)

    # Appending to the final output, plus a few more at the end
    for i in range(-1, min(s_minus) - 3, -1):
        if i not in s_minus:
            maya_output.append(i)

    # Adjusting for the charge, if needed
    if charge != 0:
        for i in range(len(maya_output)):
            maya_output[i] += charge

    # Creating the visual diagram
    visual = ""
    for i in range(max(maya_output), min(maya_output) - 3, -1):
        if i in s_plus or i in s_minus:
            visual += "*"
        else:
            visual += "o"

    # Appending the visual diagram to the final output
    maya_output.append("... Visual: " + visual + "...")

    return maya_output
```

Function takes a numerical Maya diagram (as an array) and a charge ($\in \mathbb{Z}$), and returns the *partition that corresponds to such a Maya diagram and charge.* In a sense, this is an "inverse operation" of the previous function:

```
def partition_from_maya(maya_set, charge):
    # Various sets defined in Nathan's paper
    s_plus = []
    s_minus = []

    # A "helper" set
    s_minus_adjust = []

    # Final output
    partition = []

    # Adjust for a charge, if needed
    if charge != 0:
        for i in range(len(maya_set)):
            maya_set[i] -= charge

    # Numbers to go into s_plus
    for i in range(len(maya_set)):
        if maya_set[i] != 0:
            s_plus.append(maya_set[i])
        else:
            break

    # Adjusting numbers from s_plus to go into the partition
    for i in range(len(s_plus)):
        partition.append(s_plus[i] + i)

    # Numbers to go into s_minus
    for i in range(-1, min(maya_set) - 1, -1):
        if i not in maya_set:
            s_minus.append(i)

    # Numbers to go into the s_minus adjusted set
    for num in s_minus:
        s_minus_adjust.append(-1 * num)

    for i in range(len(s_minus_adjust)):
        s_minus_adjust[i] -= i

    # Based on s_minus_adjusted, we compute the additional numbers that
                                            must be added to the partition
    for i in range(1, max(s_minus_adjust) + 1):
        count = sum(num >= i for num in s_minus_adjust)
        partition.append(count)

    return partition
```