markov chain models of classical music based on the implication-realization model

bomediano_salvador_santuyo

Introduction

project idea

markov chains as a way to model musical patterns and structure based on the **Implication-Realization Model**

The conventional way to use apply Markov chains in the field of music is to create models based on primitive musical elements such as pitch, note, duration, etc., and perhaps use those models to analyze music.

Our group aim to use Markov chains as a way to capture a piece's melodic expectancy, based on Eugene Narmour's Implication-Realization (IR) Model.

Related Literature

project idea

Implication-Realization Model

- Suggests that listeners have innate psychological mechanisms that help them predict future musical events based on the patterns and structures present in the music they hear

- **Implication:** Musical elements or patterns that lead the listener to expect a certain continuation or resolution

- **Realization:** The actual continuation or resolution that when realized, fulfills the listener's expectations, or when unrealized, surprises the listener

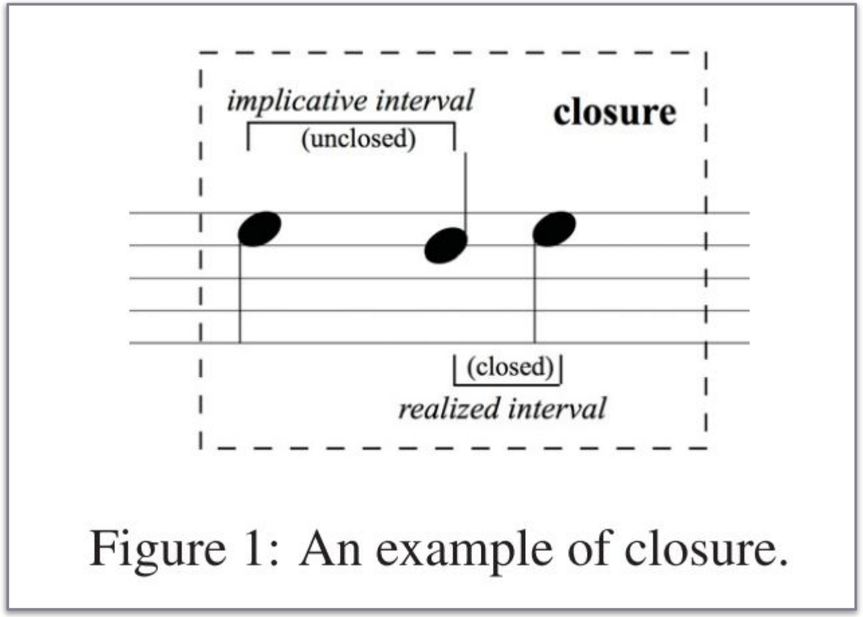


Figure 1: An example of closure.

- Universal formal hypotheses
  - **A + A → A** (repetition implies further repetition)
  - **A + B → C** (contrast implies further contrast)

Implication Realization

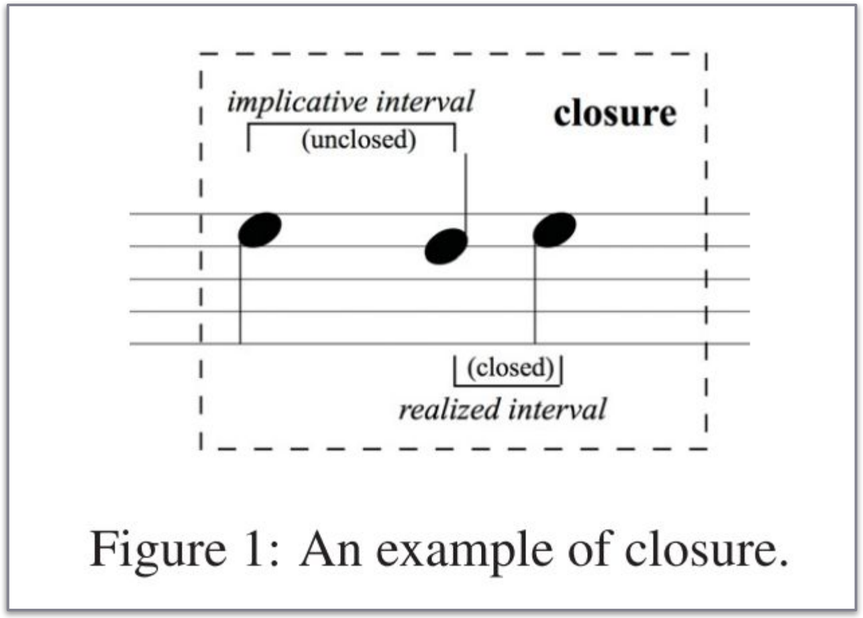


Figure 1: An example of closure.

Structure

- Two main principles
  - Intervallic registral direction
    - The distance between two notes in an interval
  - Registral direction
    - The direction of the pitch within an interval

Melodic Archetypes

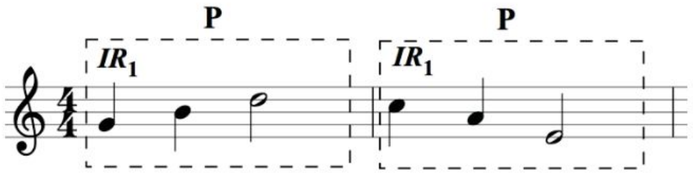


Figure 4: Examples of pattern P.

$$IR_1 : \mathbf{P} \Leftrightarrow \begin{cases} I_k \cdot I_{k+1} > 0 \\ 0 < |I_{k+1} - I_k| < 6 \end{cases}$$

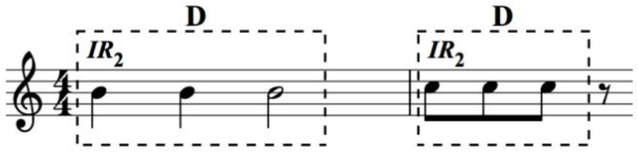


Figure 5: Examples of pattern D.

$$IR_2 : \mathbf{D} \Leftrightarrow I_k = I_{k+1} = 0$$

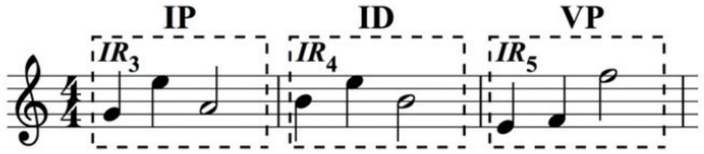


Figure 6: Examples of pattern IP, ID and VP.

$$IR_3 : \mathbf{IP} \Leftrightarrow \begin{cases} I_k \cdot I_{k+1} < 0 \\ -6 < |I_{k+1}| - |I_k| < 6 \\ |I_{k+1}| \neq |I_k| \end{cases}$$

$$IR_4 : \mathbf{ID} \Leftrightarrow \begin{cases} I_k \cdot I_{k+1} < 0 \\ |I_{k+1}| = |I_k| \end{cases}$$

$$IR_5 : \mathbf{VP} \Leftrightarrow \begin{cases} I_k \cdot I_{k+1} > 0 \\ |I_{k+1} - I_k| > 6 \\ |I_k| < 6 \end{cases}$$

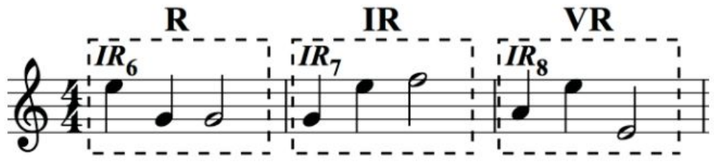


Figure 7: Examples of pattern R, IR and VR.

$$IR_6 : \mathbf{R} \Leftrightarrow \begin{cases} I_k \cdot I_{k+1} < 0 \\ ||I_{k+1}| - |I_k|| > 6 \\ |I_k| > 6 \end{cases}$$

$$IR_7 : \mathbf{IR} \Leftrightarrow \begin{cases} I_k \cdot I_{k+1} > 0 \\ ||I_{k+1}| - |I_k|| > 6 \\ |I_k| > 6 \end{cases}$$

$$IR_8 : \mathbf{VR} \Leftrightarrow \begin{cases} I_k \cdot I_{k+1} < 0 \\ |I_{k+1} - I_k| > 6 \\ |I_k| < 6 \end{cases}$$

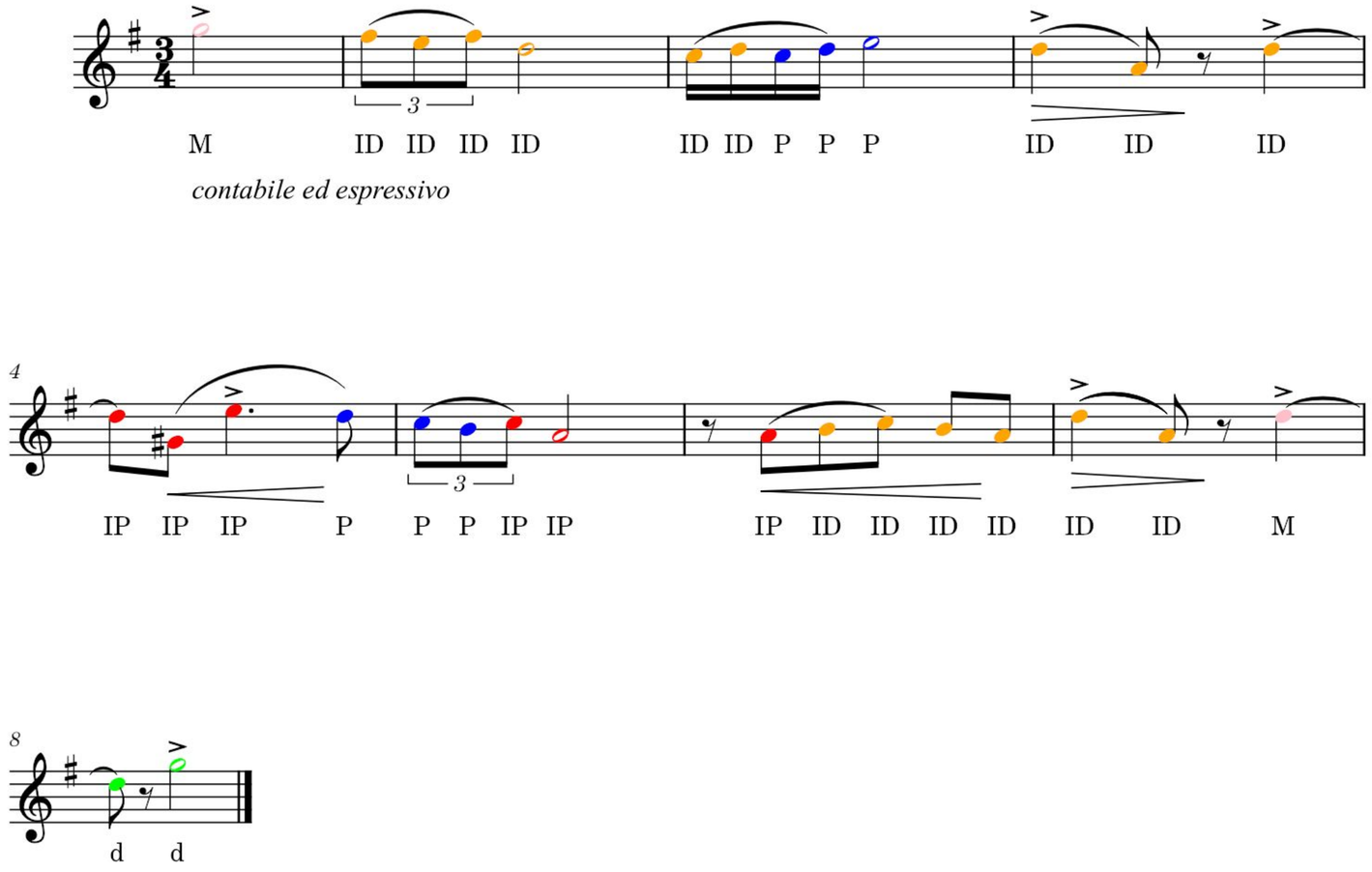
Melodic Archetypes

Example 2. Ranking of melodic structural types from most open to most closed (361)

Structure	Intervallic Motion (I)	Registral Motion (V)
VP	(NCL) mR/AB	(NCL) mR/AA
P	(NCL) mR/AA	(NCL) mR/AA
P, D	(NCL) mN/AA	(NCL) mR/AA, mN/AA
P	(NCL) mL/AA	(NCL) mR/AA
VR	(NCL) mR/AB	(CL) mL/AB
IP	(NCL) mR/AA	(CL) mL/AB
ID	(NCL) mN/AA	(CL) mL/AB
IP	(NCL) mL/AA	(CL) mL/AB
IR	(CL) mL/AB	(NCL) mR/AA
R	(CL) mL/AB	(CL) mL/AB

project
idea

Quartet for 2 Violins, Viola and Violoncello D major 3 Nocturne.xml

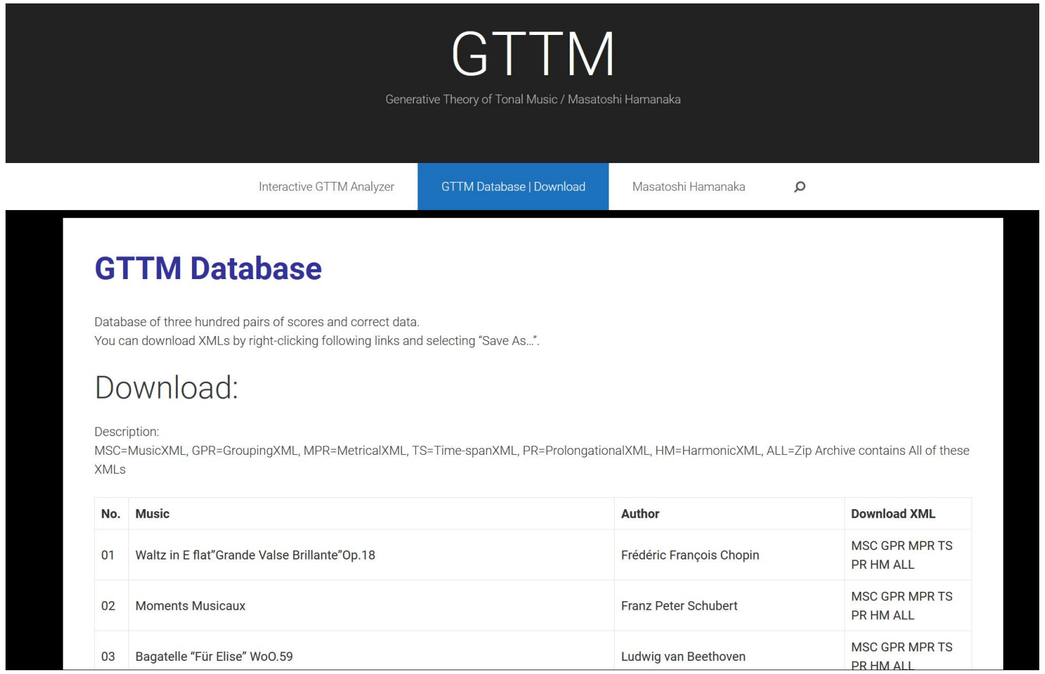


simulation and modelling.

Section III:
Simulation and Modelling

Dataset

- Database used: GTTM Database
  - Contains 300 pairs of scores and correct data
- The group used musical pieces that have composers with at least 3 pieces in the database
- mostly monophonic music



Section III:
Simulation and Modelling

Overview: Using datasets on classical artists' composition, musical data will be extracted and represented according to the Implication-Realization music theory (IR Model), wherein the resulting representations will be in the form of Markov Chain models.

Steps

1. Musical Data Extraction
2. IR Rule Definition
3. IR Symbol Assignment
4. Markov Chain Representation

Section III: Simulation and Modelling
Musical Data Extraction

```
def get_notes(score):
    element_array = []
    for part in score.parts:
        for element in part.flatten():
            element_array.append(element)
    return element_array
```

- Element extraction using music21

Section III: Simulation and Modelling
IR Patterns

```python
def calculate_ir_symbol(interval1, interval2, threshold=5):
    direction = interval1 * interval2
    abs_difference = abs(interval2-interval1)
    # Process
    if direction > 0 and (abs(interval2-interval1))<threshold:
        return 'P'
    # IR2: D (Duplication)
    elif interval1 == interval2 == 0:
        return 'D'
    # IR3: IP (Intervallic Process)
    elif ((interval1 * interval2)<0) and (-threshold <= (abs(interval2) - abs(interval1)) <= threshold) and (abs(interval2) != abs(interval1)):
        return 'IP'
    # IR4: ID (Intervallic Duplication)
    elif ((interval1 * interval2) < 0) and (abs(interval2) == abs(interval1)):
        return 'ID'
    # IR5: VP (Vector Process)
    elif (interval1 * interval2 > 0) and (abs(interval2-interval1) >= threshold) and (abs(interval1) <= threshold):
        return 'VP'
    # IR6: R (Reveral)
    elif (interval1 * interval2 < 0) and (abs(abs(interval2)-abs(interval1)) >= threshold) and (abs(interval1) >= threshold):
        return 'R'
    # IR7: IR (Intervallic Reveral)
    elif (interval1 * interval2 > 0) and (abs(abs(interval2)-abs(interval1)) >= threshold) and (abs(interval1) >= threshold):
        return 'IR'
    # IR8: VR (Vector Reveral)
    elif (interval1 * interval2 < 0) and (abs(interval2 - interval1) >= threshold) and (abs(interval1) <= threshold):
        return 'VR'
```

Section III: Simulation and Modelling
IR Pattern Identification

```
def evaluate_current_group():
    if len(current_group) == 3:
        interval1 = group_pitches[1] - group_pitches[0]
        interval2 = group_pitches[2] - group_pitches[1]
        symbol = calculate_ir_symbol(interval1, interval2)
        # symbols.append(symbol)
        color = color_map.get(symbol, 'black')  # Default to black if symbol is not predefined
        symbols.extend((note, symbol, color) for note in current_group)
    elif len(current_group) == 2:
        # symbols.append('d')  # Dyad
        symbols.extend((note, 'd', color_map['d']) for note in current_group)
    elif len(current_group) == 1:
        # symbols.append('M')  # Monad
        symbols.extend((note, 'M', color_map['M']) for note in current_group)
    else:
        symbols.append('Error: Invalid note object')
    current_group.clear()
    group_pitches.clear()
```

- Note grouping

Section III: Simulation and Modelling
IR Pattern Identification

```
for element in elements:
    if isinstance(element, note.Note):
        current_group.append(element)
        group_pitches.append(element.pitch.ps)
        if len(current_group) == 3:
            evaluate_current_group()
    elif isinstance(element, chord.Chord):
        current_group.append(element)
        group_pitches.append(element.root().ps)
        if len(current_group) == 3:
            evaluate_current_group()
    elif isinstance(element, note.Rest):
        continue
        rest_tuple = (element, 'rest', 'black')
        evaluate_current_group()
        symbols.append(rest_tuple)
    else:
        if current_group:
            evaluate_current_group()
```

- Handling element cases

Section III: Simulation and Modelling
Markov Chain Representation

```
def matrix_calculation(symbols):
    mat_calc = np.zeros((10, 10))
    # Count transitions
    for (current, upcoming) in zip(symbols[:-1], symbols[1:]):
        mat_calc[state_index[current[1]]][state_index[upcoming[1]]] += 1

    # Normalize the transition matrix to get probabilities
    sums = mat_calc.sum(axis=1)
    mat_calc = np.divide(mat_calc, sums[:, None], out=np.zeros_like(mat_calc), where=(sums[:, None] != 0))

    return mat_calc
```

Section III: Simulation and Modelling
Markov Chain Representation

```
for composer in composer_matrices:
    filepath = "./Music Database/GTTM Database/" + composer

    # Initializing the transition_matrix for a specific composer
    composer_matrix = np.zeros((10, 10))
    composer_pieces = []
    aggregated_symbols = []

    piece_count = 0
    for piece in os.listdir(filepath):
        score_path = os.path.join(filepath, piece)  # Get the path of part
        score = converter.parse(score_path)         # Convert the piece ir
        notes = get_notes(score)                    # Extract elements for
        ir_symbols = assign_ir_symbols(notes)       # Assign IR symbols ba
        aggregated_symbols.append(ir_symbols)       # A list for aggregate
```

```
        piece_count += 1

    # Save the list of individual pieces in the dict
    composer_piece_matrices[composer] = composer_pieces

    # Markov chain for aggregated piece
    aggregated_symbols = [item for sublist in aggregated_symbols for item in sublist]
    piece_matrix = matrix_calculation(aggregated_symbols)
    aggregated_piece_matrices[composer] = piece_matrix

    # Average out all the piece matrices
    composer_matrix = composer_matrix/piece_count
    # Normalize the matrices
    for row in composer_matrix:
        row_sum = np.sum(row)
        if 0< row_sum <1:
            for i in range(len(row)):
                row[i] /= row_sum
```

Section III: Simulation and Modelling
Markov Chain Representation

```python
class MarkovChain:
    def __init__(self, transition_matrix, initial_state, state_labels=None):
        # Ensure transition matrix is np array and column stochastic
        transition_matrix = np.array(transition_matrix)
        if not np.allclose(transition_matrix.sum(axis=0), 1):
            transition_matrix = transition_matrix.T
            # raise ValueError("The transition matrix is not column stochastic.")

        self.transition_matrix = transition_matrix

        # Initialize
        initial_state = np.array(initial_state).reshape(-1, 1)
        self.initial_state = initial_state
        self.current_state = initial_state

        # State labels
        if state_labels is None:
            state_labels = list(range(transition_matrix.shape[0]))
        self.state_labels = state_labels
        self.states_history = [initial_state.flatten().tolist()]

    def iterate(self, steps=1):
        for i in range(steps):
            self.current_state = np.dot(self.transition_matrix, self.current_state)
            self.states_history.append(self.current_state.flatten().tolist())
        return self.current_state
```

Section III: Simulation and Modelling
Graph Visualization

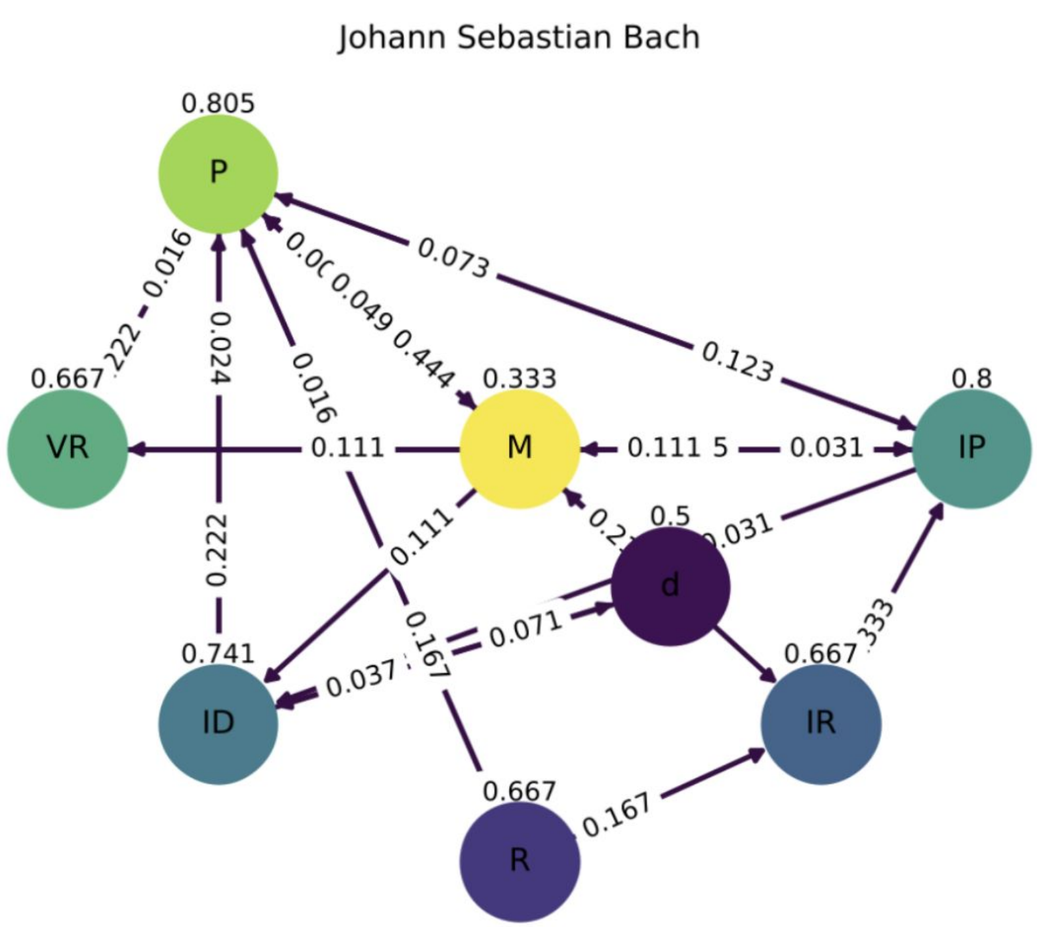
```python
def create_graphs(transition_matrices):
    n = len(transition_matrices)

    # Define node positions (adjust dictionary for your layout)
    node_positions = {
        'VP': (0, 1.5),
        'P': (-1, 1),
        'D': (1, 1),
        'VR': (-1.5, 0),
        'IP': (1.5, 0),
        'ID': (-1, -1),
        'IR': (1, -1),
        'R': (0, -1.5),
        'M': (0, 0),
        'd': (0.5, -0.5),
    }

    # Define node colors (adjust color names for preference)
    node_colors = {
        'VP': '#dce317',
        'P': '#94d840',
        'D': '#56c667',
        'VR': '#29af7f',
        'IP': '#1f968b',
        'ID': '#277d8e',
        'IR': '#32648e',
        'R': '#453781',
        'M': '#fde725',
        'd': '#440d54',
    }
```

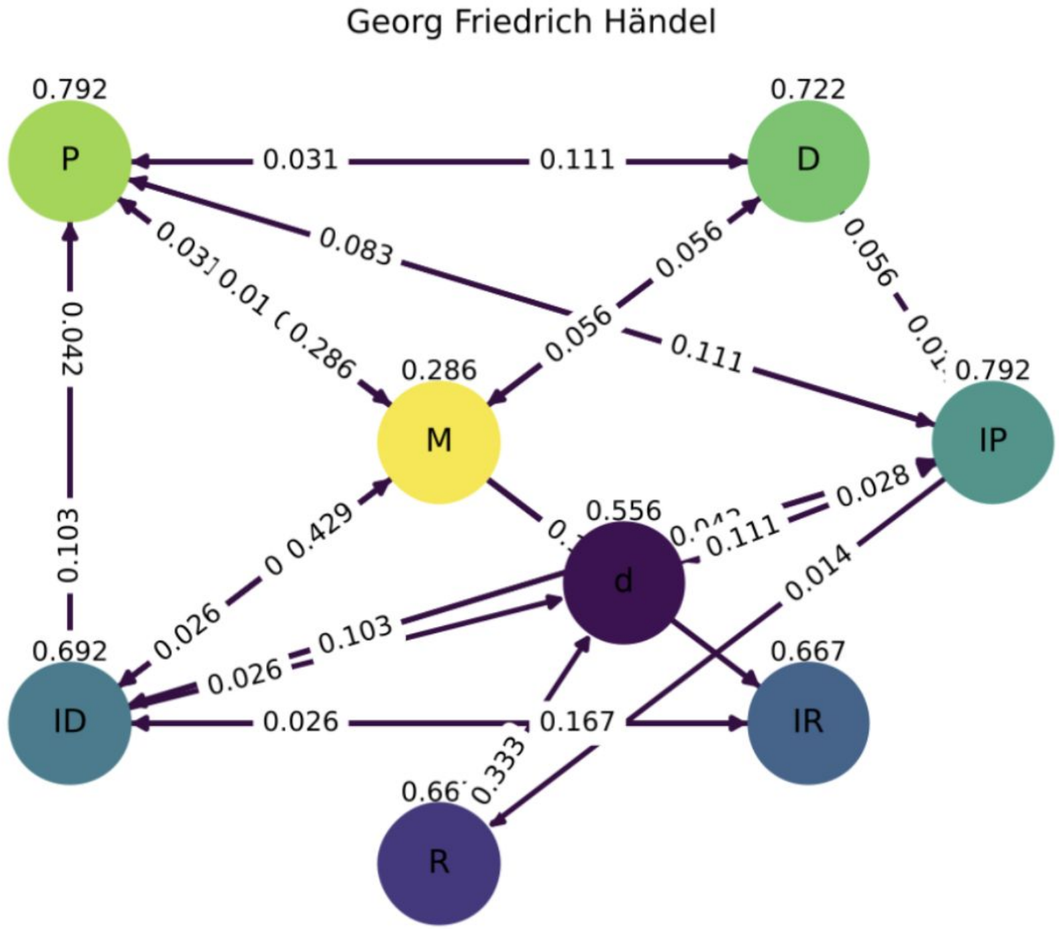
```python
# Calculate the number of rows needed for two columns
rows_needed = (n + 1) // 2  # Rounds up if odd number of graphs
fig, axes = plt.subplots(rows_needed, 2, figsize=(12, rows_needed * 5))  # 2 columns, rows as needed

if n == 1:
    axes = np.array([[axes]])  # Ensure axes is 2D array for single graph
elif n == 2:
    axes = axes.reshape(-1, 2)  # Reshape to ensure 2 columns when exactly two graphs

# Flatten the axes array and iterate over each subplot axis
axes = axes.flatten()
for ax, (title, matrix) in zip(axes[:n], transition_matrices.items()):
    transition_dict = transition_adjacency_matrix(matrix)
    G = nx.DiGraph()
    for source, transitions in transition_dict.items():
        for dest, prob in transitions.items():
            G.add_edge(source, dest, weight=prob)

    # Use node_positions for fixed positions
    pos = node_positions

    # Use node_colors dictionary for node coloring
    node_colors_list = [node_colors[node] for node in G.nodes]

    edge_labels = nx.get_edge_attributes(G, 'weight')

    nx.draw(G, pos, with_labels=True, node_color=node_colors_list, node_size=2000, edge_color='#3b0a45', width=2, arrowstyle='-|>', arrowsize=10, ax=ax)
    nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels, label_pos=0.3, ax=ax)

    ax.set_title(title)
    ax.axis('off')

# Hide any unused axes if there are fewer graphs than subplots
for ax in axes[n:]:
    ax.axis('off')

plt.tight_layout()
plt.savefig("markovchains.svg", dpi=800)
plt.show()
```

results and discussions

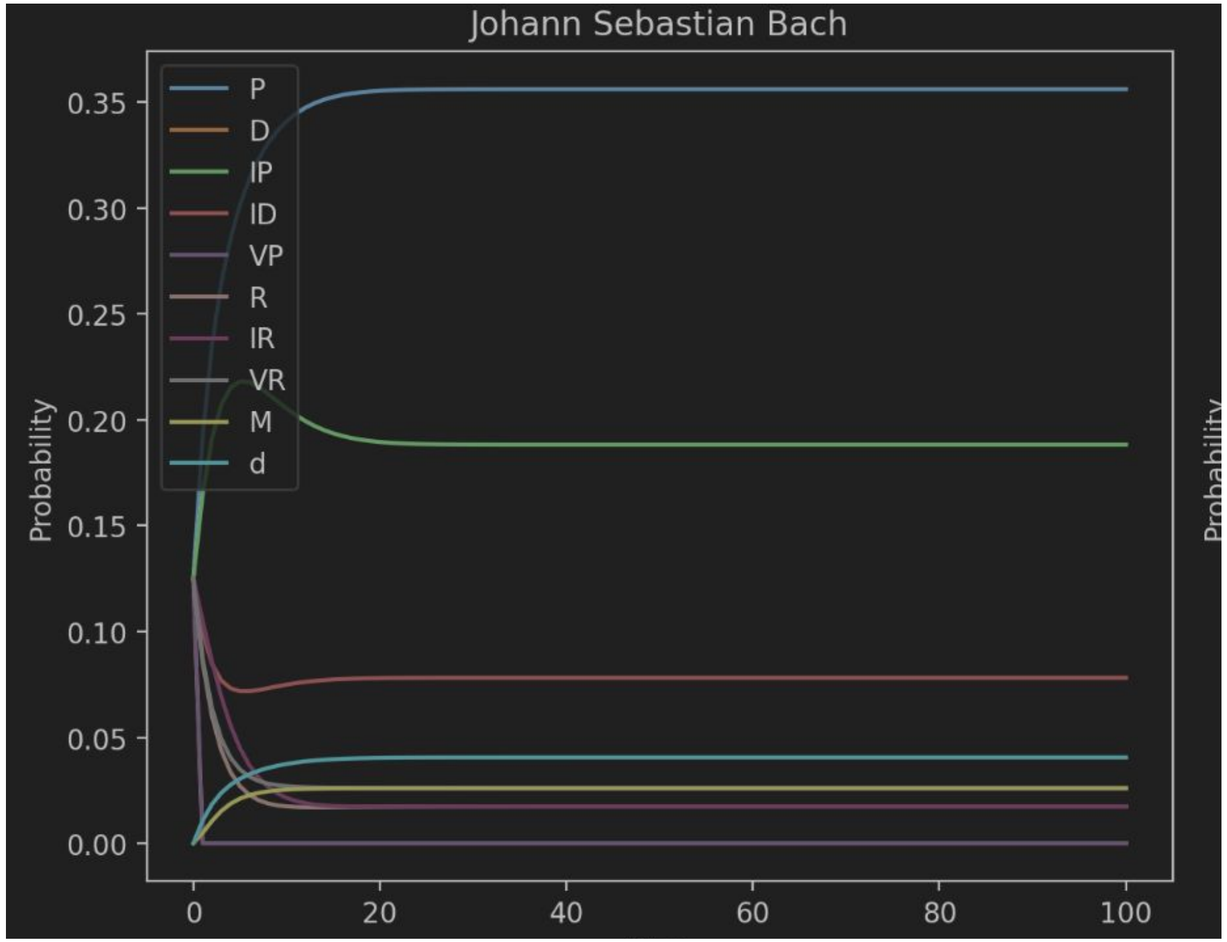
```python
for ax, composer in zip(axes, directories):
    matrix = transition_adjacency_matrix(aggregated_piece_matrices[composer])
    df_transition = pd.DataFrame(matrix, index=states, columns=states)

    open_states = [state for state, degree in closure_degrees.items() if degree > 0.5]
    closed_states = [state for state, degree in closure_degrees.items() if degree <= 0.5]

    # Calculate probabilities
    prob_open_to_closed = df_transition.loc[open_states, closed_states].sum().sum()
    prob_closed_to_open = df_transition.loc[closed_states, open_states].sum().sum()
    prob_open_to_open = df_transition.loc[open_states, open_states].sum().sum()
    prob_closed_to_closed = df_transition.loc[closed_states, closed_states].sum().sum()

    # Normalize probabilities (if needed)
    total = prob_open_to_closed + prob_closed_to_open + prob_open_to_open + prob_closed_to_closed
    probabilities = [prob_open_to_closed, prob_closed_to_open, prob_open_to_open, prob_closed_to_closed] / total
```

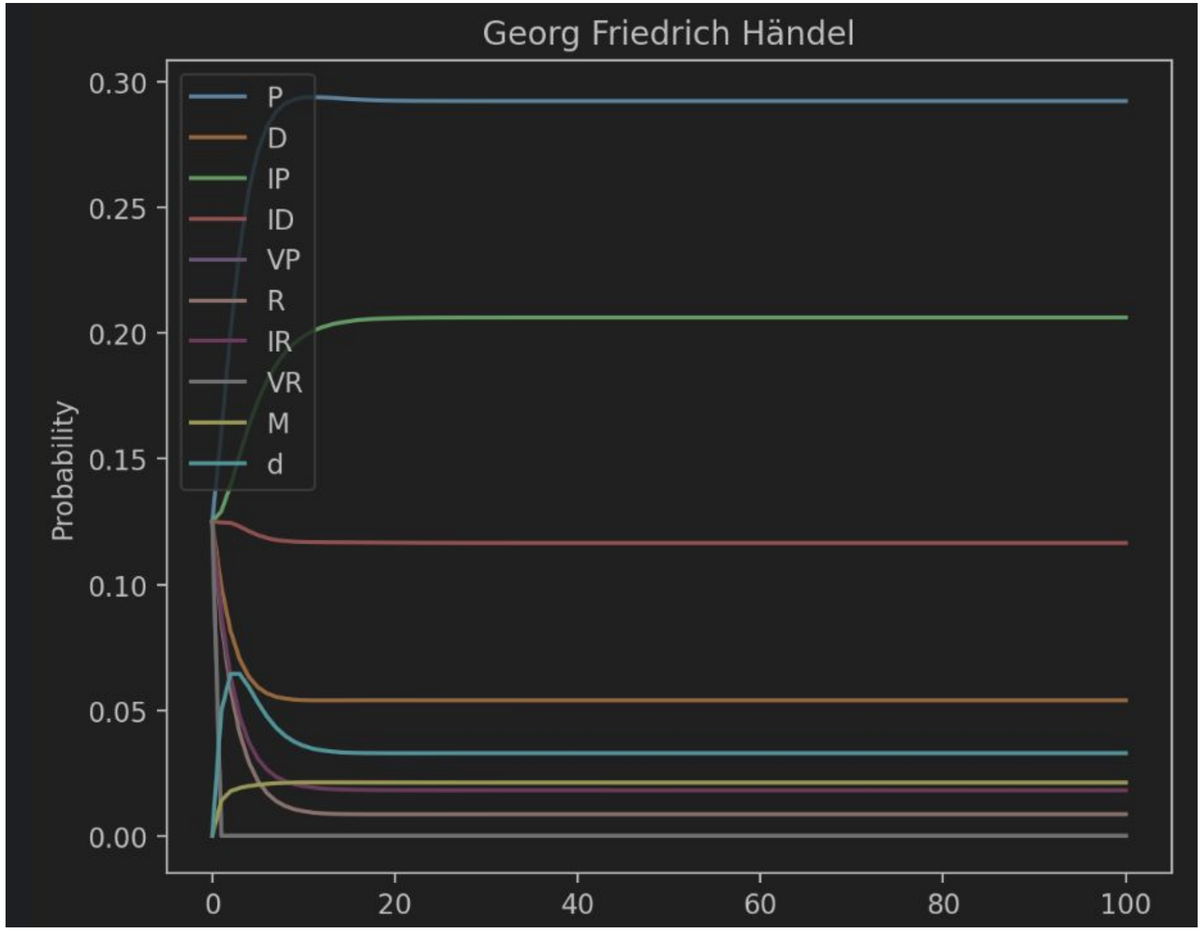
Section IV:
Baroque Era



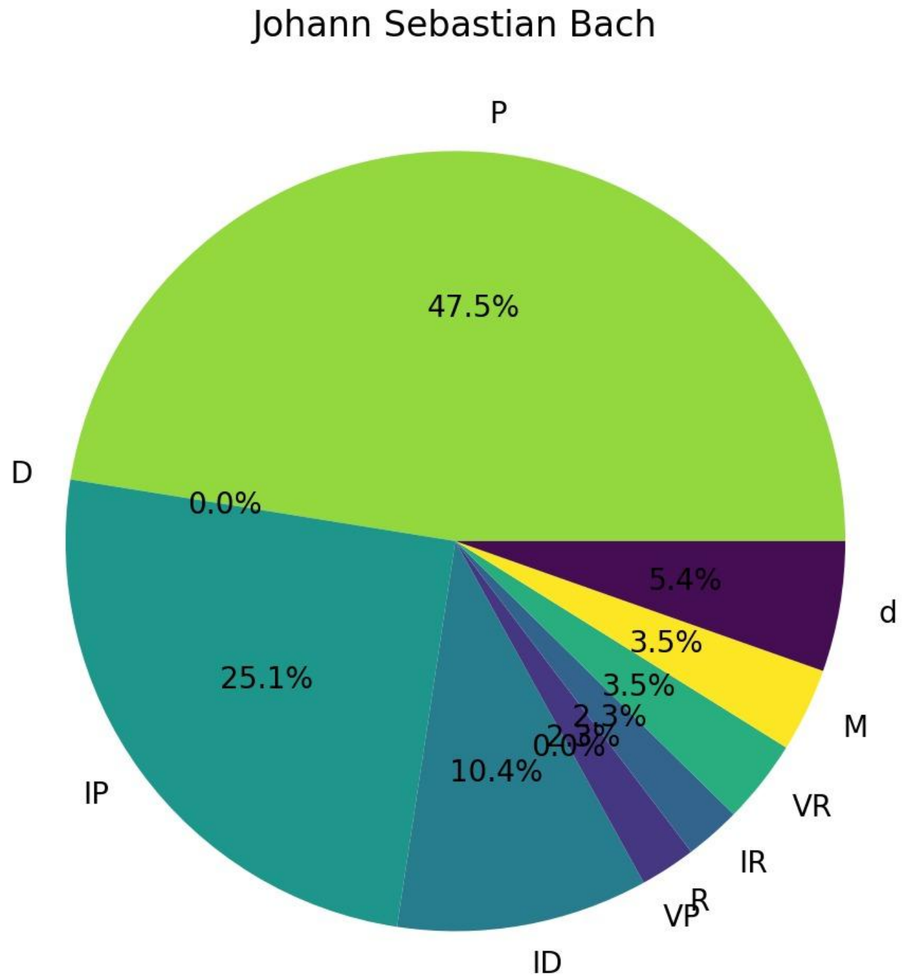


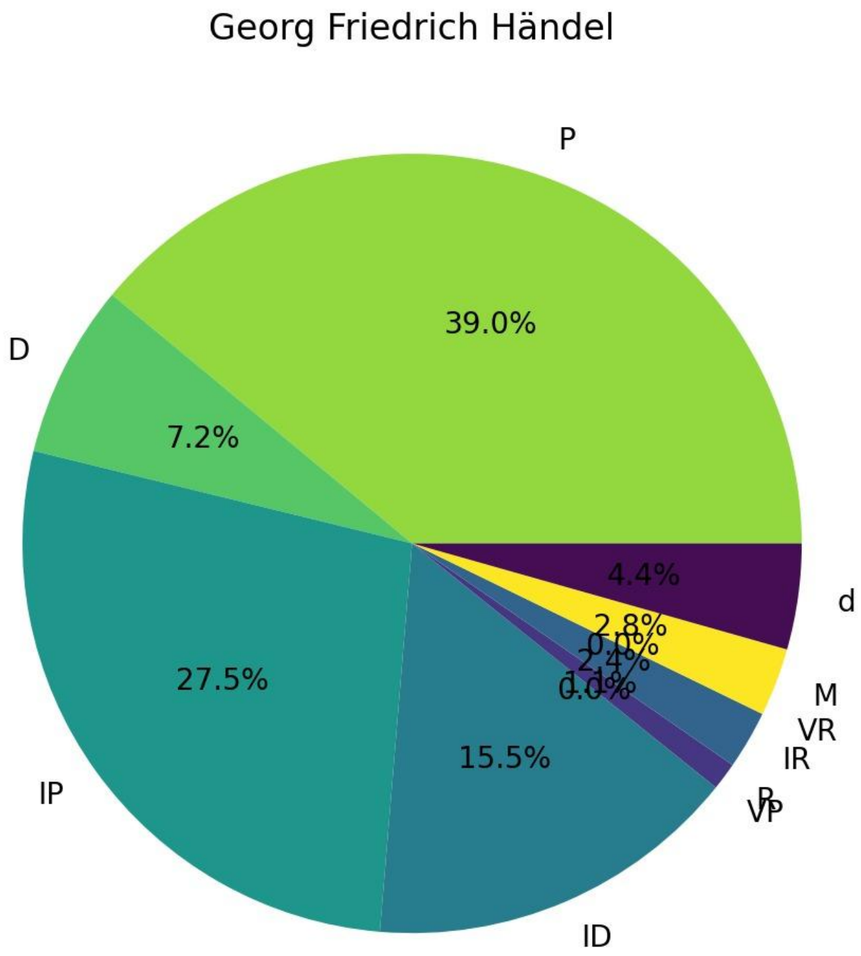
Section IV:
Baroque Era



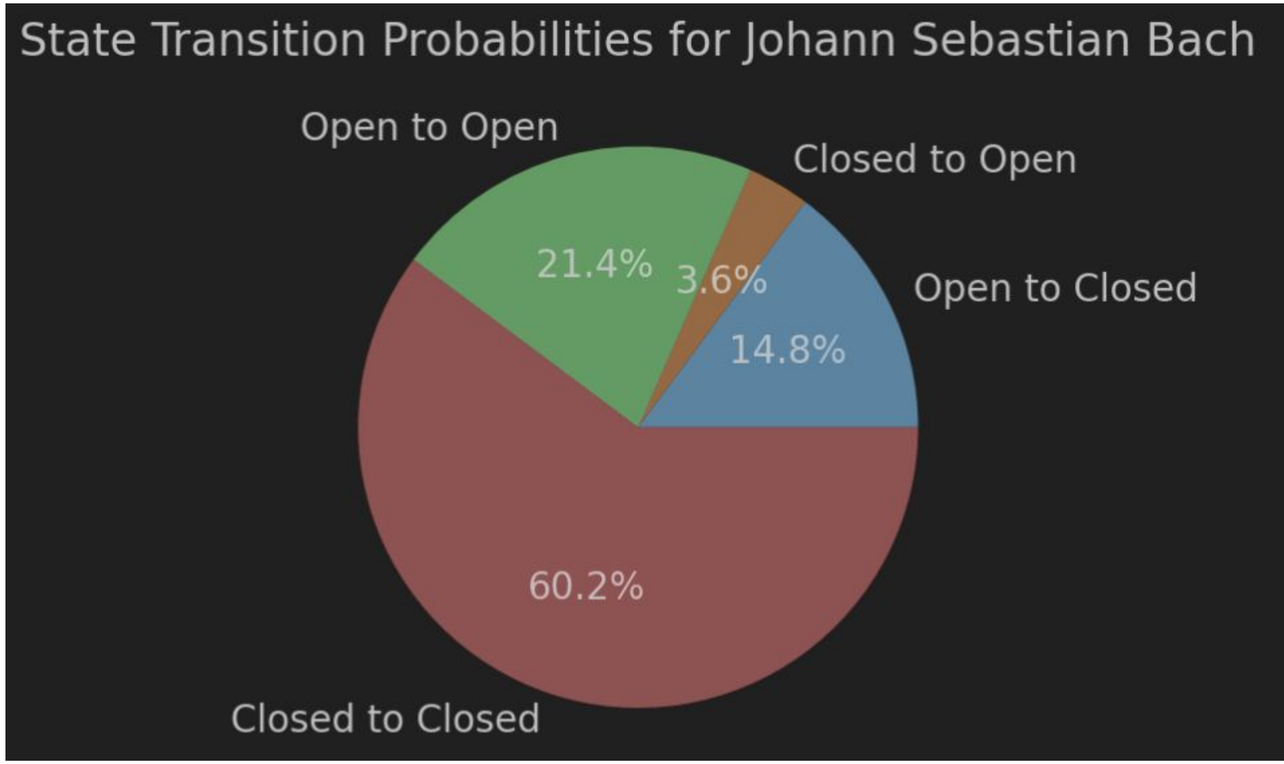


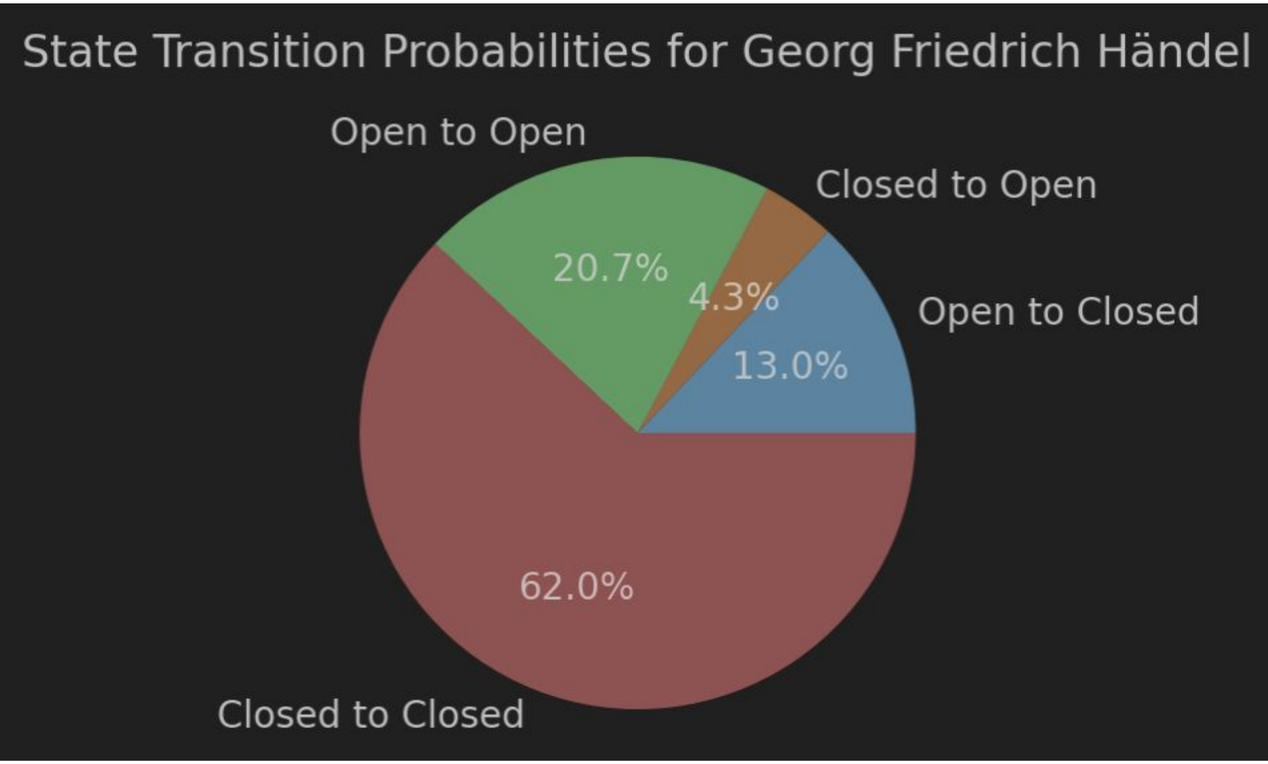
Section IV:
Baroque Era



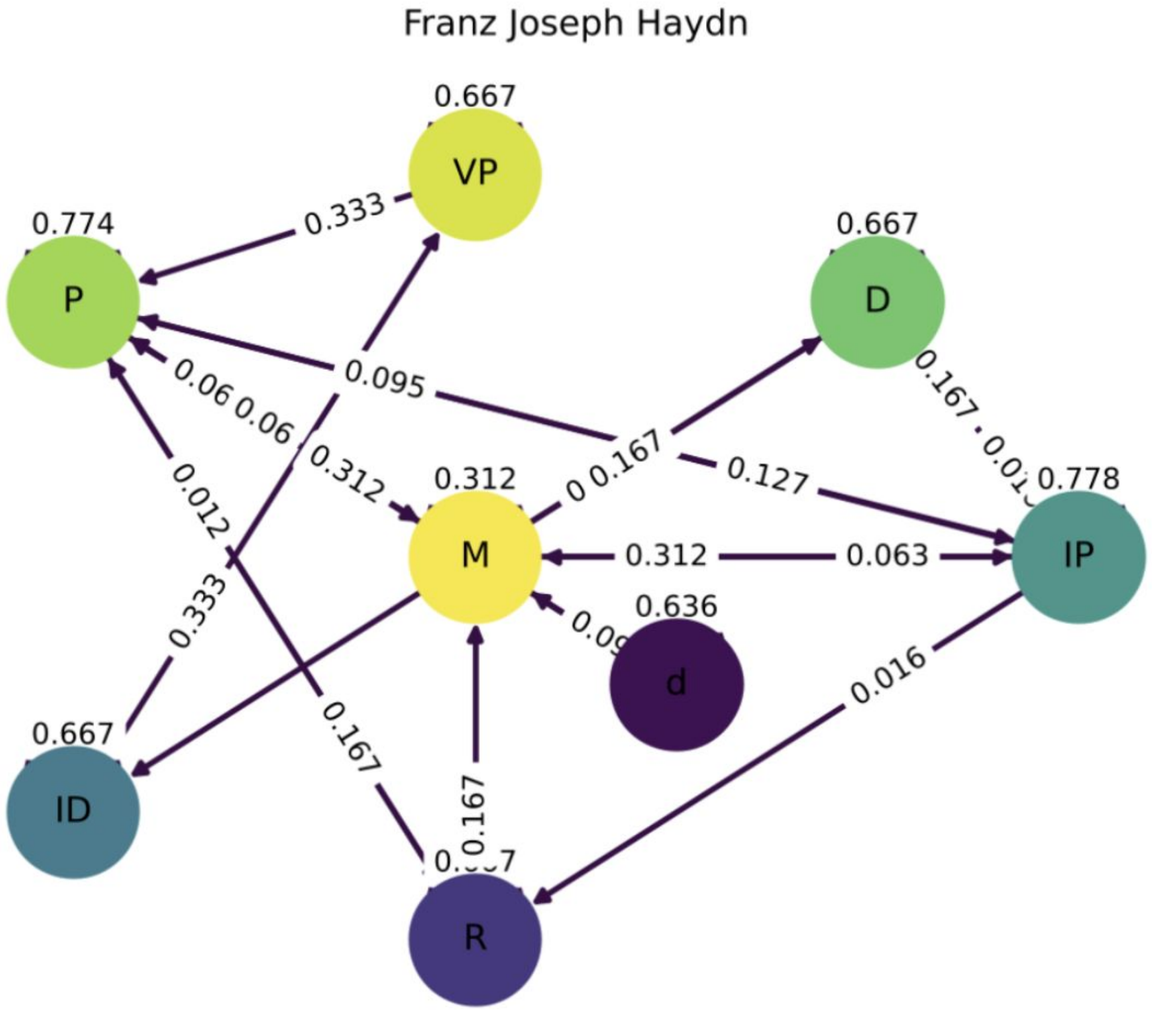


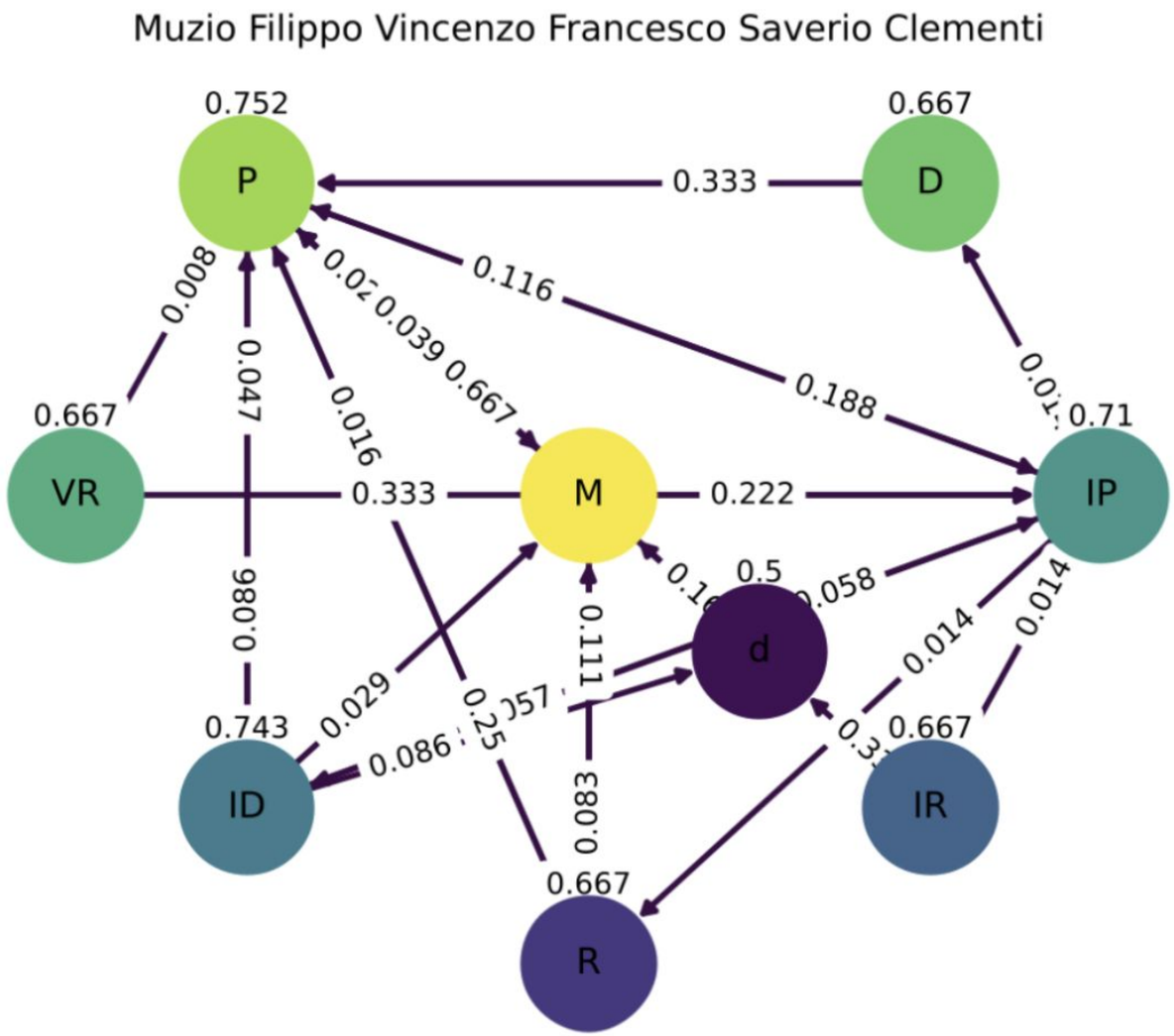
Section IV:
Baroque Era



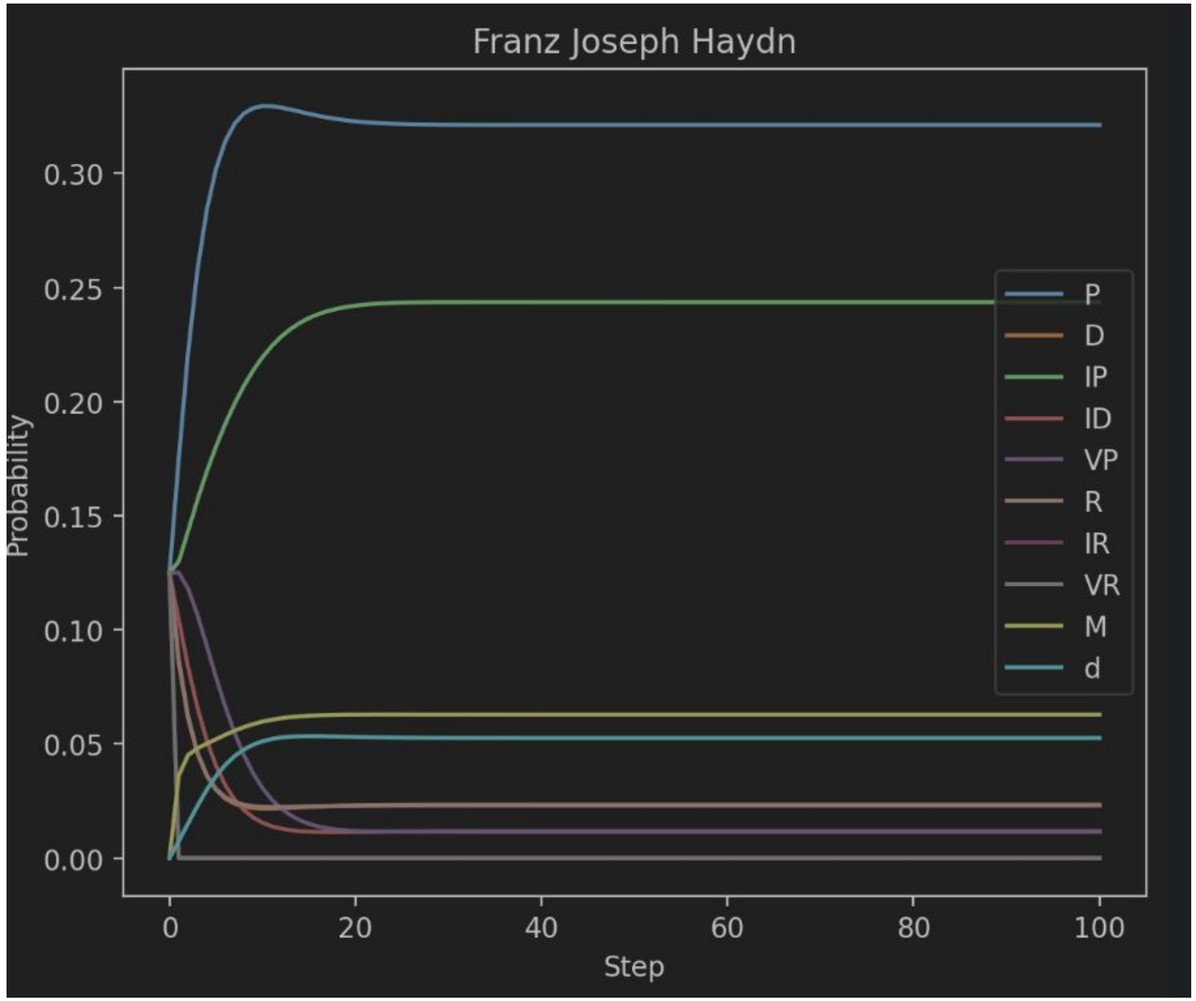


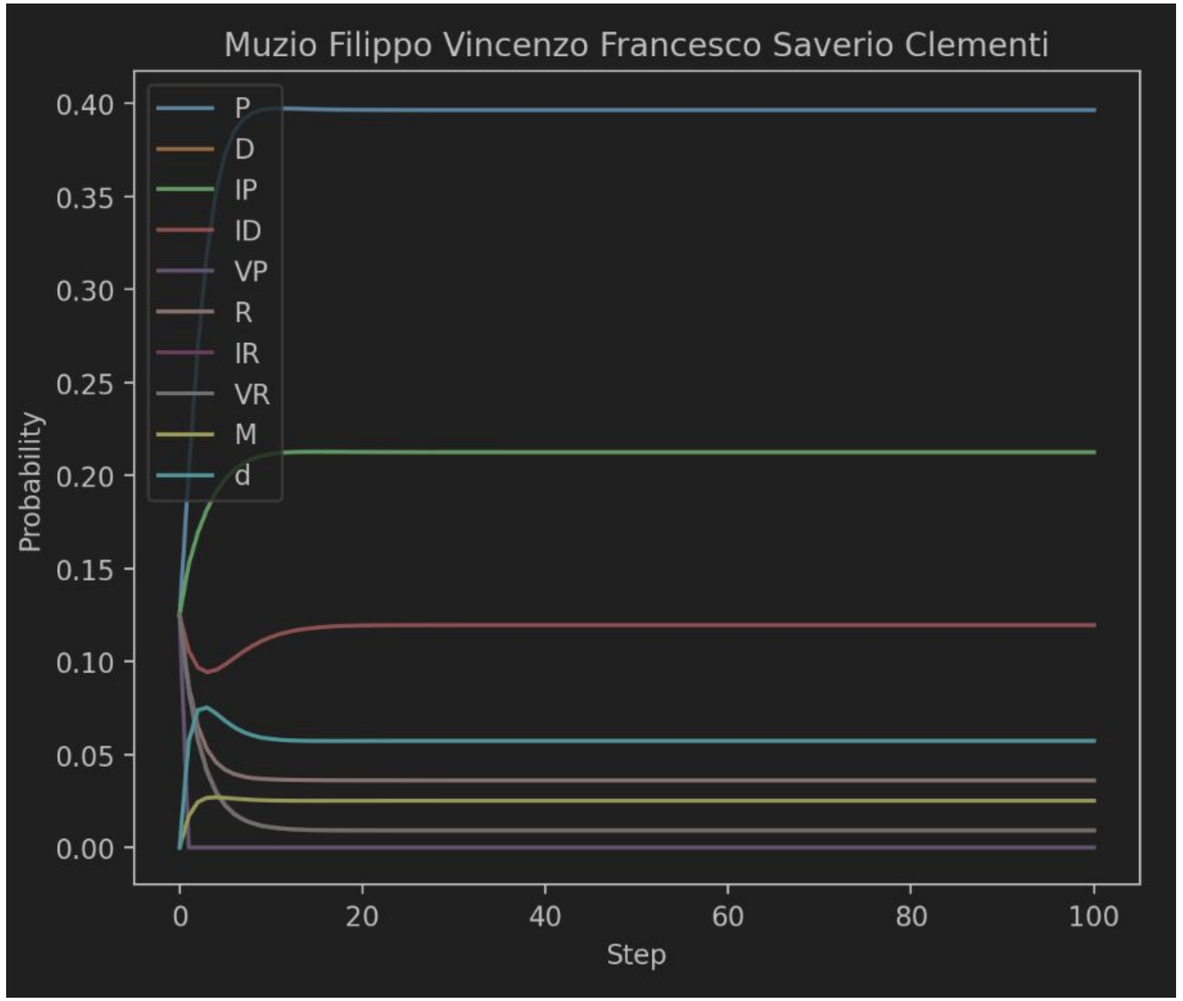
Section IV:
Classical Era



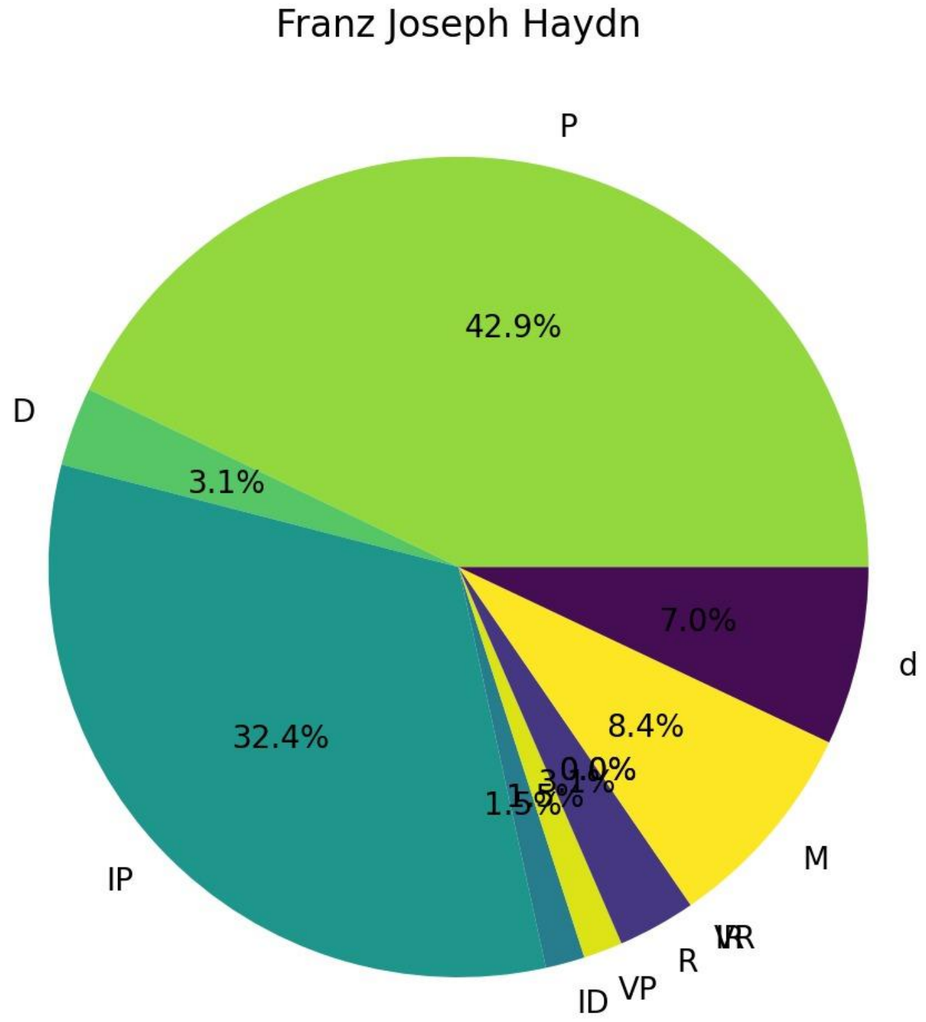


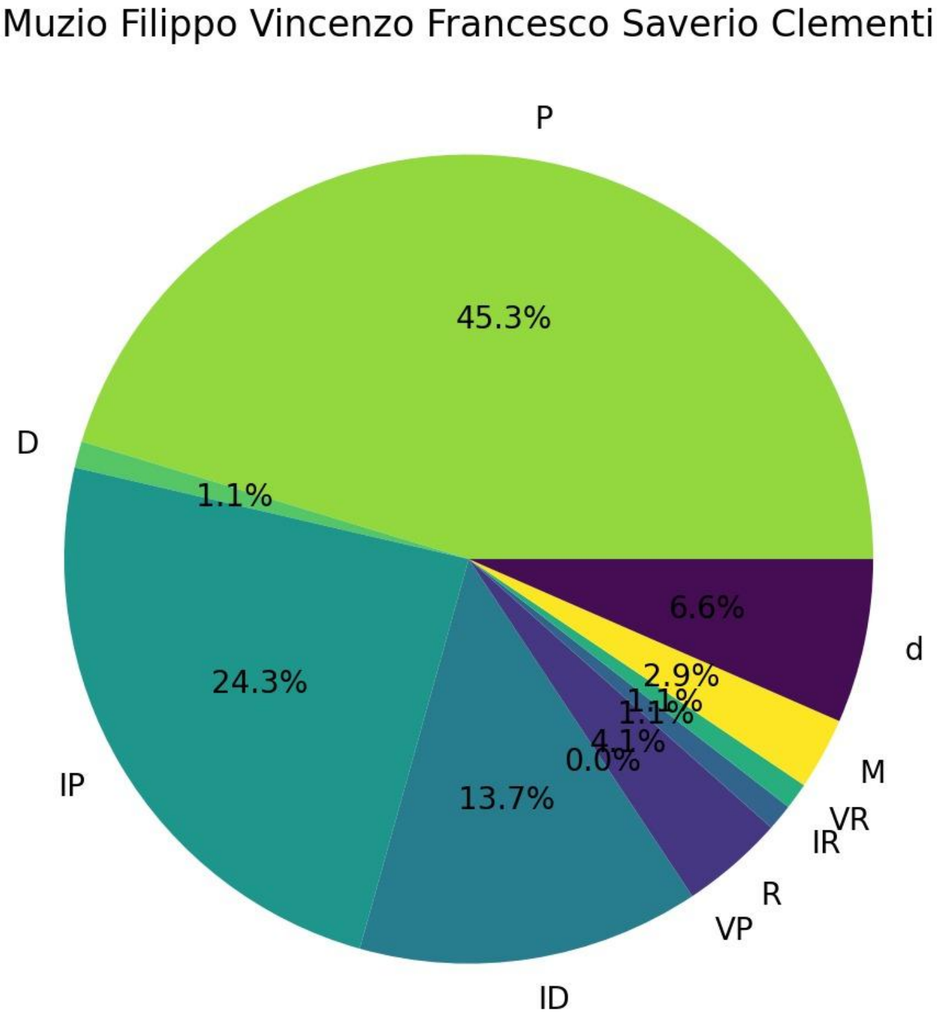
Section IV:
Classical Era



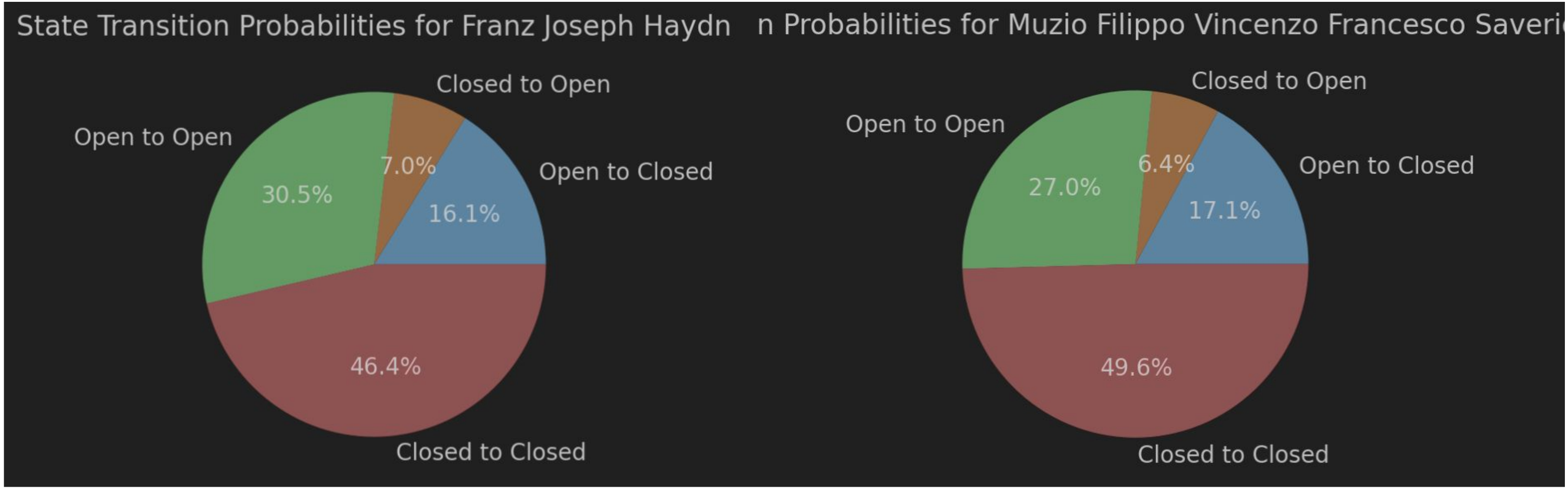


Section IV:
Classical Era

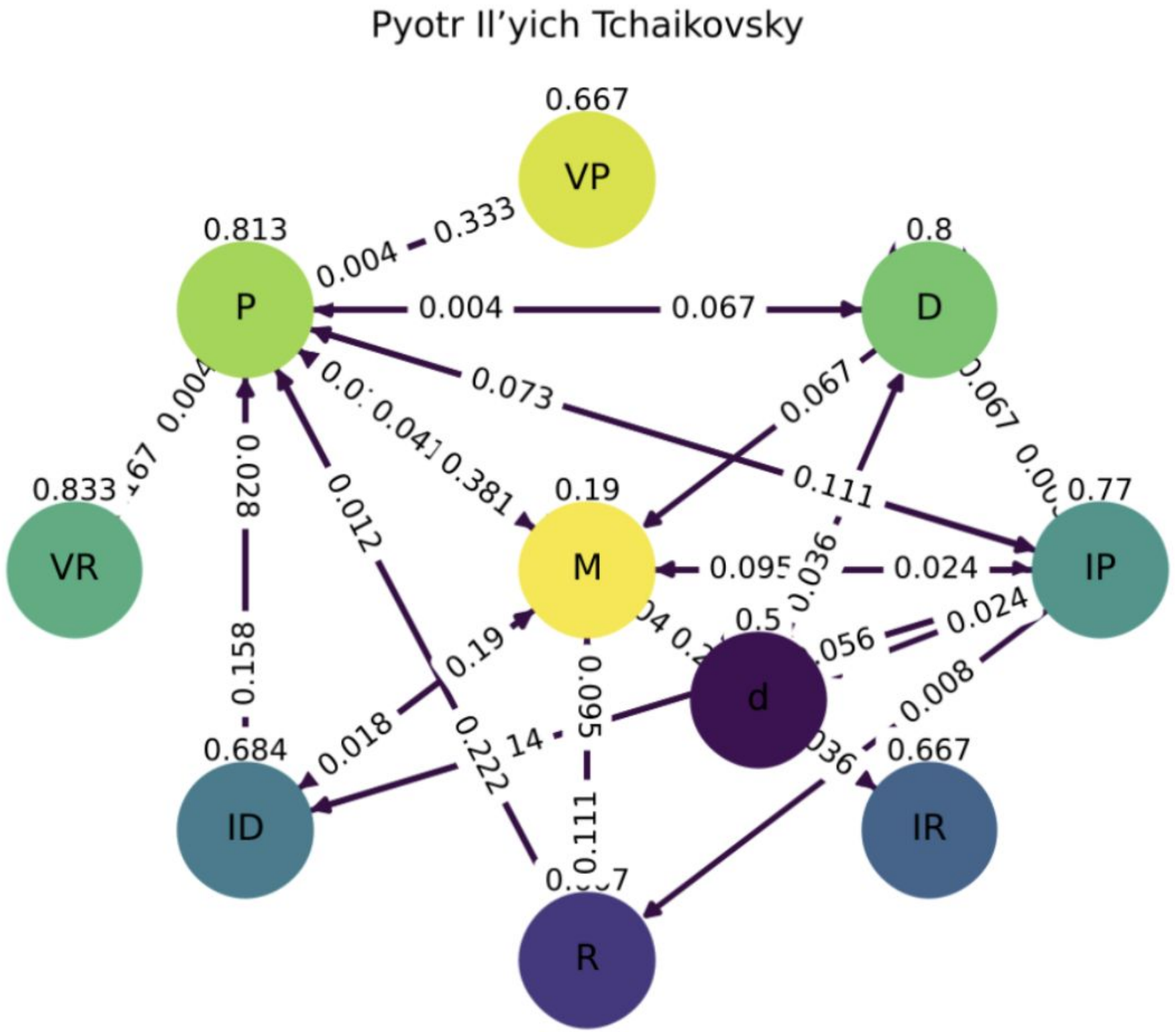


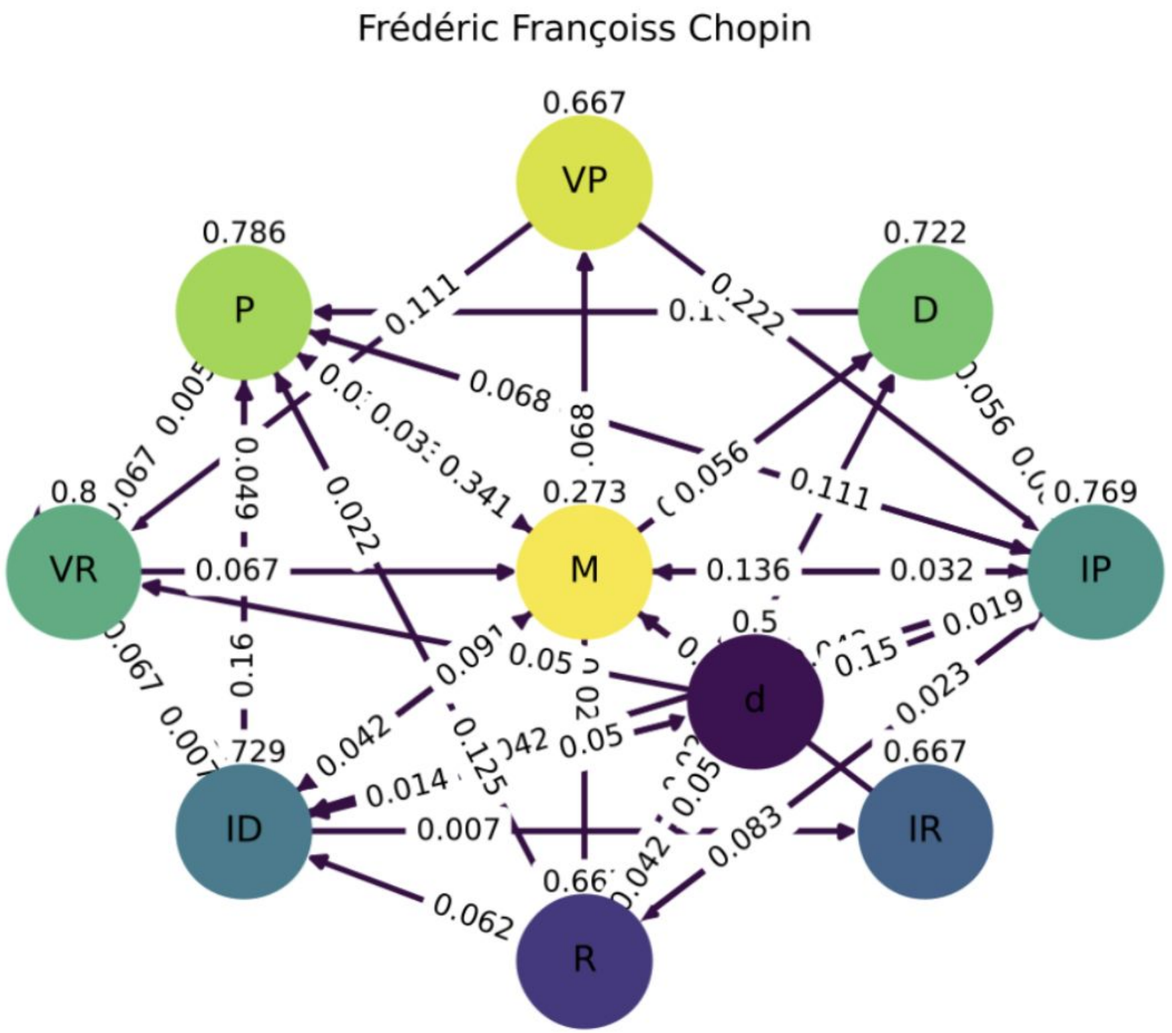


Section IV:
Classical Era

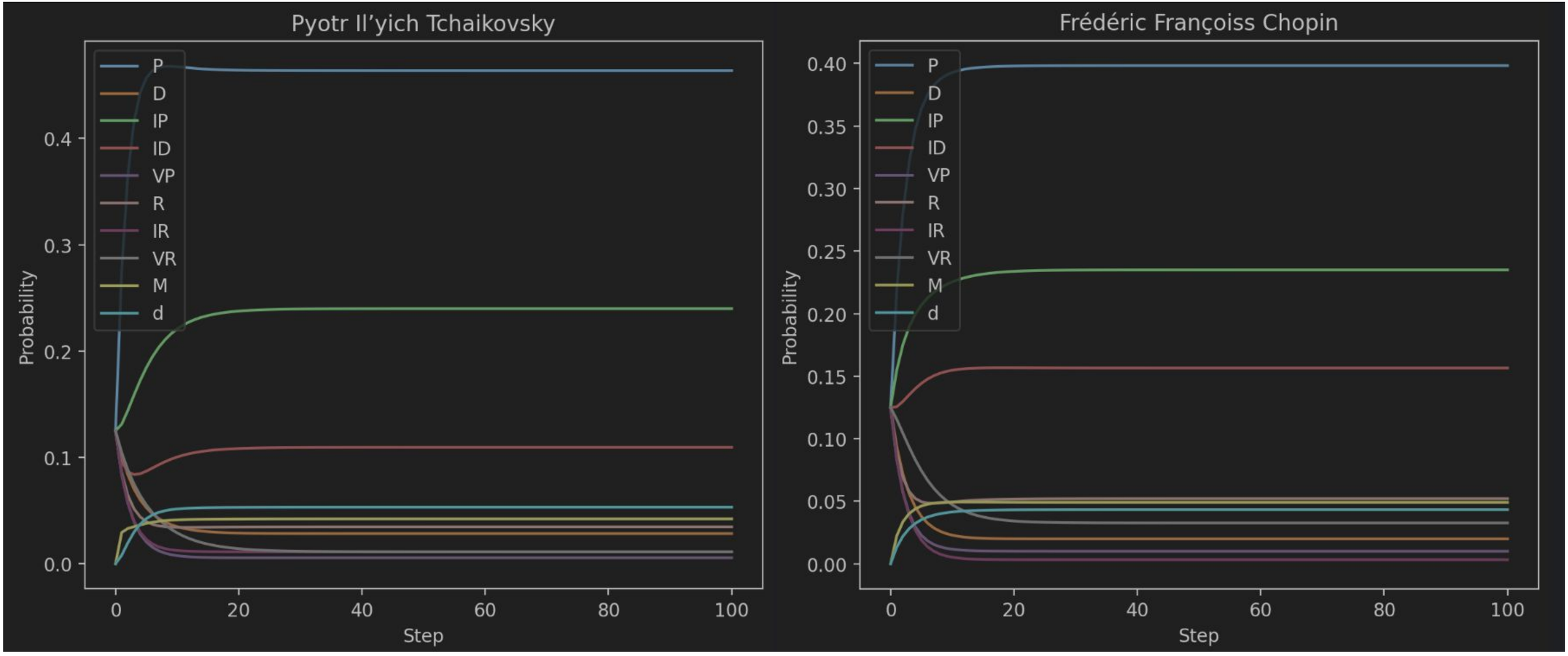


Section IV:
Romantic Era

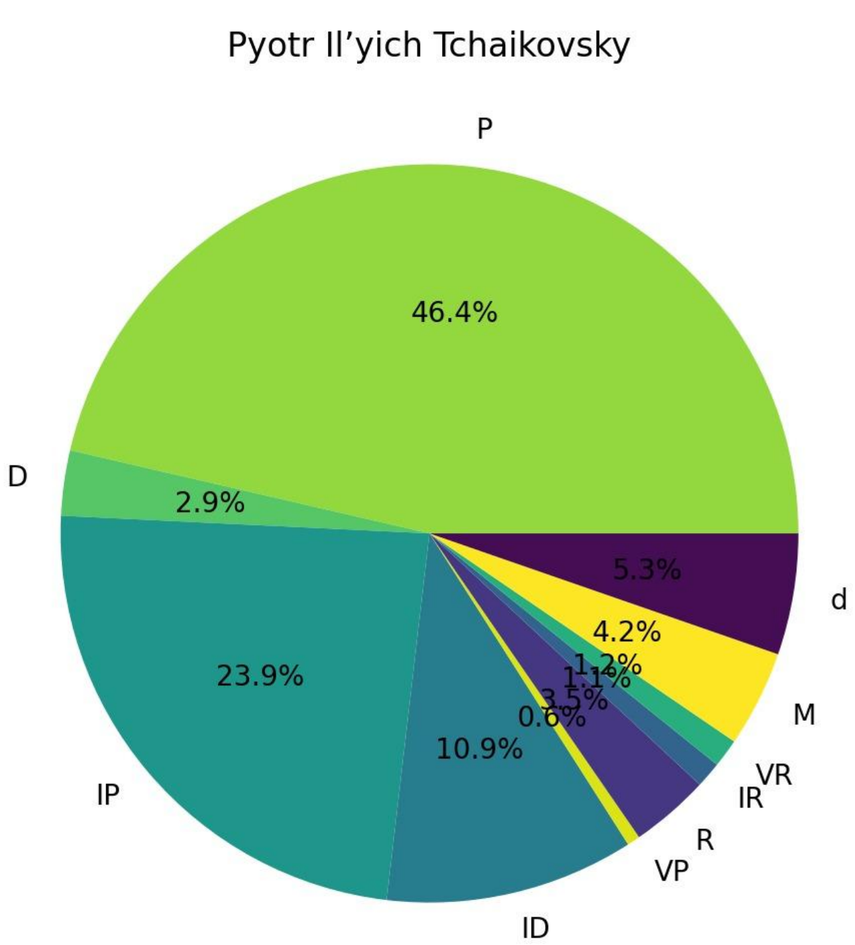


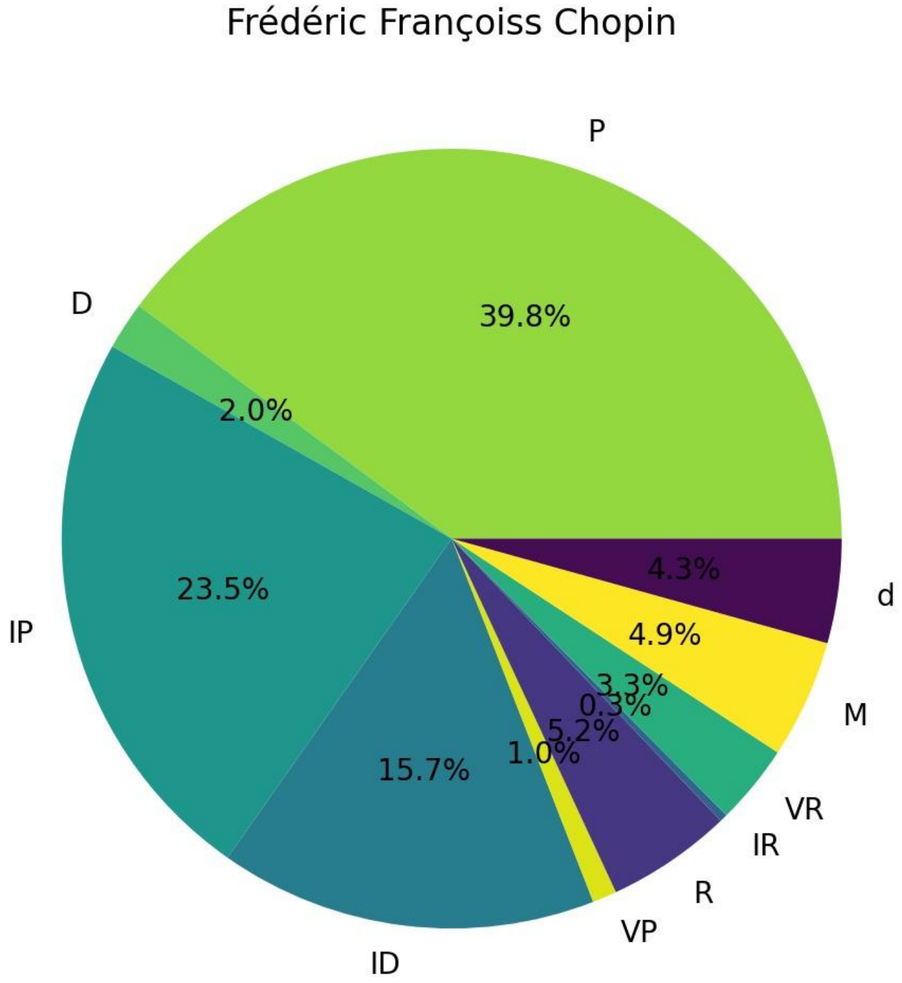


Section IV:
Romantic Era

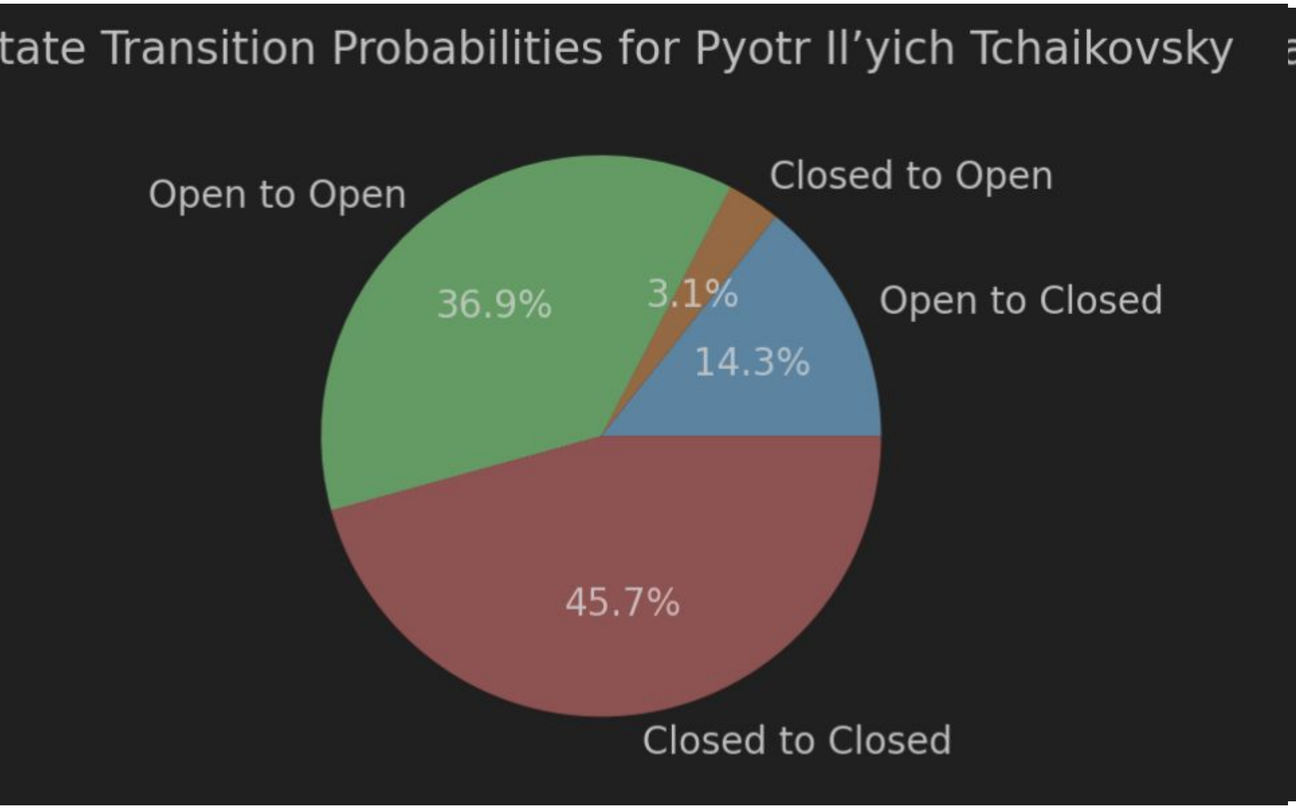


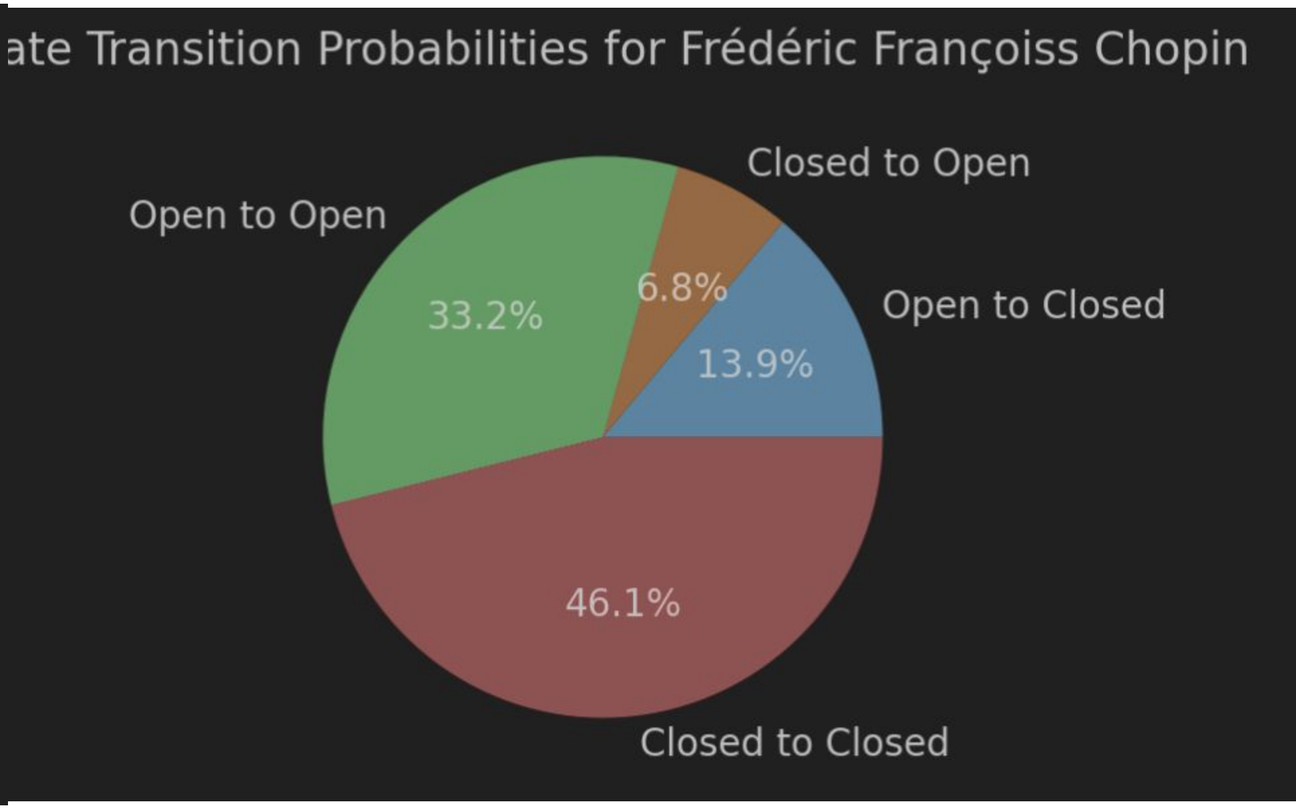
Section IV:
Romantic Era





Section IV:
Romantic Era





Section IV:
Validation Methods

Entropy rate

- **Measure of Uncertainty**:
  - Higher entropy rates indicate more variability and less predictability in musical transitions, reflecting potentially more complex or innovative musical structures.
- **Analysis of Musical Styles***:
  - Compare different styles or composers based on the predictability of their musical transitions.

```
def calculate_entropy_rate(transition_matrices):
    results = []
    for title, matrix in transition_matrices.items():
        try:
            matrix = np.array(matrix)

            # Check if matrix is square
            if matrix.shape[0] != matrix.shape[1]:
                raise ValueError("Matrix must be square to form a valid Markov chain.")

            # Calculate the stationary distribution
            eigenvalues, eigenvectors = np.linalg.eig(matrix.T)
            stationary_distribution = np.real(eigenvectors[:, np.isclose(eigenvalues, 1)].flatten())
            if stationary_distribution.size == 0:
                raise ValueError("No stationary distribution found.")
            stationary_distribution = stationary_distribution / stationary_distribution.sum()

            # Calculate the entropy rate
            entropy_rate = 0
            for i in range(len(matrix)):
                entropy_rate -= stationary_distribution[i] * np.sum(
                    matrix[i] * np.log2(matrix[i] + (matrix[i] == 0))
                )
            results.append({"Title": title, "Entropy Rate": entropy_rate})
        except Exception as e:
            results.append({"Title": title, "Entropy Rate": f"Error: {str(e)}"})
    return pd.DataFrame(results)
```

Section IV:
Validation Methods

Insights

- Composers like Frédéric Chopin, Ludwig van Beethoven, and Jakob Mendelssohn Bartholdy have relatively high entropy rates, indicating a wide range of musical ideas and structures within their works.
- Composers with lower entropy rates, such as Grieg, Wagner, and Puccini, tend to exhibit more structural consistency or repetition within their compositions. This could be due to particular stylistic tendencies or adherence to certain compositional forms.

	Title	Entropy Rate
0	Edvard Hagerup Grieg	1.004593
1	Franz Joseph Haydn	1.163731
2	Franz Peter Schubert	1.341759
3	Frédéric Françoiss Chopin	1.452031
4	Georg Friedrich Händel	1.177495
5	Georges Bizet	1.225403
6	Giacomo Antonio Domenico Michele Secondo Maria...	1.075666
7	Giuseppe Fortunino Francesco Verdi	1.239079
8	Jakob Ludwig Felix Mendelssohn Bartholdy	1.448044
9	Jean-Philippe Rameau	1.298289
10	Johann Sebastian Bach	1.177880
11	Johannes Brahms	1.213878
12	Louis A. Saint-Jacome	0.000000
13	Ludwig van Beethoven	1.431866
14	Muzio Filippo Vincenzo Francesco Saverio Clementi	1.209946
15	Pyotr Il'yich Tchaikovsky	1.215981
16	Robert Alexander Schumann	1.333885
17	Wilhelm Richard Wagner	1.062831
18	Wolfgang Amadeus Mozart	1.193948

Section IV:
Validation Methods

Chi-Squared Test

- **Comparison Across Pieces or Styles**:
  - Statistically compare the transition frequencies between different pieces or styles to see if they significantly differ.

```
# Perform chi-squared test on the smoothed data
try:
    combined_data = np.vstack(results)
    chi2, p_value, _, _ = chi2_contingency(combined_data.T)
    return chi2, p_value
except ValueError as e:
    return f"Chi-squared test failed: {str(e)}"
```

```
def test_homogeneity(transition_matrices, smoothing_value=1e-3):
    results = []
    for title, matrix in transition_matrices.items():
        try:
            # Apply Laplace smoothing to avoid zeros
            smoothed_matrix = matrix + smoothing_value

            # Normalize rows to sum to 1 after smoothing
            smoothed_matrix = smoothed_matrix / smoothed_matrix.sum(axis=1, keepdims=True)

            # Flatten the matrix and append
            results.append(smoothed_matrix.flatten())
        except Exception as e:
            print(f"Error processing matrix {title}: {str(e)}")
            continue
```

Section IV:
Validation Methods

Insights

- These results indicate that compositions within each period (Baroque, Classical, Romantic) exhibit significant homogeneity in terms of patterns and structure, with p-values of 1.0 suggesting strong evidence of homogeneity. However, the degree of homogeneity varies across periods, with Romantic composers demonstrating the highest level of homogeneity, followed by Classical and Baroque composers.

```
test_homogeneity(baroque_dict)
✓ 0.0s
(8.650428705320898, 1.0)
```

```
test_homogeneity(classical_dict)
✓ 0.0s
(17.501005785438895, 1.0)
```

```
test_homogeneity(romantic_dict)
✓ 0.0s
(64.8292065005868, 1.0)
```

Section IV:
Validation Methods

Kullback-Leibler (KL) divergence

- **Comparison Across Pieces:**
  - Measures the difference between probability distributions directly and provides insights into information content.
  - KL Divergence allows us to assess the level of similarity or dissimilarity in the structural patterns of IR transitions within compositions belonging and not belonging to the same historical period.

```
def kl_divergence(p, q):
    epsilon = 1e-10  # small epsilon value to avoid division by zero
    p_safe = p + epsilon
    q_safe = q + epsilon
    return np.sum(p_safe * np.log(p_safe / q_safe))
```

Section IV:
Validation Methods

```
Average KL Divergence within each era:
Baroque Era: 43.519094238568506
Classical Era: 46.53694473989092
Romantic Era: 46.56795552837973

Average KL Divergence between different eras:
Baroque Era vs Classical Era: 55.027081338980175
Baroque Era vs Romantic Era: 55.405373766843624
Classical Era vs Romantic Era: 51.7213395213562
```

Insights

- The Baroque Era exhibits the lowest average KL Divergence, indicating relatively similar compositional patterns and structures within this period.
- The Classical and Romantic Eras show slightly higher average KL Divergence, suggesting greater variability or diversity in compositional styles and structures compared to the Baroque Era.
- Comparisons between the Baroque and Classical Eras, as well as between the Baroque and Romantic Eras, show higher KL Divergence values, indicating greater differences in compositional styles between these eras.
- The KL Divergence value between the Classical and Romantic Eras is slightly lower, suggesting a relatively closer relationship or similarity in compositional styles between these two periods compared to the comparisons involving the Baroque Era.

Conclusion and Recommendations

conclusions
recommendations

Conclusions

- Much of our initial hypotheses on analyzing the music has not been proven correct. Since the IR model handles features based on how people perceive and realize musical patterns, much of the conventional and traditional predispositions we had would not hold.

- While Markov models based on the IR model is not an ideal representation of reality, the model produces reasonable approximations.

Recommendations

- Should it be possible to have a music dataset with the fulfillment or denial of expectation, a Hidden Markov Model implementation would likely yield good insights on how the melodic archetypes influence expectation.

- The current database we used is small, and the pieces are incomplete. Moreover, the data used is skewed greatly towards the romantic era.
  - The accuracy of results may vary across era.

- IR Model segments based on multiple factors, such as metric emphasis, intervallic motion, and registral direction, to name a few.
  - Perhaps a clearer grouping and segmentation would yield better results.

References:

Ala, & Wadi. (2012). Analysis of Music Note Patterns Via Markov Chains (Issue 2). http://collected.jcu.edu/honorspapershttp://collected.jcu.edu/honorspapers/2

Clinton, W., & Strong, A. (2019). Music Improvisation in Python using a Markov Chain Algorithm.

Foster, R. (1990). Review of Eugene Narmour, The Analysis and Cognition of Basic Melodic Structures: The Implication-Realization Mode

Glenn, E. (1997). Simplifying the Implication-Realization Model of Melodic Expectancy

Neve, G., & Orio, N. (n.d.). LNCS 3652 - A Comparison of Melodic Segmentation Techniques for Music Information Retrieval

Noto, K., & Takegawa, Y. (2023). A Rule-Based Method for Implementing Implication-Realization Model

Ramanto, A. S., Nur, U., & Maulidevi, S. T. (2017). Markov Chain Based Procedural Music Generator with User Chosen Mood Compatibility.

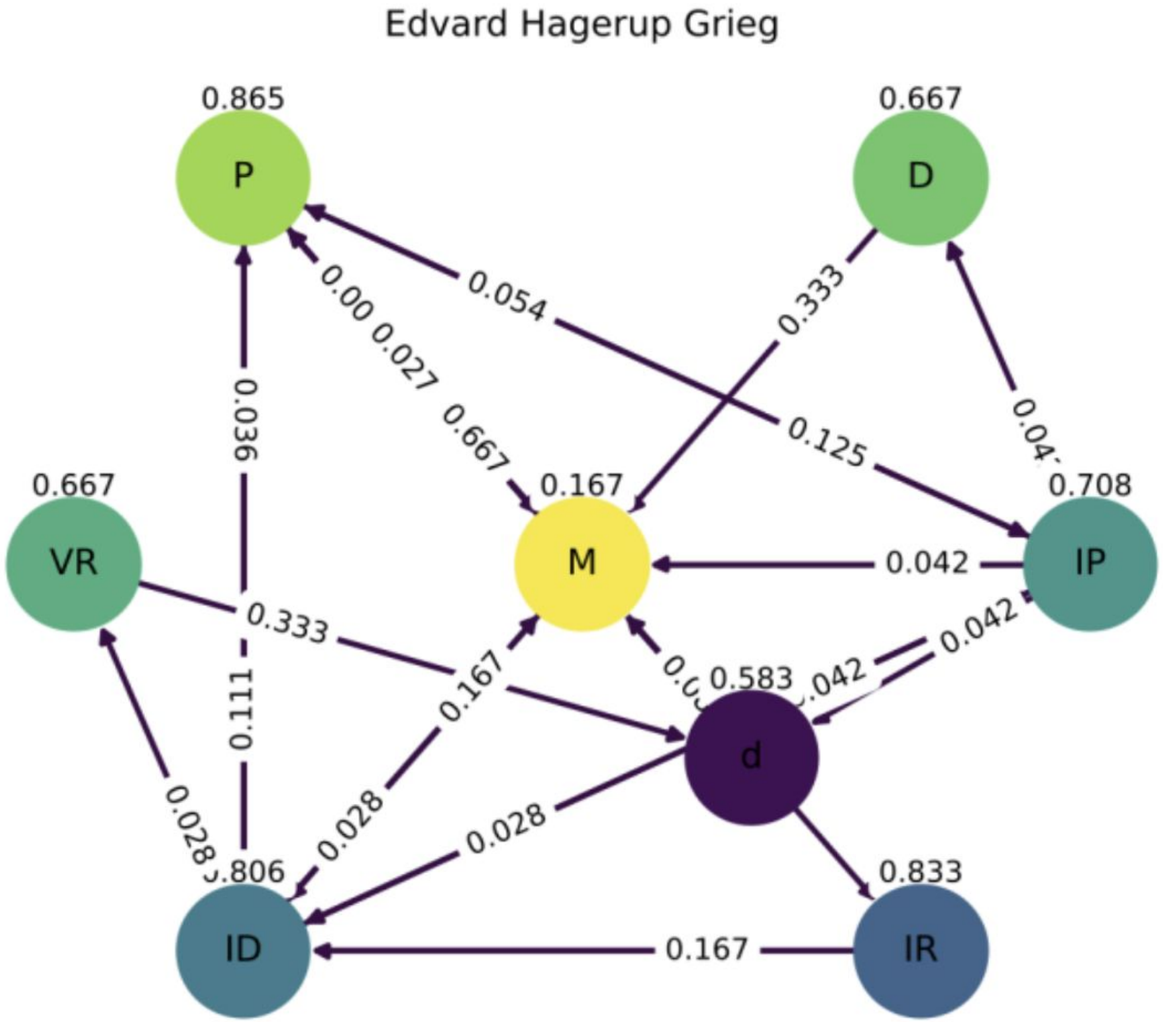
Royal, M. (1995). Review of The Analysis and Cognition of Basic Melodic Structures and The Analysis of Cognition of Melodic Complexity by Eugene Narmour

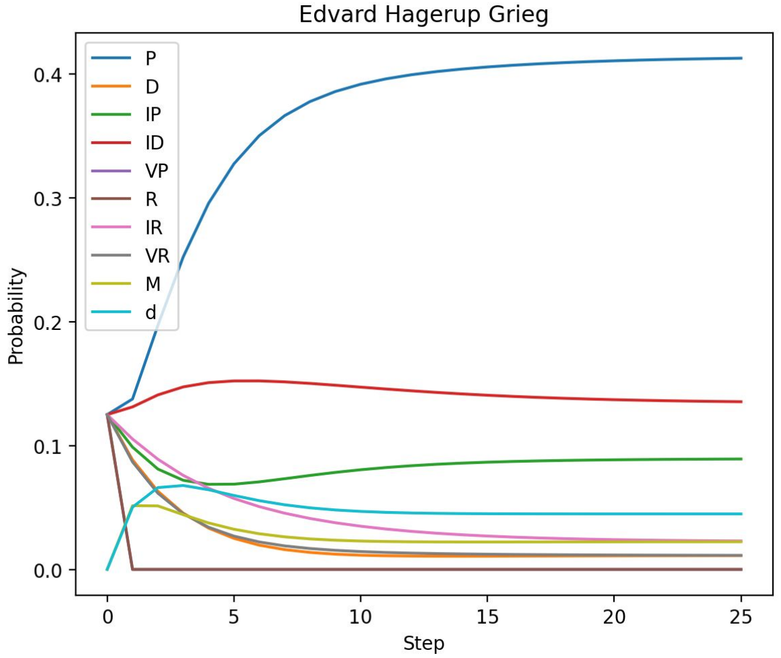
Shapiro, I., & Huber, M. (2021). Markov Chains for Computer Music Generation.

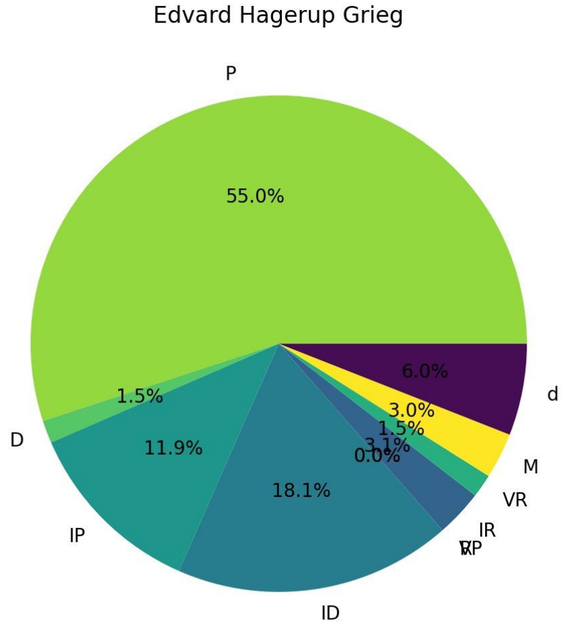
Wen, R., Chen, K., Zhang, Y., Huang, W., Tian, J., Xu, K., & Wu, J. (2018). A Model of Music Perceptual Theory Based on Markov Chains.

Yazawa, S., Hasegawa, Y. (n.d.). MIREX 2013 SYMBOLIC MELODIC SIMILARITY: Melodic Similarity based on Extension Implication-Realization Model

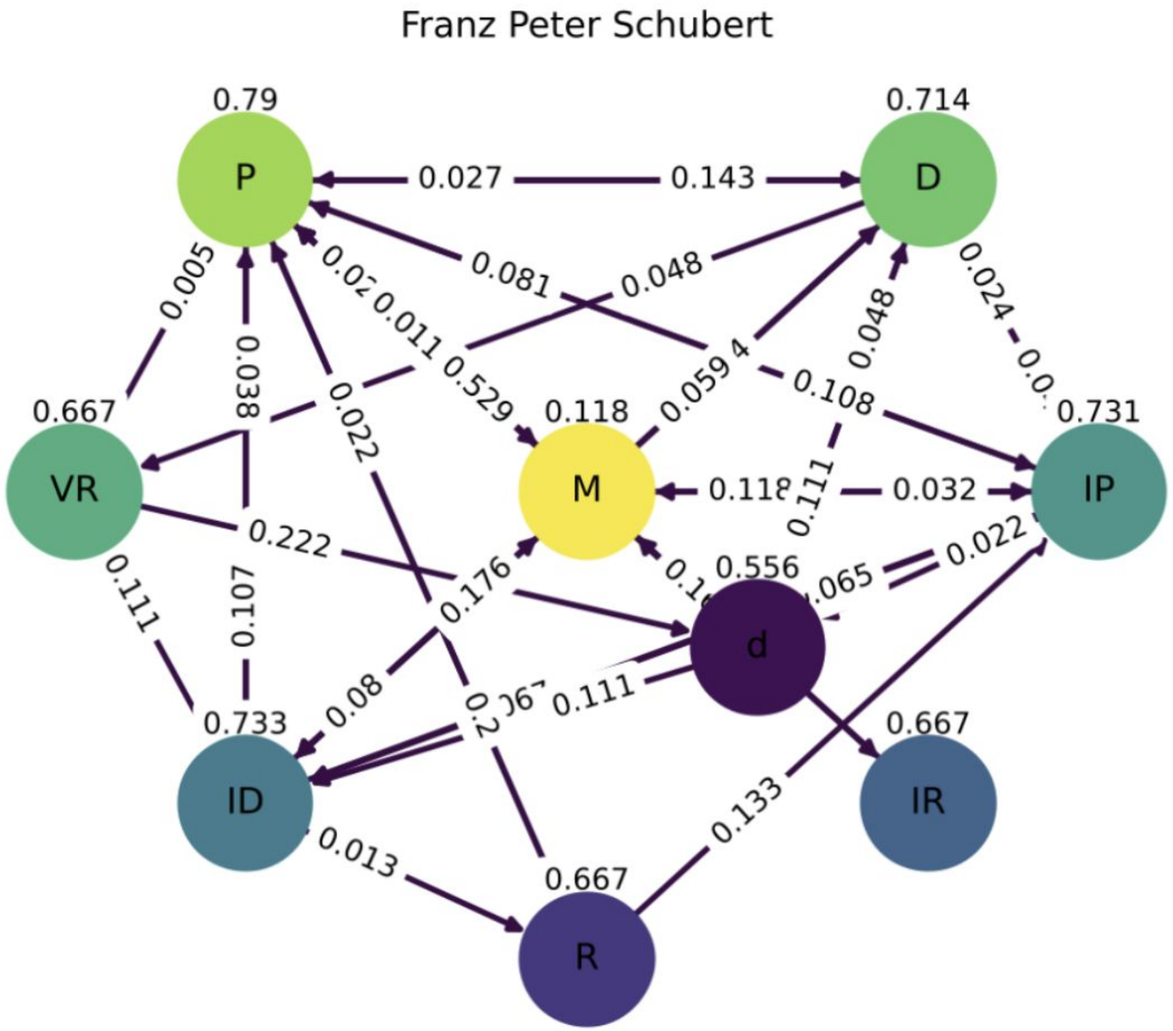
Section IV:
Results

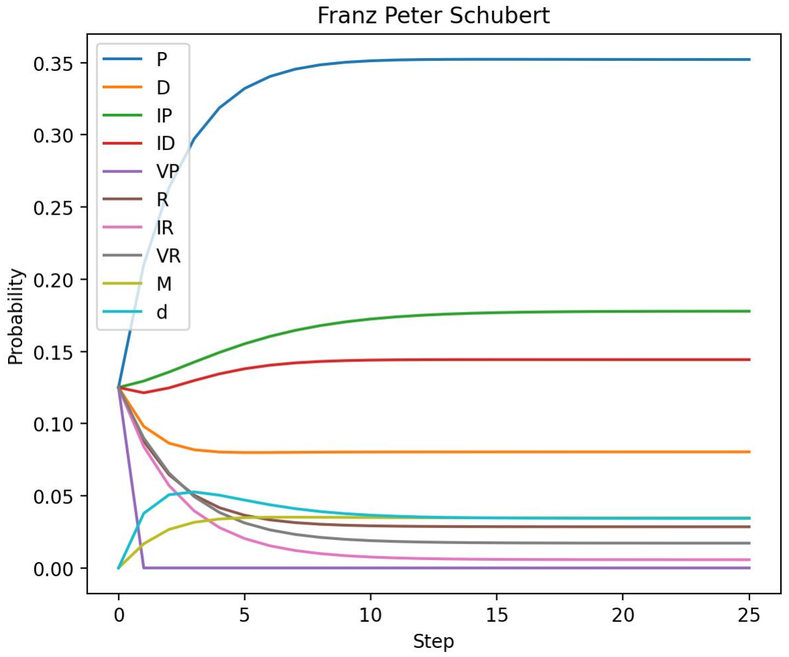


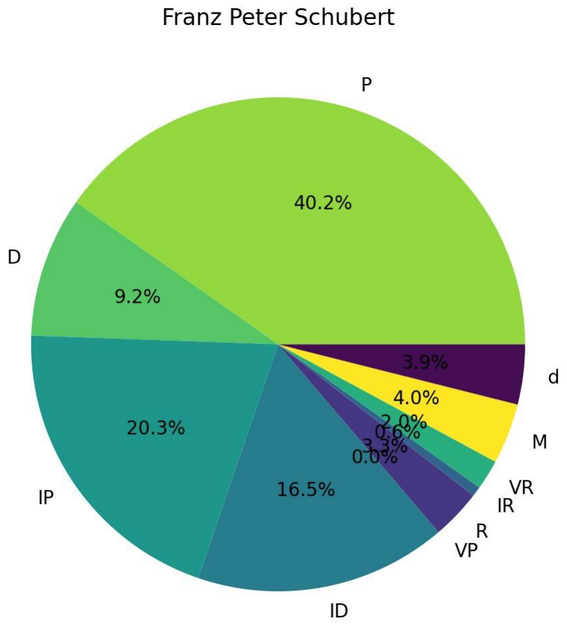




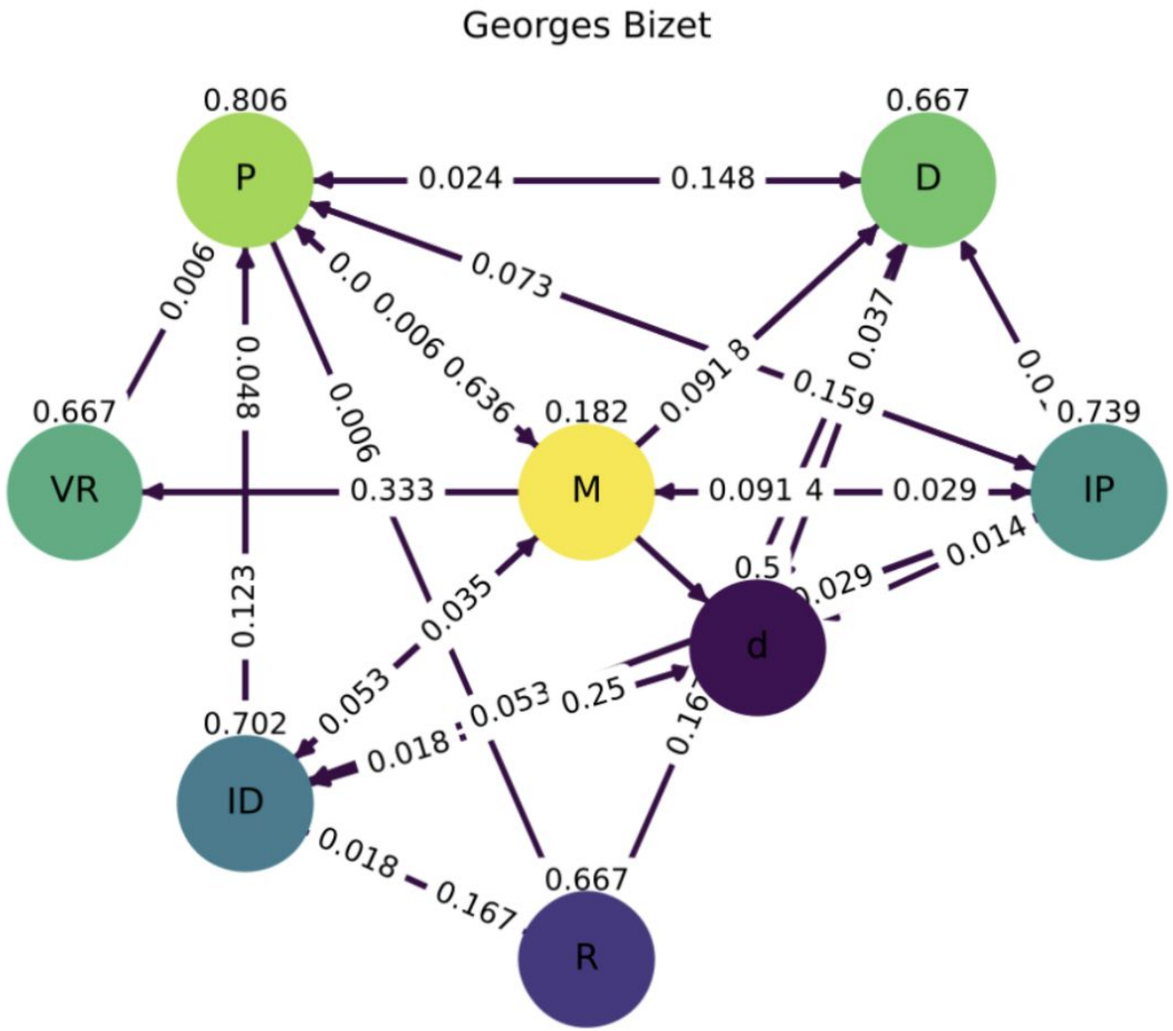
Section IV:
Results

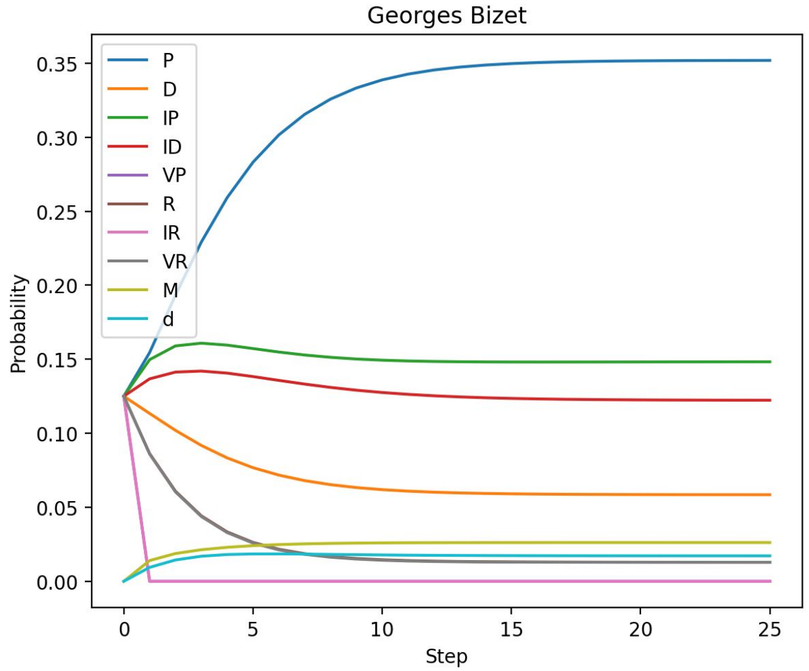


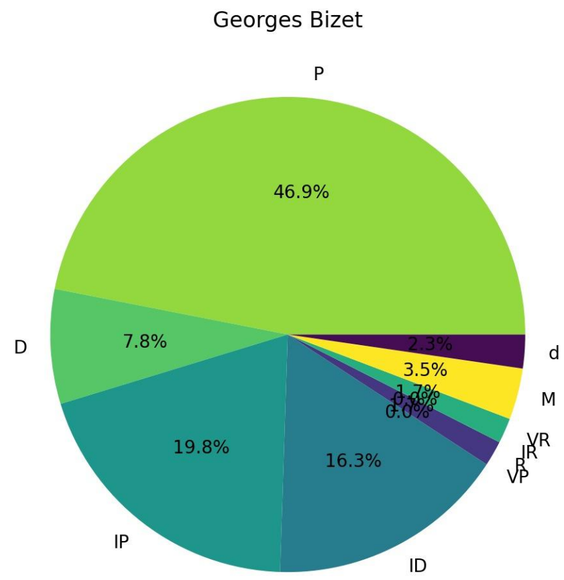




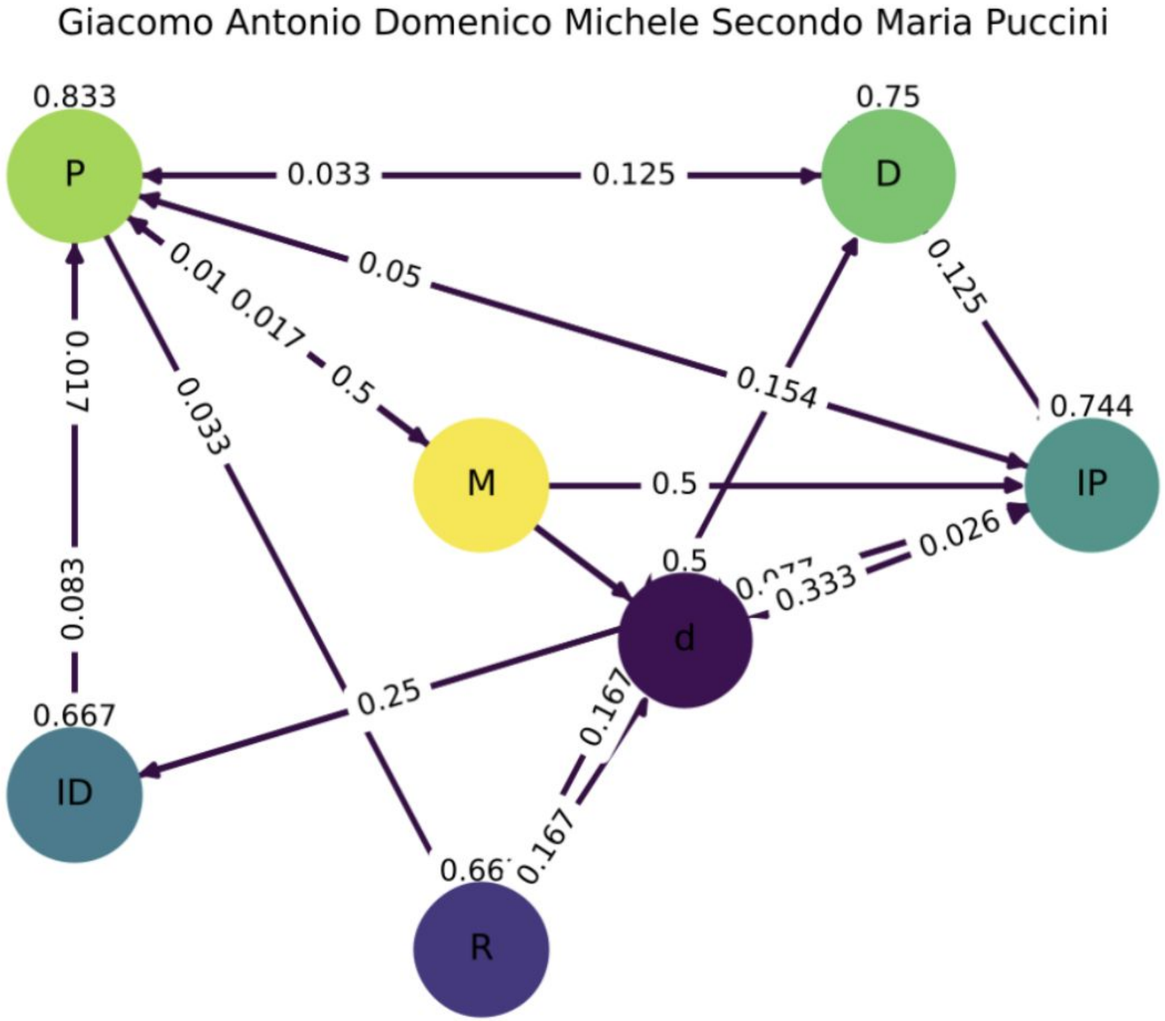
Section IV:
Results

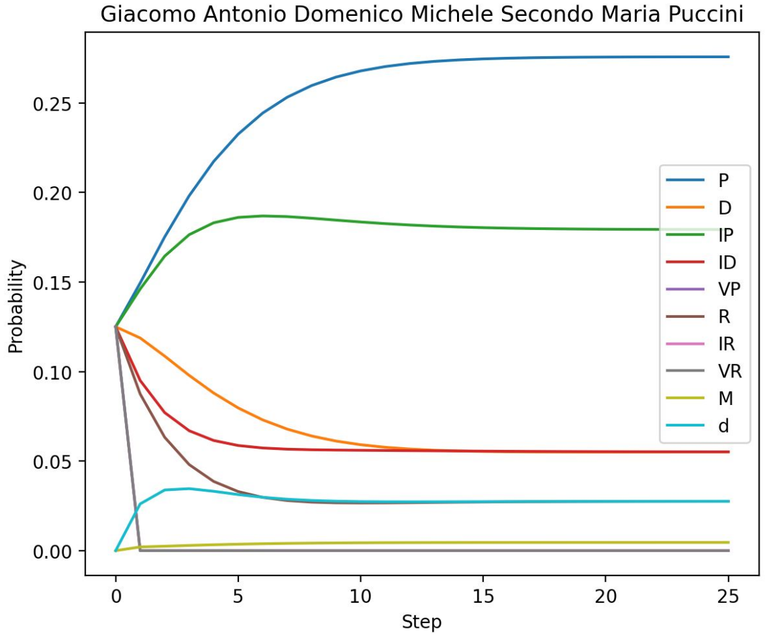


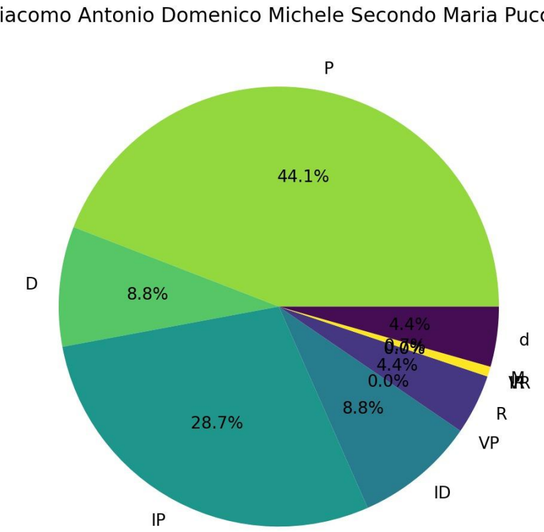




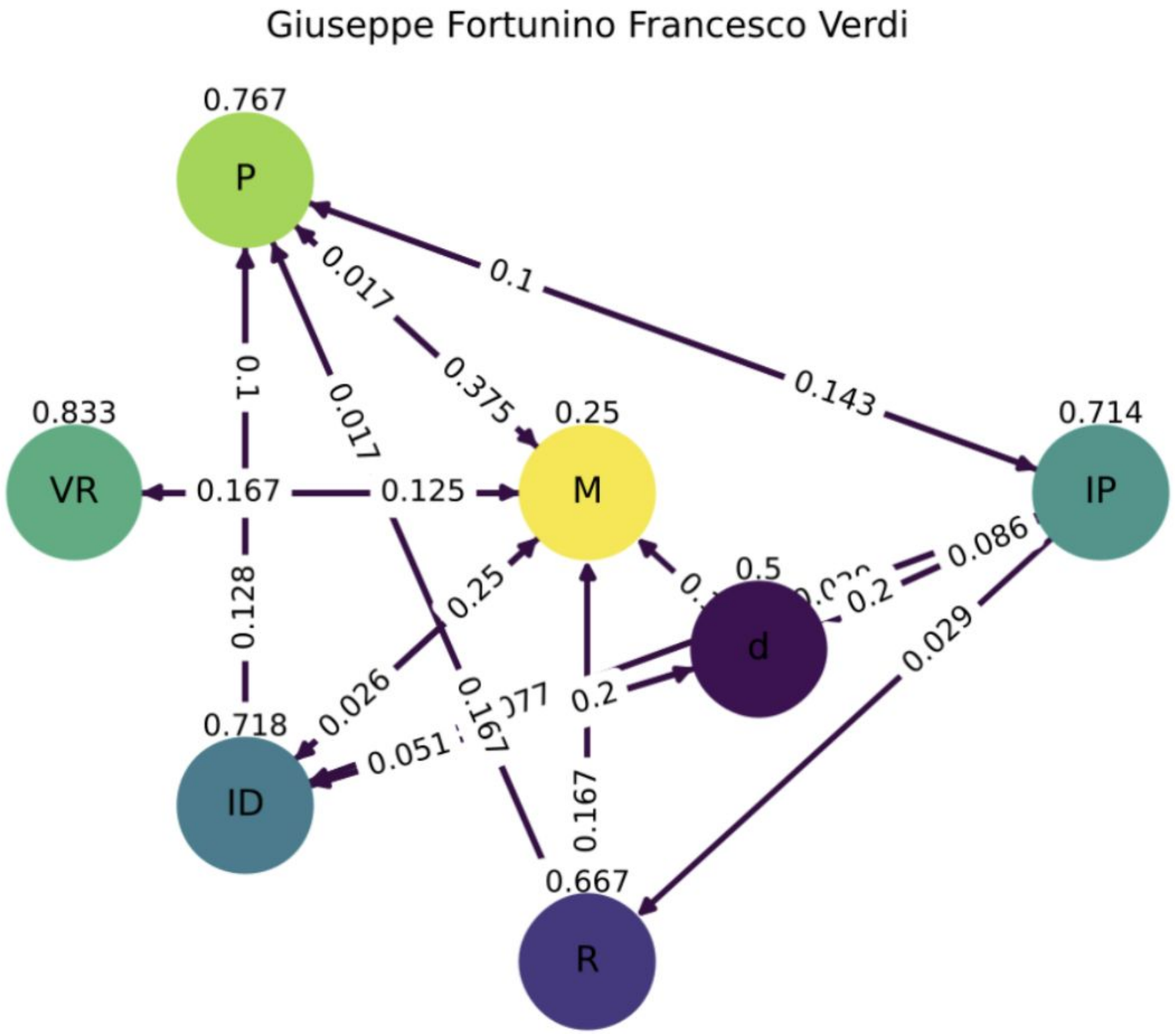
Section IV:
Results

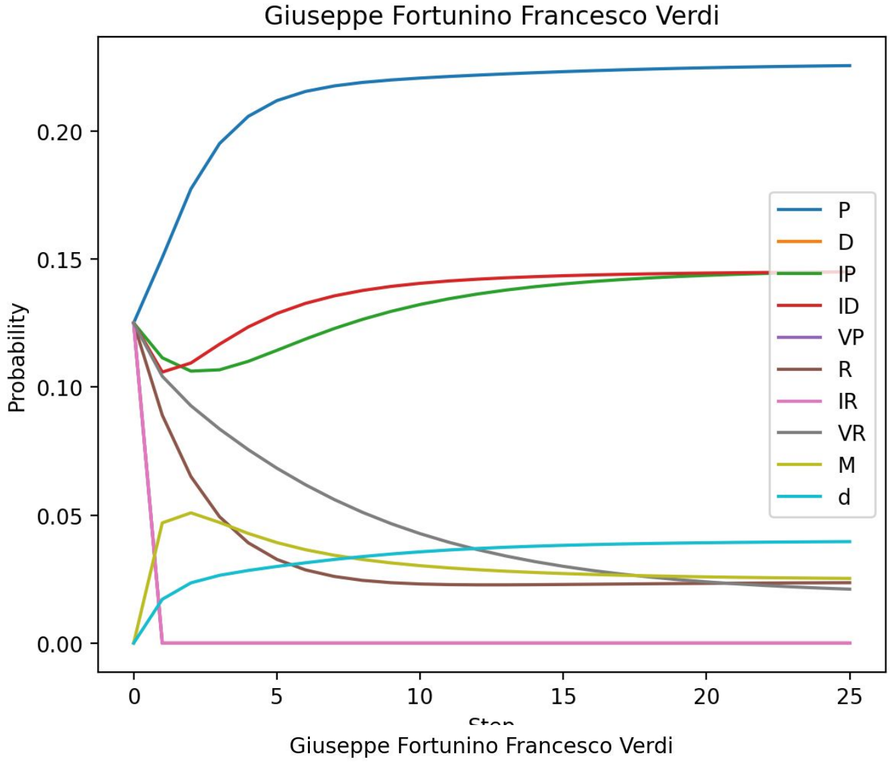


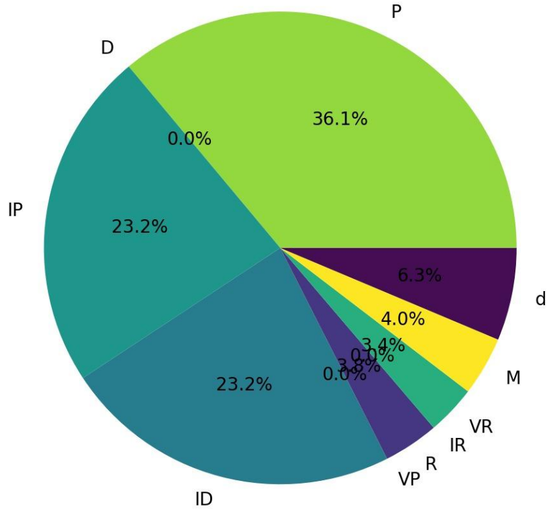




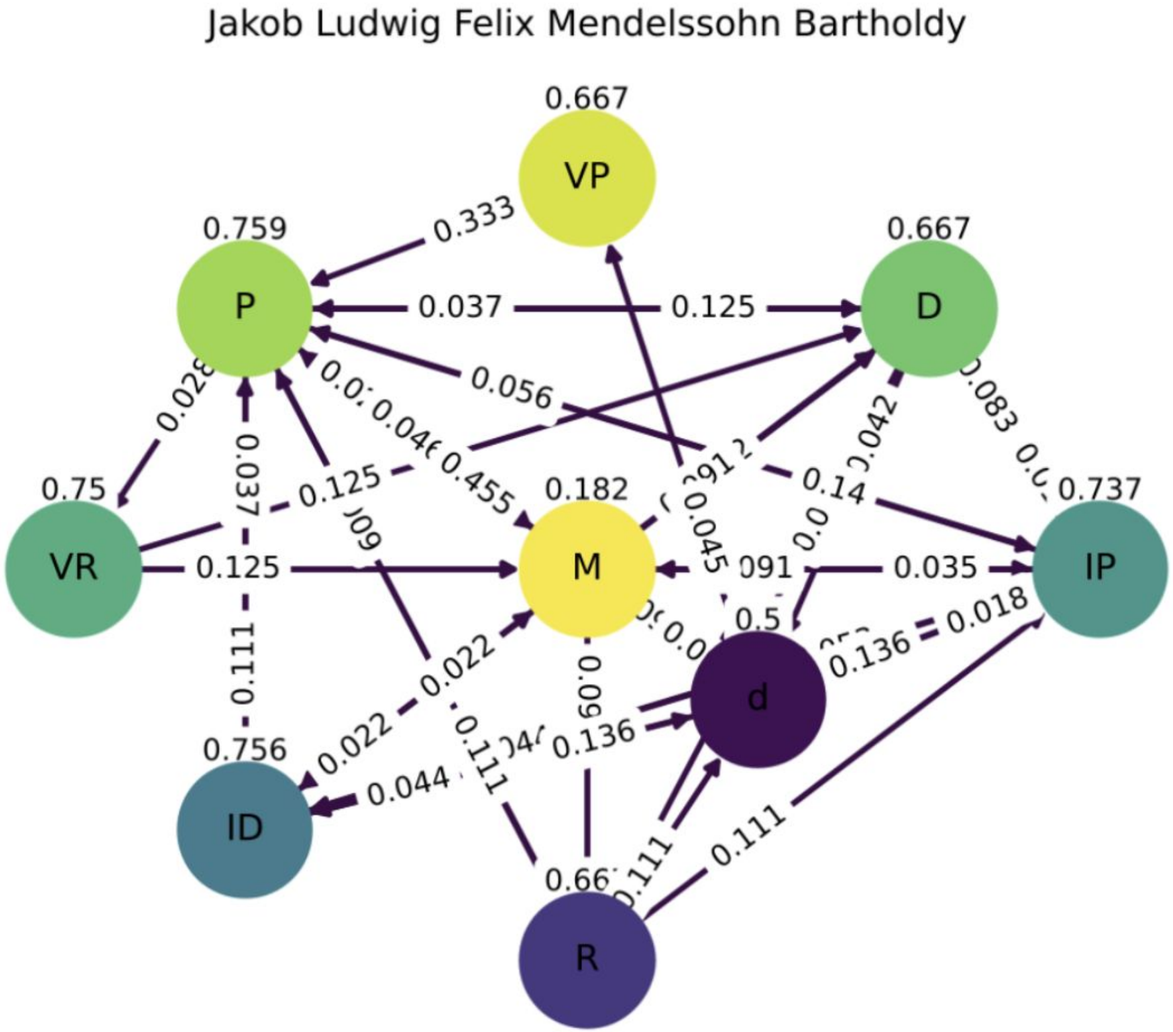
Section IV:
Results

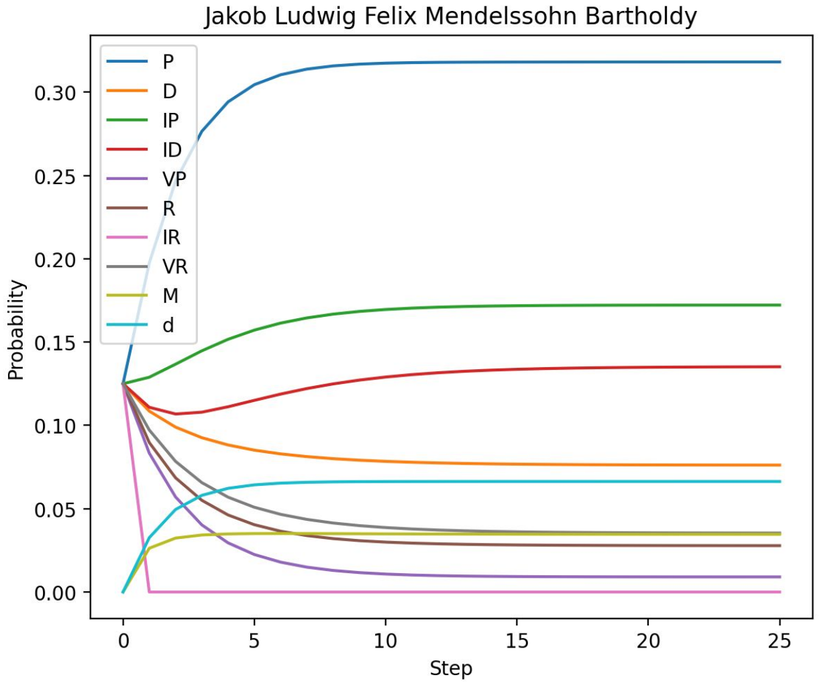


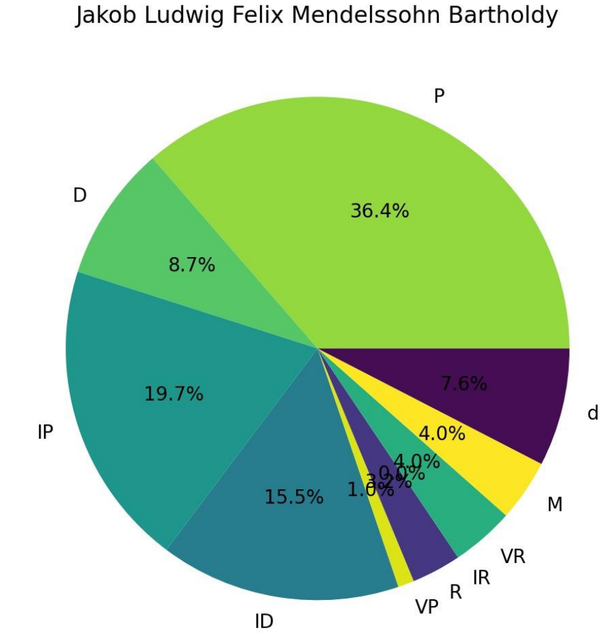




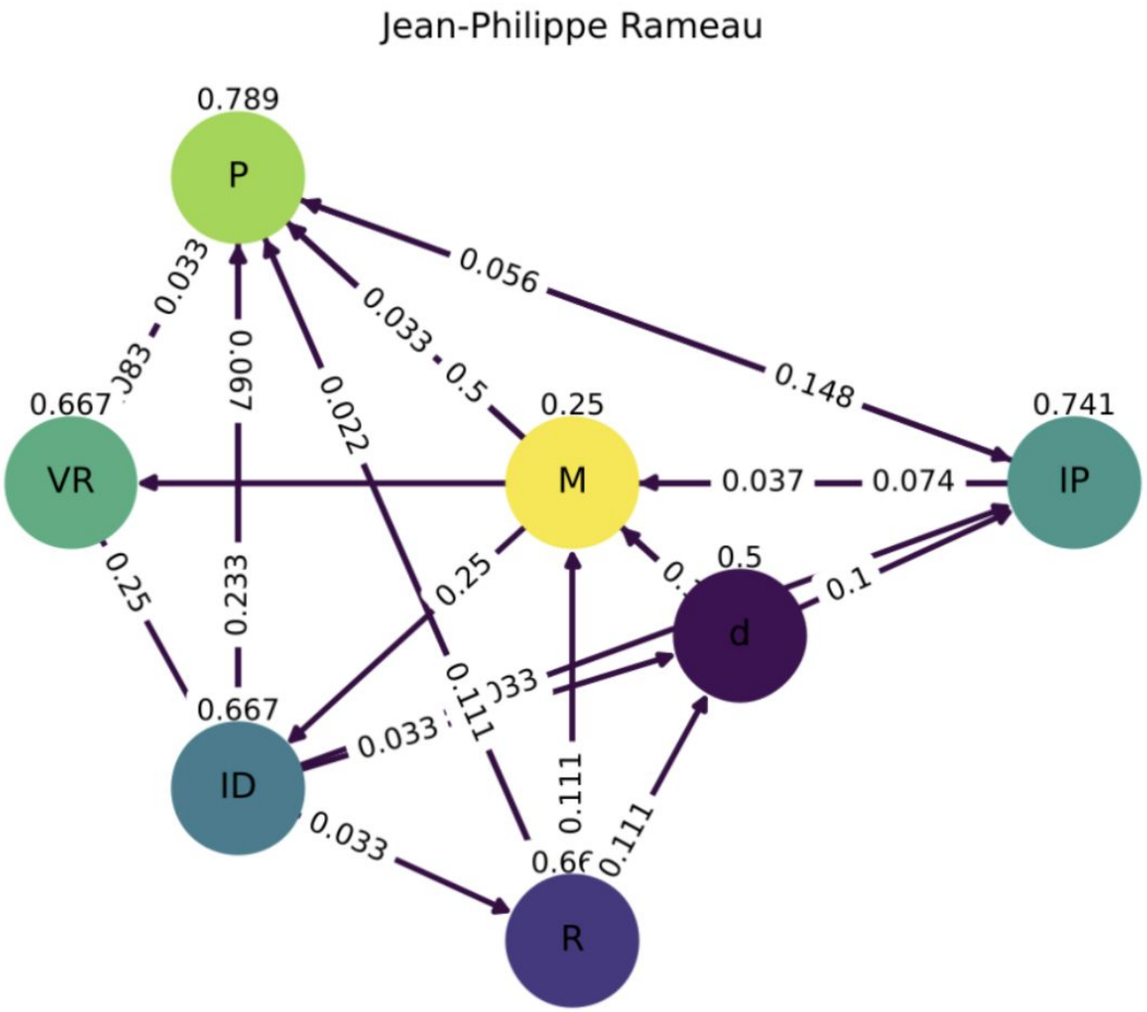
Section IV:
Results

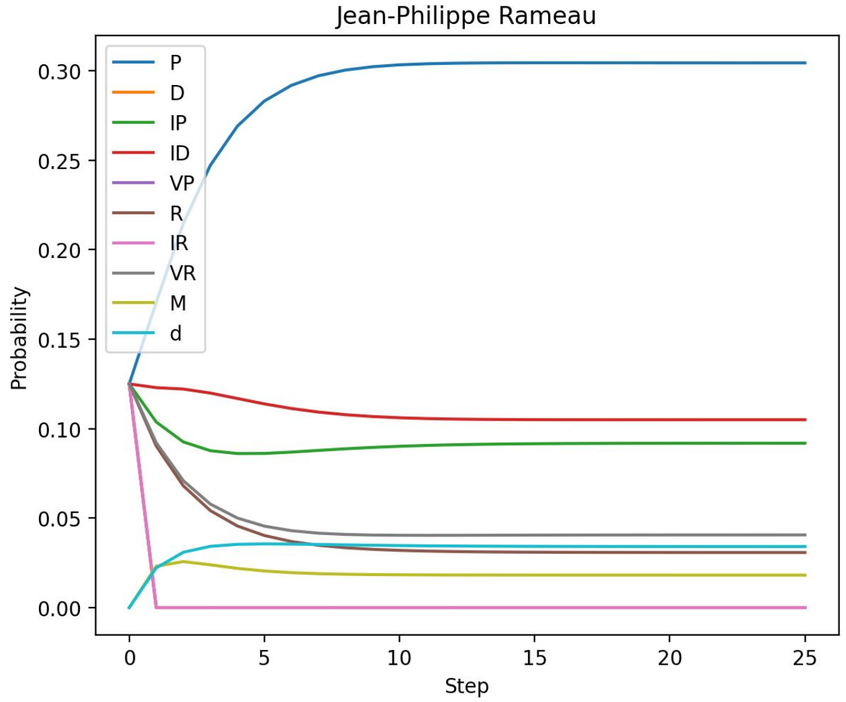


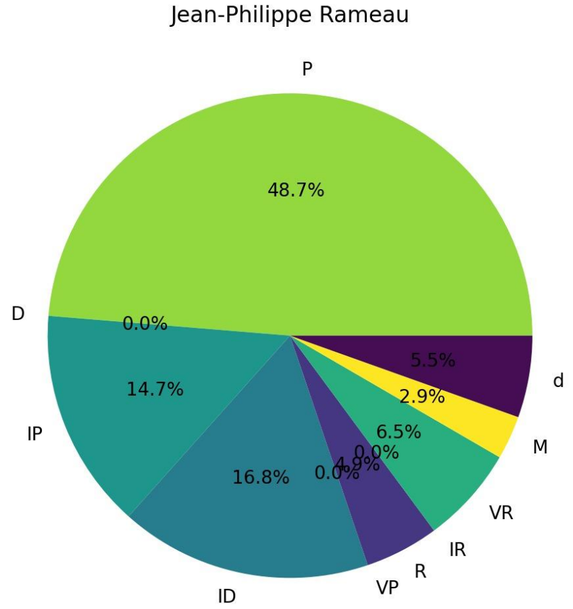




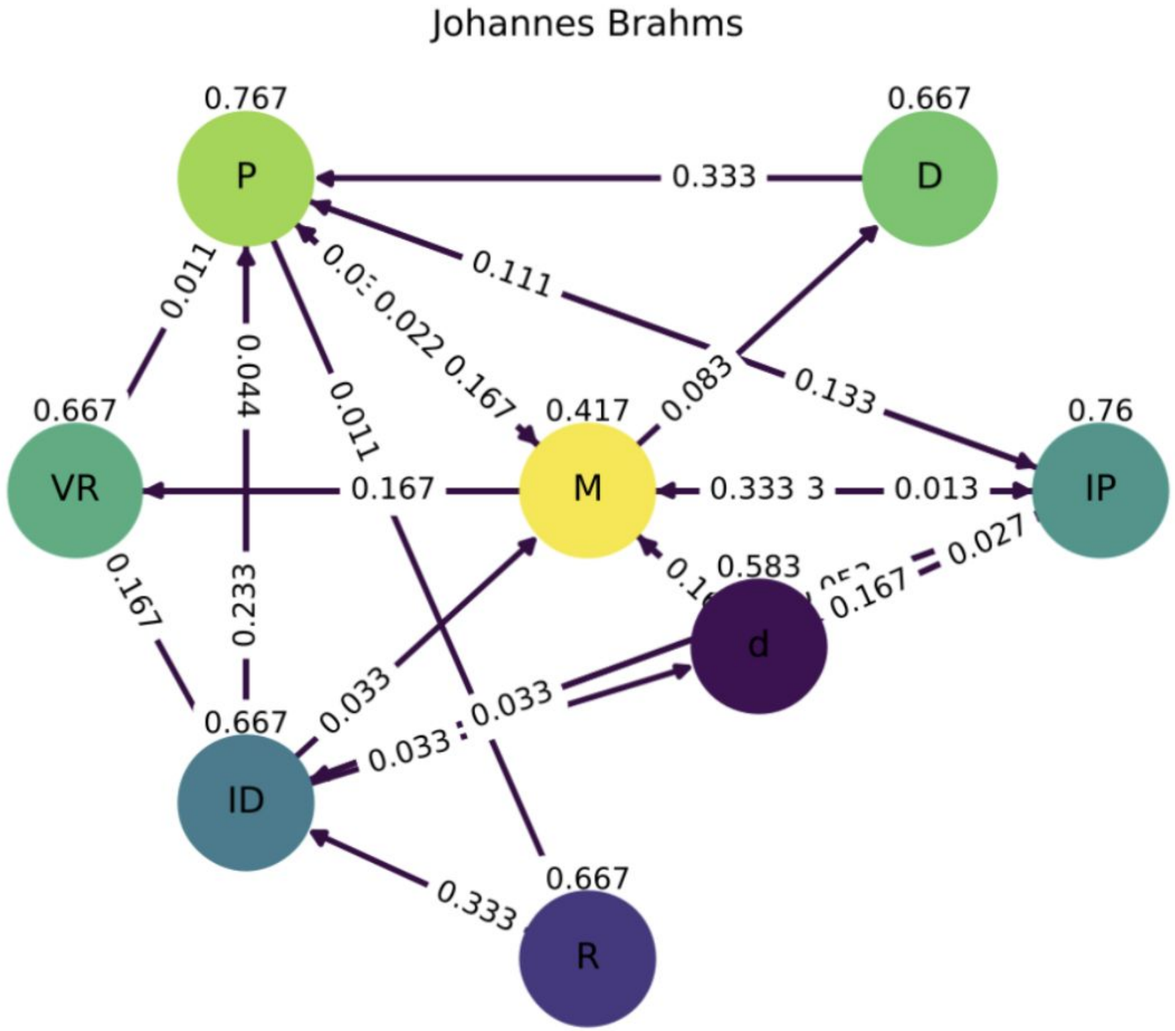
Section IV:
Results

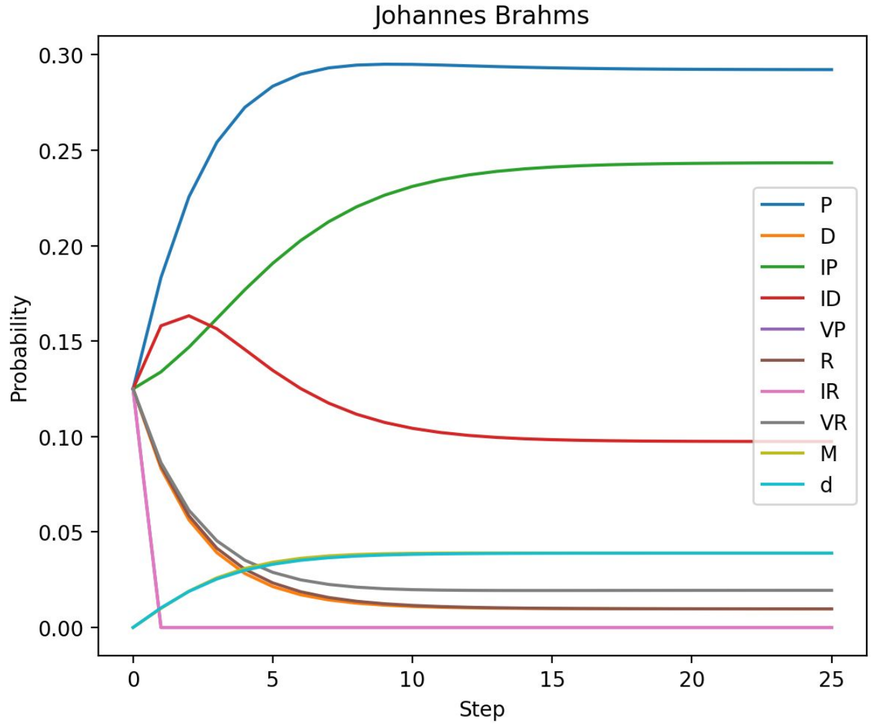


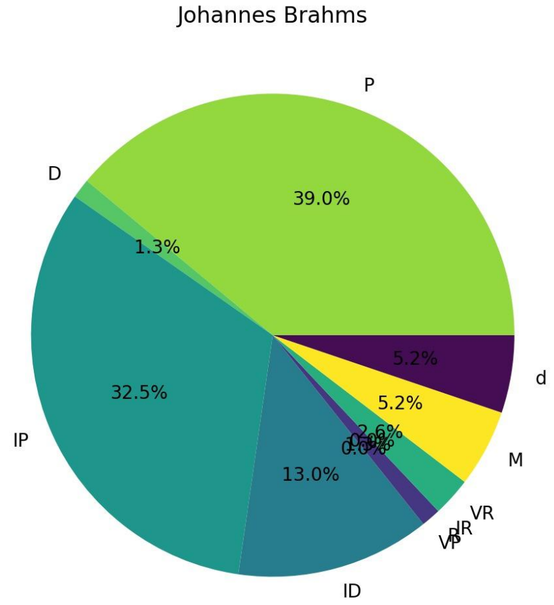




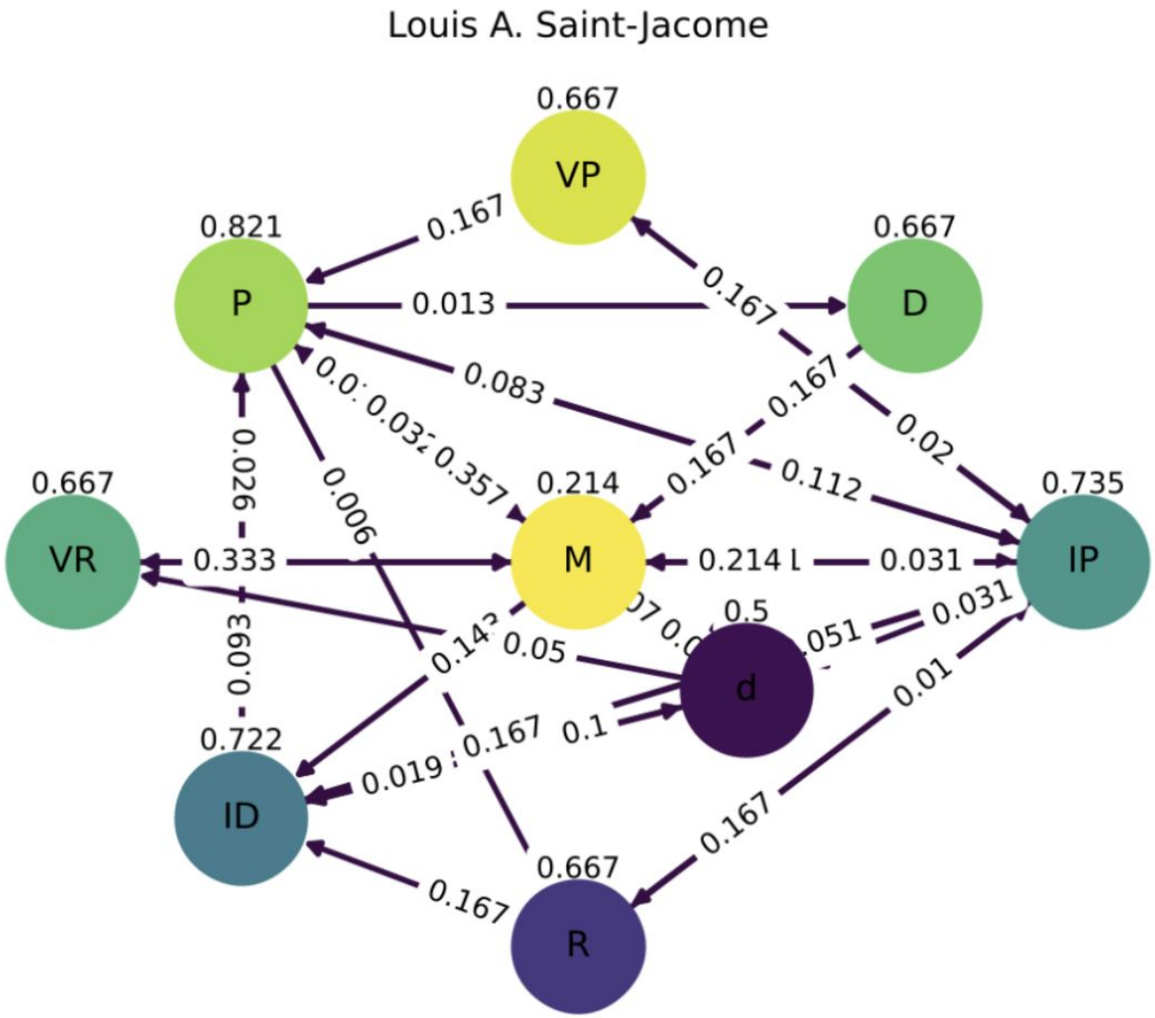
Section IV:
Results

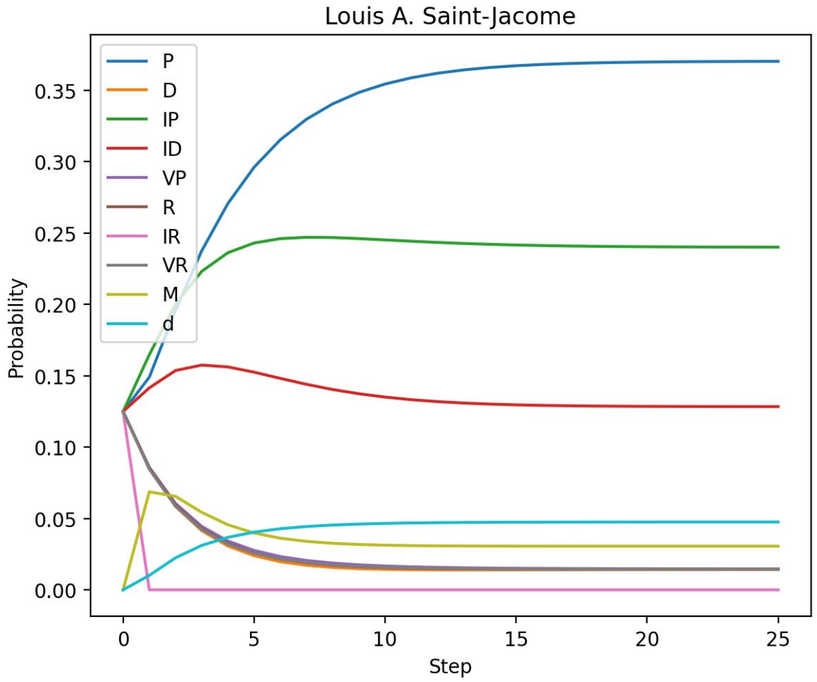


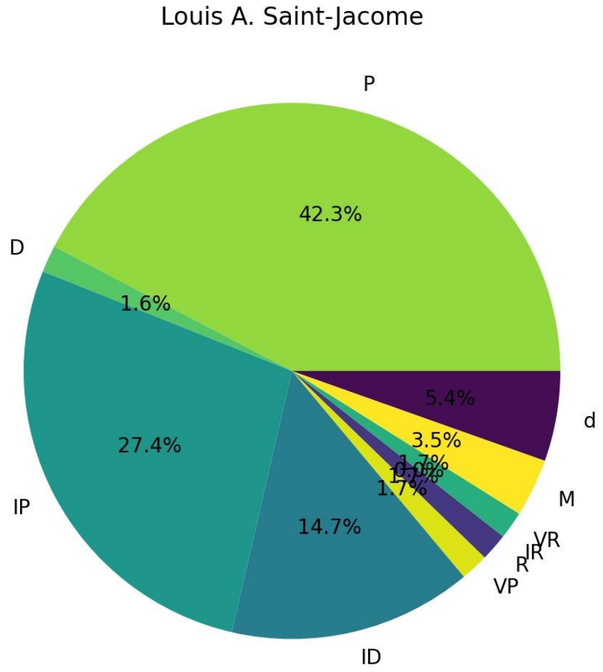




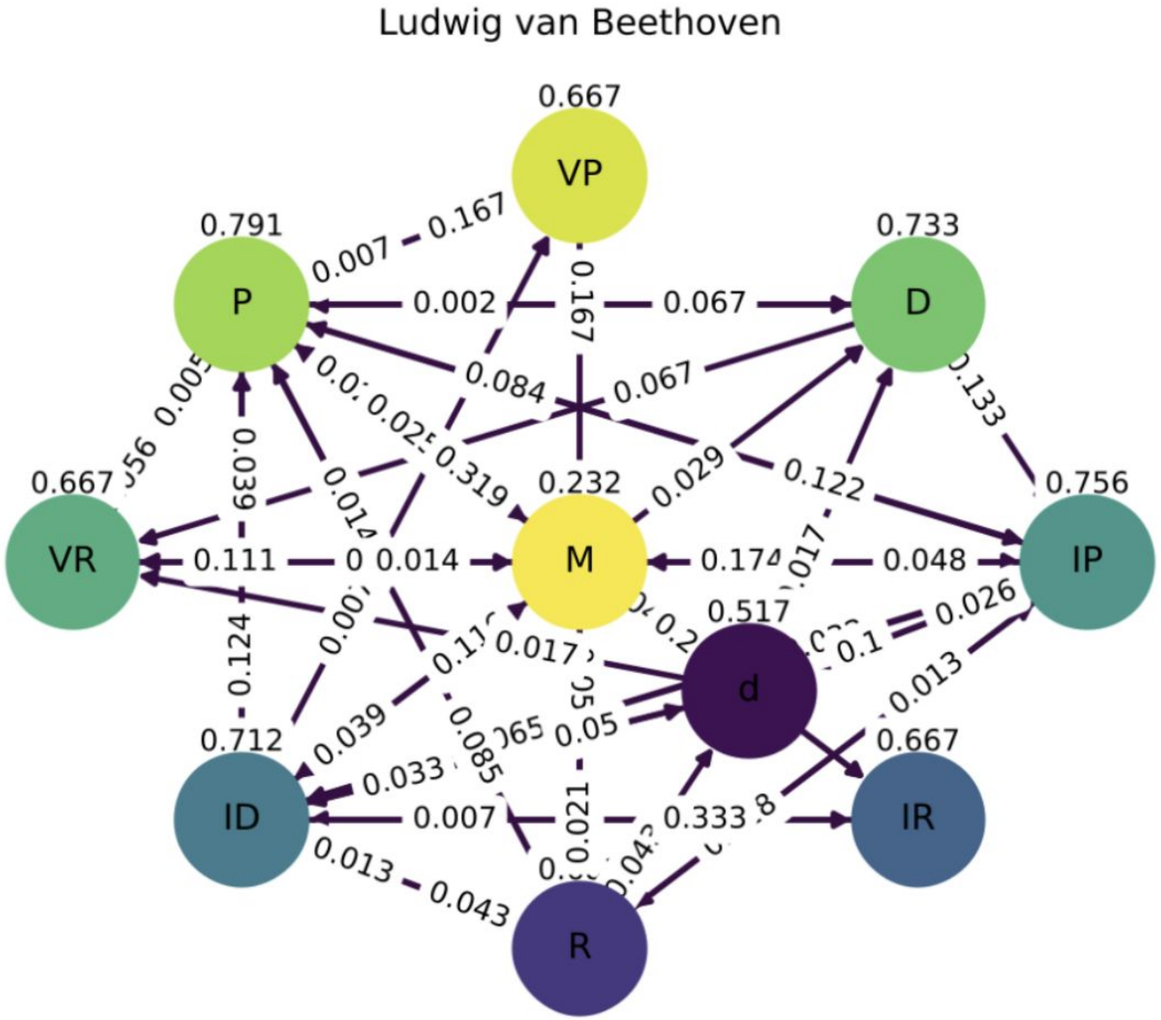
Section IV:
Results

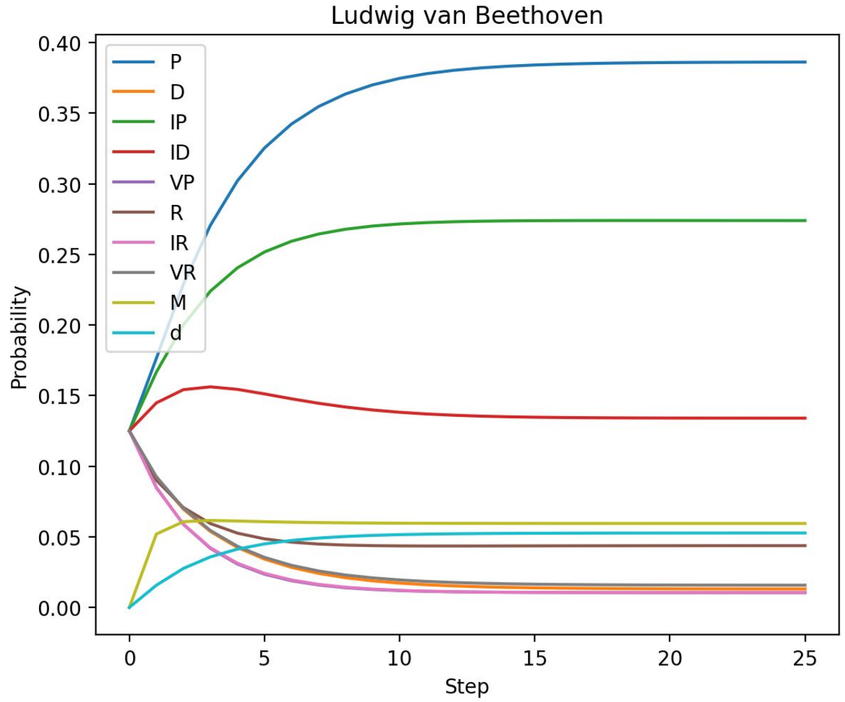


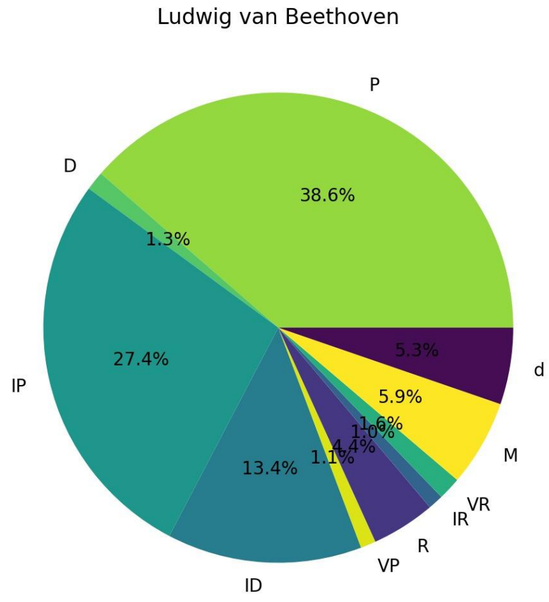




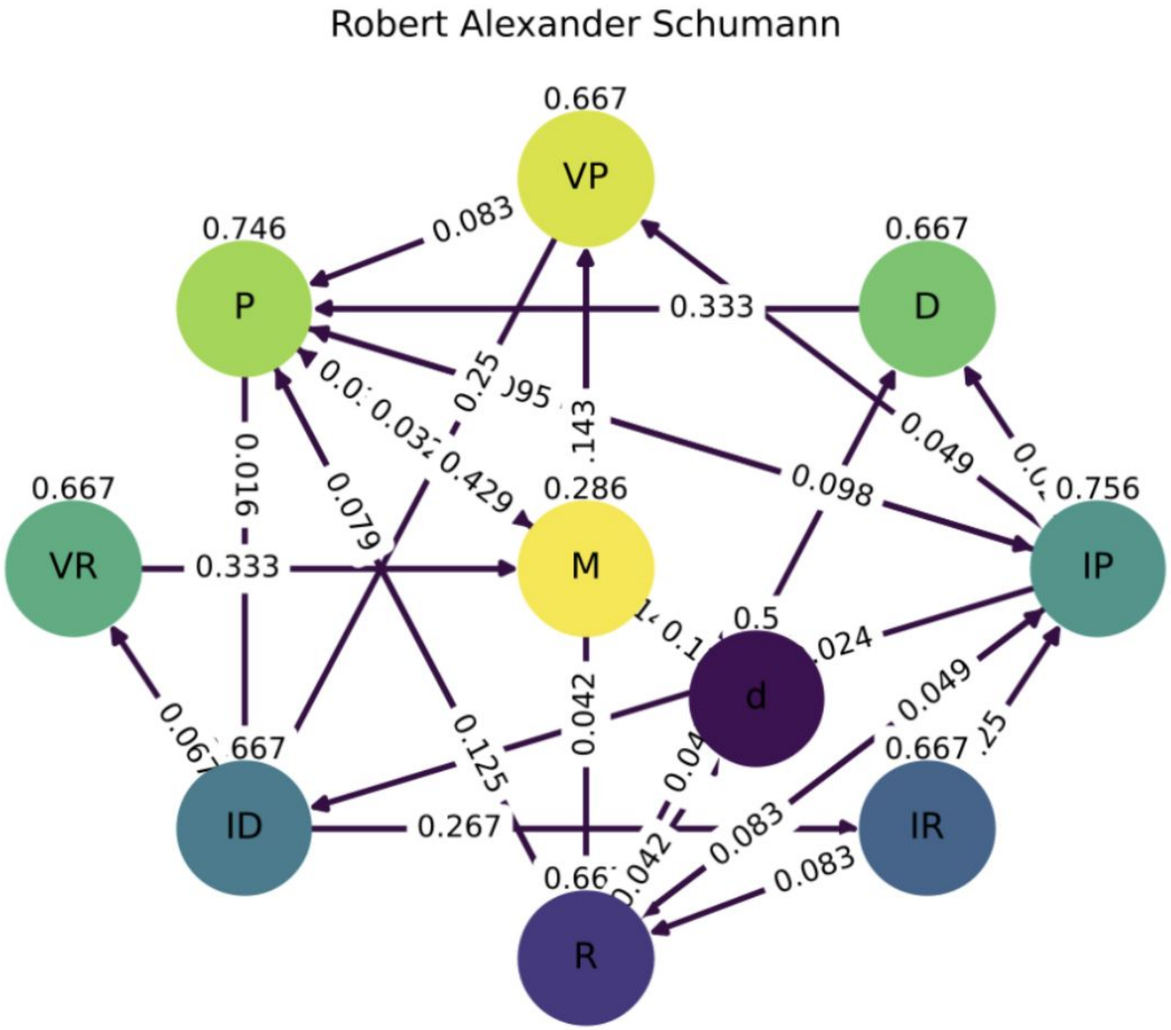
Section IV:
Results

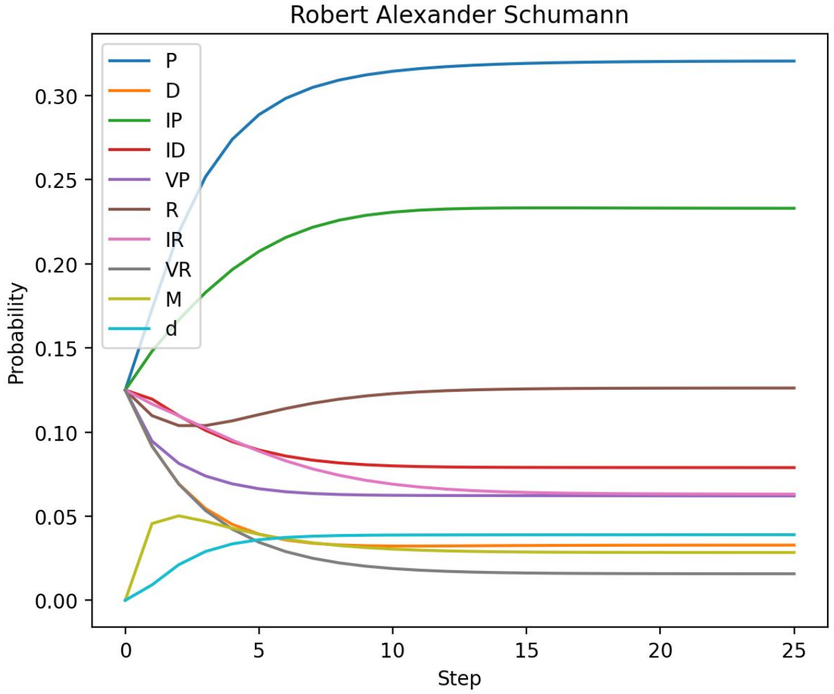


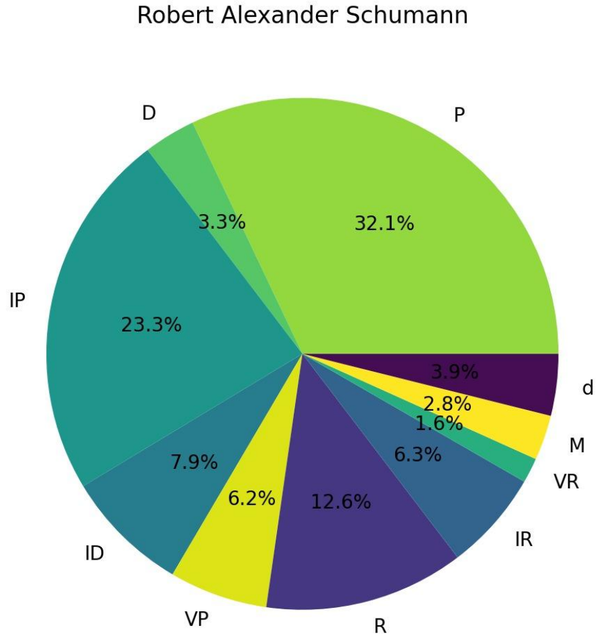




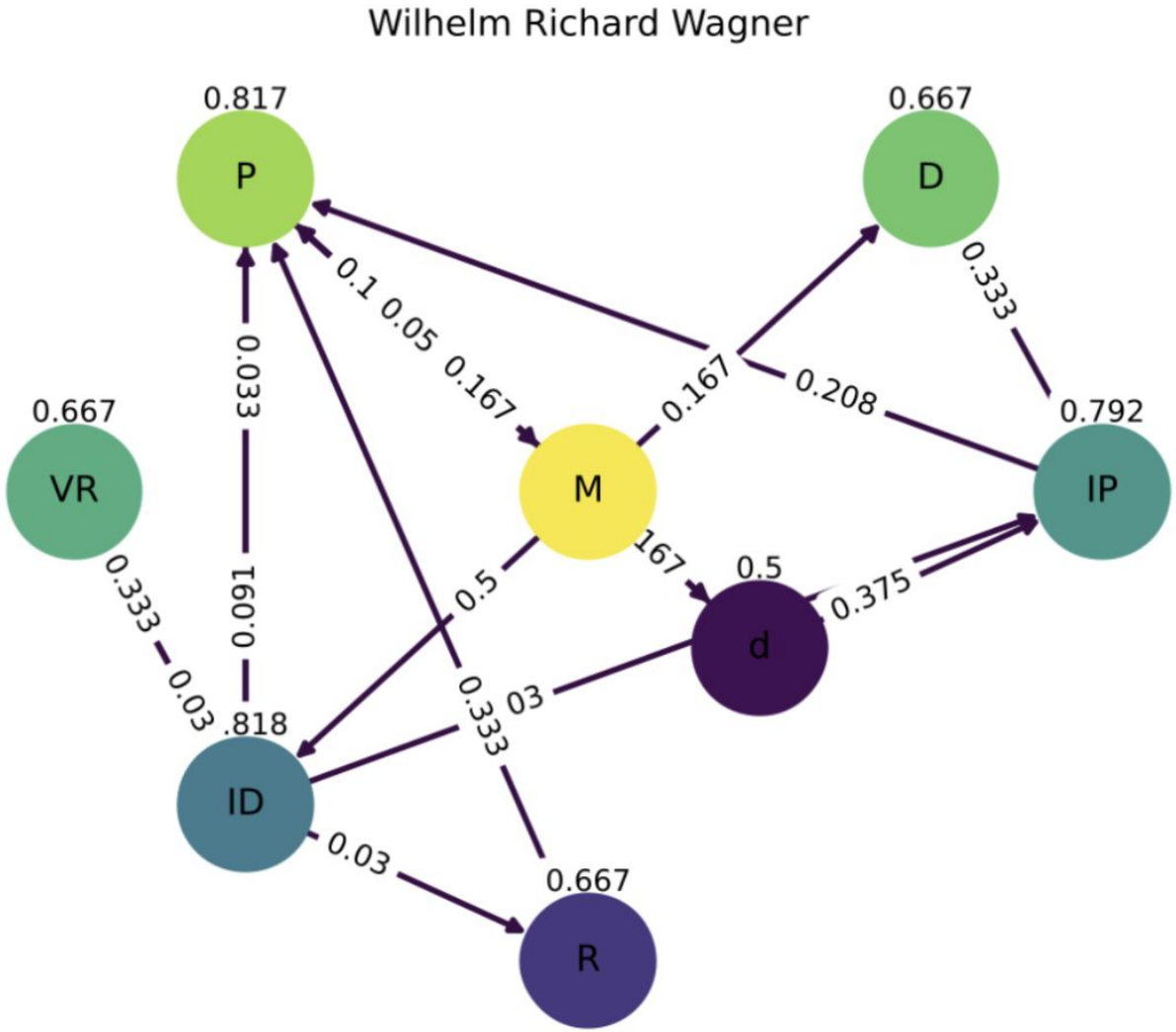
Section IV:
Results

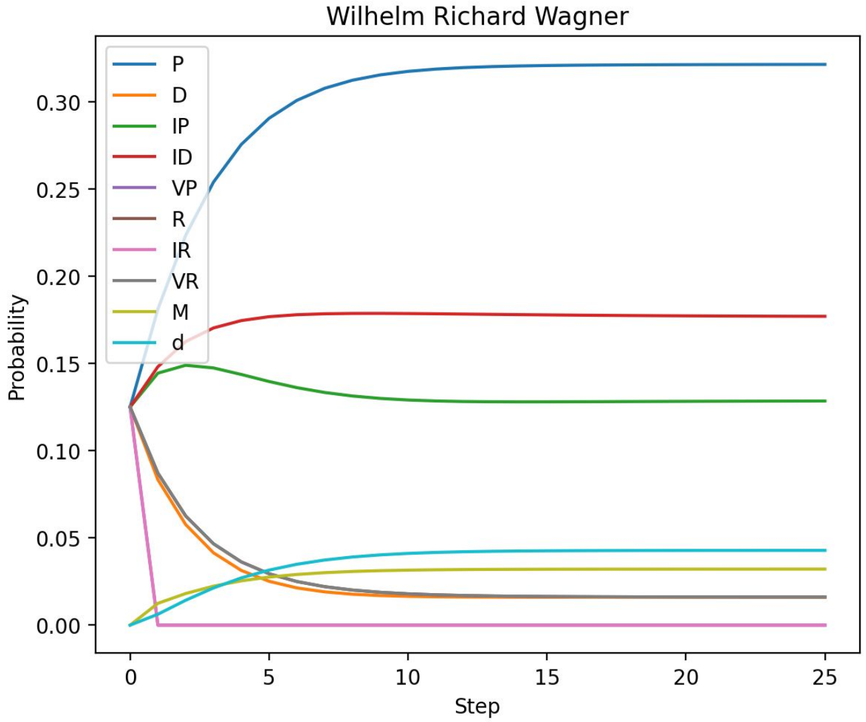


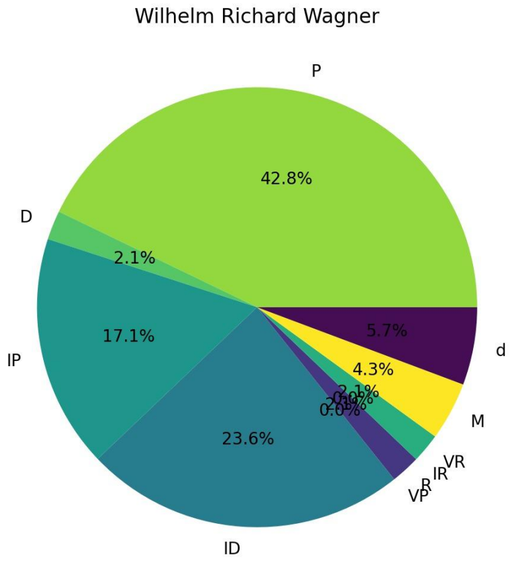




Section IV:
Results







Section IV:
Results