Fear Generalization

Antoin Sader

Antoine Nasra

Supervised by: Professor Giuseppe Boccignone

Department of Computer Science,

University Of Milan

November 2024

Contents

Introduction

Fear generalization is the process by which fear responses learned from one stimulus extend to similar stimuli. While this process is adaptive in many contexts, overgeneralization is a hallmark of anxiety-related disorders, including post-traumatic stress disorder (PTSD), generalized anxiety disorder (GAD), and phobias. Understanding the mechanisms underlying fear generalization can provide insights into these disorders and inform therapeutic interventions.

Despite its importance, individual differences in fear generalization remain poorly understood. Factors such as sensitivity to stimuli, generalization gradients, and extinction learning are thought to contribute to these differences, but their interactions and relative importance are not yet fully known.

The study by Tuerlinckx et al. addresses this gap by using a probabilistic graphical model (PGM) to investigate the latent mechanisms driving fear generalization. Their model incorporates both observed variables, such as behavioral responses to conditioned and generalized stimuli, and latent variables, such as sensitivity and noise parameters. Bayesian inference, implemented through Markov Chain Monte Carlo (MCMC) methods, is used to estimate these parameters.

The goal of this report is to replicate the findings of Tuerlinckx et al. using their provided data and R/JAGS code, and to port the model to Python using PyMC. In addition, we aim to explore potential extensions of the model using alternative datasets or experiments.

This report is structured as follows: Section 1 provides the theoretical background, detailing fear generalization and the PGM framework. Section 2 describes the methodology, including data preparation and code translation. Section 3 presents the results and their validation. Finally, Section 4 discusses the findings and outlines potential future directions.

1 Theoretical Background

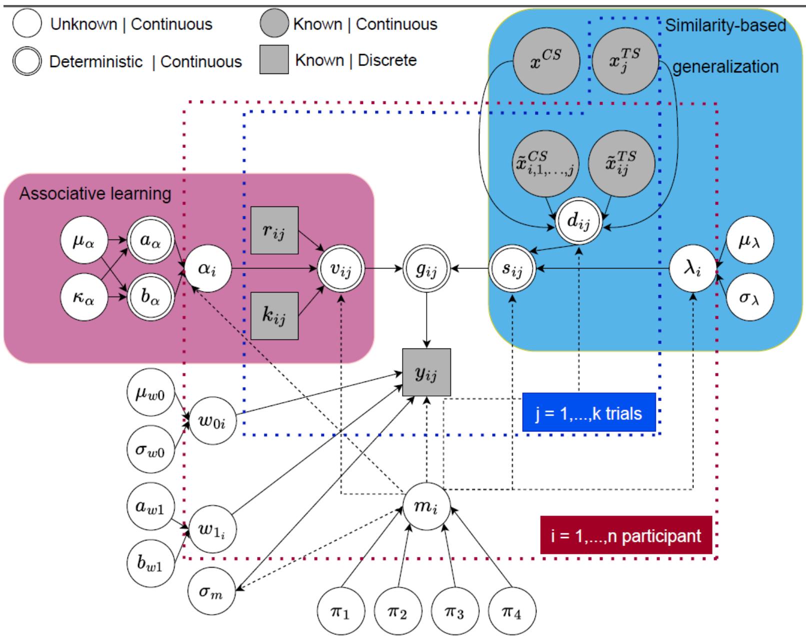
1.1 Fear Generalization

Definition and Importance. Fear generalization is a psychological process where fear responses learned from one stimulus extend to other similar, but non-threatening, stimuli. This phenomenon plays a critical role in adaptive survival behaviors by enabling organisms to avoid potential dangers based on prior experiences. However, excessive fear generalization is a hallmark of anxiety-related disorders such as post-traumatic stress disorder (PTSD), generalized anxiety disorder (GAD), and phobias.

Mechanisms of Fear Generalization. Several mechanisms contribute to fear generalization:

- *Associative Learning*: Fear responses are typically acquired through classical conditioning, where a neutral stimulus (conditioned stimulus, CS) is paired with an aversive event (unconditioned stimulus, US). Generalization occurs when similar stimuli (generalization stimuli, GS) elicit the same response due to their resemblance to the CS.
- *Generalization Gradient*: The generalization gradient describes how the intensity of the fear response decreases as the GS becomes less similar to the CS. Steeper gradients indicate narrower generalization, while flatter gradients suggest broader generalization.
- *Extinction Learning*: Extinction involves the gradual reduction of fear responses when the CS is repeatedly presented without the US. Impaired extinction can lead to persistent or exaggerated generalization.
- *Individual Differences*: Factors such as anxiety levels, attention biases, and cognitive flexibility influence how individuals generalize fear.

Clinical Relevance. Understanding fear generalization is critical for developing effective treatments for anxiety-related disorders. For instance, therapeutic interventions such as exposure therapy and cognitive-behavioral therapy (CBT) aim to reduce maladaptive generalization by improving extinction learning and modifying attention biases.



Mathmatical relationship

$$y_{ij} \sim \mathcal{N}\left(1 + \frac{9}{1 + e^{-(w_{0_i} + w_{1_i} g_{ij})}}, \sigma^2_{m_i}\right)$$

$$g_{ij} = v_{ij} s_{ij}$$

$$v_{ij+1} = \begin{cases} 0 & \text{for } m_i = 1, \\ v_{ij} + \alpha_i (r_{ij} - v_{ij}) k_{ij} & \text{otherwise.} \end{cases}$$

$$s_{ij} = \begin{cases} 1 & \text{for } m_i = 1 \text{ or } v_{ij} = 0, \\ e^{-\lambda_i d_{ij}} & \text{for } m_i = 2, 3, 4. \end{cases}$$

$$d_{ij} = \begin{cases} \mid x^{CS} - x_j^{TS} \mid & \text{for } m_i = 2, 3, \\ \mid \tilde{x}^{CS}_{i,1,...,j} - \tilde{x}^{TS}_{ij} \mid & \text{for } m_i = 4. \end{cases}$$

Figure 1: Directed Acyclic Graph (DAG) representation of the probabilistic graphical model for fear generalization.

1.2 Probabilistic Graphical Models (PGMs)

Overview. Probabilistic graphical models (PGMs) are mathematical frameworks that represent the relationships between variables using a graph structure. In PGMs:

- *Nodes*: Represent variables, which can be observed (e.g., fear responses) or latent (e.g., sensitivity to stimuli).
- *Edges*: Indicate how variables are probabilistically related to each other.

In this study, PGMs are used to model the relationships between latent mechanisms (e.g., sensitivity to threat, group assignments) and observed behavioral responses to stimuli.

DAG for Fear Generalization. Figure 1 depicts the probabilistic graphical model from Tuerlinckx et al.'s study. This DAG shows how variables interact in the model, highlighting relationships such as associative learning, similarity-based generalization, and group-level differences.

Key Components of the Model.

- *Observed Variables*:
  - y_{ij}: Fear response of participant i during trial j, which is influenced by individual latent traits, stimulus properties, and noise.

- *Latent Variables*:
  - $\alpha_i, \lambda_i, w_{1i}$: Participant-specific traits, such as sensitivity to stimuli, generalization tendencies, and noise parameters.
  - m_i: Latent group assignment, indicating whether the participant is a specific or broad generalizer.
- *Known Variables and Stimulus Properties*:
  - x^{CS}, x^{TS}: Properties of the conditioned stimulus (CS) and test stimuli (TS).
  - k_{ij}, r_{ij}: Experimental variables influencing responses.

Relationships in the Model. The DAG organizes relationships into three main components:

- *Associative Learning (Pink Box)*:
  - Captures how responses are adjusted trial-by-trial based on the participant's learning parameters $(\alpha_i, r_{ij}, k_{ij})$.
- *Similarity-Based Generalization (Blue Box)*:
  - Describes how the similarity $(\text{sim}(s, \text{CS}))$ between the test stimulus and conditioned stimulus affects fear responses.
- *Group-Level Differences (Red Box)*:
  - Models differences between participant groups, categorized by m_i into generalization profiles (e.g., narrow vs. broad generalizers).

Mathematical Representation. The relationships between variables in the DAG are formalized with these equations:

- *Observed Response Model*:

$$y_{ij} \sim \mathcal{N}\left(1 + \frac{9}{1 + e^{-(w_{0i} + w_{1i} g_{ij})}}, \sigma_m^2\right)$$

  This describes how the observed fear response (y_{ij}) depends on latent traits (w_{0i}, w_{1i}) and generalization factors (g_{ij}).

- *Generalization and Similarity Effects*:

$$g_{ij} = v_{ij} s_{ij}, \quad s_{ij} = \begin{cases} 1, & \text{if } m_i = 1 \text{ or } v_{ij} = 0 \\ e^{-\lambda_i d_{ij}}, & \text{if } m_i = 2, 3, 4 \end{cases}$$

where s_{ij} represents similarity effects and d_{ij} is the distance between the CS and TS.

- *Distance-Based Similarity*:

$$d_{ij} = \begin{cases} |x^{CS} - x^{TS}|, & \text{if } m_i = 2, 3 \\ |\tilde{x}^{CS}_{i,1,\ldots,j} - \tilde{x}^{TS}_{ij}|, & \text{if } m_i = 4 \end{cases}$$

  This defines how the physical or perceived distance between stimuli influences responses.

Inference in the Model. Using Bayesian inference, the model estimates:

- Posterior distributions of latent variables (z_i) to capture individual differences.
- Model parameters ($\alpha_i, w_{1i}, \lambda_i, \sigma_m$) for each participant.

These inferences provide insights into how fear generalization patterns vary across individuals and groups.

1.3 Bayesian Inference and MCMC

Bayesian Framework. Bayesian inference provides a systematic way to update our beliefs about model parameters (θ) after observing data. It is based on Bayes' theorem:

$$P(\theta|\text{data}) \propto P(\text{data}|\theta) \cdot P(\theta)$$

where:

- $P(\theta|\text{data})$: The posterior distribution, which represents updated beliefs about the parameters after observing the data.
- $P(\text{data}|\theta)$: The likelihood, describing the probability of the data given the parameters.
- $P(\theta)$: The prior distribution, encoding prior knowledge or assumptions about the parameters.

In the fear generalization model:

- *Parameters (θ)*: Include sensitivity (α_i), generalization gradient (λ_i), baseline response (w_{0i}), and noise (σ_m).

- *Likelihood*: Represents the probability of observing y_{ij} given the parameters and stimulus similarity (s_{ij}):

$$y_{ij} \sim \mathcal{N}\left(1 + \frac{9}{1 + e^{-(w_{0i} + w_{1i} g_{ij})}}, \sigma_m^2\right)$$

- *Posterior*: Combines the prior distributions and likelihood to infer the most probable values of the parameters.

Markov Chain Monte Carlo (MCMC). For complex models like this, direct computation of the posterior distribution is often infeasible. MCMC methods approximate the posterior by generating samples from it.

Gibbs Sampling. Gibbs sampling, a specific MCMC algorithm, is used to sample each parameter iteratively from its conditional distribution:

- *Step 1:* Sample β_0 given current estimates of β_1, σ_m, and the data.
- *Step 2:* Sample β_1 given current estimates of β_0, σ_m, and the data.
- *Step 3:* Repeat for all parameters.

Prior Distributions. The priors for key parameters in the model are:

$$\alpha_i \sim \mathcal{N}(\mu_\alpha, \sigma_\alpha^2), \quad \lambda_i \sim \mathcal{N}(\mu_\lambda, \sigma_\lambda^2)$$

These priors incorporate prior knowledge about the parameters and ensure regularization.

Inference Goals. Using MCMC and Bayesian inference, the model estimates:

- Individual differences, such as broad vs. narrow generalizers, through posterior distributions of α_i and λ_i.
- The effect of stimulus similarity on fear responses, quantified through posterior estimates of sensitivity (w_{1i}).

1.4 Individual Differences in Fear Generalization

Understanding Individual Differences. Fear generalization varies across individuals, with some participants exhibiting specific responses only to the conditioned stimulus (CS), while others react broadly to similar stimuli (generalization stimuli or GS). These differences are driven by latent psychological mechanisms, including:

- *Sensitivity to Stimuli* (α_i): The strength of a participant's reaction to threatening stimuli.
- *Generalization Gradient* (λ_i): The breadth of fear generalization, indicating whether fear responses extend narrowly or broadly to similar stimuli.
- *Group Membership* (m_i): Categorizes participants into distinct generalization profiles, such as specific responders or broad generalizers.

How the Model Captures Individual Differences. The probabilistic graphical model (PGM) in Figure 1 incorporates individual-specific parameters:

- *Latent Variables* (m_i, α_i, λ_i): Participant-specific traits that influence observed fear responses.
- *Observed Variables* (y_{ij}): Fear responses recorded during trials, which reflect both latent traits and stimulus properties.

Using Bayesian inference, the model estimates these parameters, capturing how participants generalize their fear across trials and stimuli.

Linking to the Experiment.

- *Data Collection*: For each participant (i), multiple trials (j) are conducted. Observed data include fear responses (y_{ij}), conditioned stimulus properties (x^{CS}), and test stimulus properties (x^{TS}).
- *Modeling Responses*: The likelihood of y_{ij} is modeled based on participant-specific parameters (α_i, λ_i), trial-specific variables (v_{ij}, s_{ij}), and group membership (m_i).
- *Group-Level Differences*: Participants are categorized into:
  - **Non-Learners**: Participants who do not display any associative learning or generalization, showing flat response patterns across all stimuli..
  - **Overgeneralizers**: Participants who generalize fear or responses broadly to all stimuli, showing minimal differentiation between CS+ and other stimuli.
  - **Physical Generalizers**: Participants whose responses decline sharply based on the physical similarity of stimuli to CS+, indicating generalization rooted in physical features.
  - **Perceptual Generalizers**: Participants whose responses decline gradually, generalizing based on perceptual similarity (e.g., subjective or inferred similarity) rather than strict physical attributes.

Mathematical Representation.

$$y_{ij} \sim \mathcal{N}\left(1 + \frac{9}{1 + e^{-(w_{0i} + w_{1i} g_{ij})}}, \sigma_m^2\right)$$

where:

- $g_{ij} = v_{ij} s_{ij}$, and s_{ij} represents the similarity effect:

$$s_{ij} = \begin{cases} 1, & \text{if } m_i = 1 \text{ or } v_{ij} = 0 \\ e^{-\lambda_i d_{ij}}, & \text{if } m_i = 2, 3, 4 \end{cases}$$

- d_{ij}: Distance between the CS and GS.

Insights from the DAG. The DAG in Figure 1 demonstrates how individual differences $(m_i, \alpha_i, \lambda_i)$ influence fear responses (y_{ij}) through trial-specific variables (v_{ij}, s_{ij}) and similarity-based effects.

Clinical Implications.

- *Personalized Interventions*: Identify participants with maladaptive generalization (e.g., overgeneralizers) for targeted therapies.
- *Predicting Response Patterns*: Use posterior distributions of sensitivity (α_i) and generalization gradients (λ_i) to predict responses to novel stimuli.

2 Methods

we will clean the two datasets used in the paper Humans display interindividual differences in the latent mechanisms underlying fear generalization behaviour. We will also generate the required data structures for modeling and create some visualizations to better understand the fundamental structure of the data.

2.1 Data Clean

2.1.1 a. Learning Phase

Participants learn the relationship between:

- **Conditioned stimulus (CS+):** (e.g., image of a triangle)

- **Unconditioned stimulus (US):** (e.g., shocks)

Objective: To make participants expect the unconditioned stimulus (US) when presented with the conditioned stimulus (CS+).

2.1.2 b. Generalization Phase

- After learning the association, participants are shown stimuli similar to the original CS+ (e.g., triangles of different colors).
- **Goal:** To determine whether participants generalize their fear to these similar stimuli or respond only to the original CS+.

2.1.3 Variables

- **US (0/1)**: Indicates whether the US (unconditioned stimulus) is present or not.
- **Per-size**: Participant's perception of the size of the stimulus.
- **US-expectation (y)**: Expectation of a shock (between 0% and 10%).
- **Shock Presence (r)**: Indicates whether a shock was actually delivered (0: No shock, 1: Shock delivered).
- **Stimulus Type**:
  - **CS+**: Original stimulus that participants learned to associate with the US.
  - **CS-**: Stimulus never paired with the US.
  - **Generalization**: Stimuli that differ slightly from CS+ (e.g., S4, S5, S6, etc.).
- **CS Indicators**:
  - **CS Indicator 1 (0, 1)**:
    - 1: Trial involves the CS+ stimulus.
    - 0: Trial involves other stimuli.

    Tracks whether the stimulus in a trial is CS+ or not.
  - **CS Indicator 2**: Modified version of CS Indicator 1.
    - For the **generalization phase**: Always 0, even if the trial involves CS+.
    - For the **learning phase**: Same as CS Indicator 1.
- **Perceptual Distance (d-per)**: Difference between participants' perception of the current stimulus and their perception of CS+.

- **Physical Distance (d-phy)**: Difference in size between the current stimulus and CS+.
- **CS Trials (0, 1)**:
  - 0: Not CS+ (used for CS indicators).
  - 1: CS+.

2.1.4 Dataset 1: Simple Conditioning

1.1 Pre-process

This code chunk performs the following tasks:

- Creates a list of data files for the first experiment, excluding the data for participants 15, 17, and 31.
- Loads the data and processes it by:
  - Creating a `stimulus` column and replacing `999` values with `NA`.
  - Creating a `stimulus_phy` column.
  - Removing practice trials and ITI trials.
  - Creating `CStrials` and `CS_phy` columns.
  - Renaming the `Size` column to `Per_size` and selecting the desired columns.
  - Creating a `Phy_size` column.
  - Changing the levels of the `stimulus` and `stimulus_phy` columns.
  - Creating a `trials` column.

The full code can be found in:

- `python/1_data_clean/1_data_clean_1_data.py`
- `python/1_data_clean/1_data_clean.ipynb`

Python Code Snippet

In Python, we use `pandas` dataframes to manipulate the list and process the data:

```
participants = list(range(1, 15)) + list(range(16, 31)) +
    list(range(32, 37)) + list(range(38, 44))
data_s1_list = [f"{root_folder}Data/Experiment_1/{p}/{p}
   _results.txt" for p in participants]
data_s1 = (
pd.concat(
[pd.read_csv(file, sep="\t").assign(participant=
   participant)
for participant, file in enumerate(data_s1_list, start=1)
   ],
ignore_index=True
)
)
```

1.2 Long-Wide Format (for PYMC in Python)

Jags-input-S1-pre: We see that for every participant in `data-S1`, we have 188 trials. We make a pivot table for each variable and save the values. For example:

- `y: US-expect`:
  - The `y` value is an array of shape (`N participant: 40, N trials: 188`) containing values of `US-expect`.
  - `US-expect (y)`: A value between (0, 10), representing how much the participant expected the shock to happen.
- `SPhy (Phy-size)`:
  - The actual size of the stimulus (`Phy-size`).
- `SPer (Per-size)`:
  - How the participant perceives the size of the stimulus (`Per-size`).
- `CSphy (CS-phy)`:
  - The physical size of the conditioned stimulus (`CS+`).

Python Code Snippet for Long-Wide Format

Below is the Python code used to convert the dataset into a long-wide format suitable for Bayesian modeling:

Listing 1: Converting data to long-wide format for Bayesian modeling.

```
variable_list_s1 = {
        'y': 'US_expect',
        'Sphy': 'Phy_size',
        'Sper': 'Per_size',
        'CSphy': 'CS_phy',
        'CSindicator1': 'CStrials',
        'CSindicator2': 'CStrials',
        'shock': 'US'
}
jags_input_s1_pre = {}
# Convert the data from long to wide format.
for key, value in variable_list_s1.items():
df_wide = data_s1.pivot(index='participant',
   columns='trials', values=value).reset_index()
# Reorder the rows by participant.
df_wide = df_wide.set_index('participant').loc[
   natsorted(df_wide['participant'])].reset_index
   ()
df_wide = df_wide.drop(columns=['participant'])
jags_input_s1_pre[key] = df_wide
```

1.3 Compute Distance to CS

This code chunk has the following purposes:

1. **Create a CS Index:** Summing the values of the `CSindicator1` column and calculating the sum of the `CSindicator1` values up to the current column. Extracts CS perception by selecting the `Per_size` column for rows where `stimulus` is "CS+", adding a `trials` column, and converting the data to wide format. The `CS_per_s1` data is reordered by participant and the first column is removed.

Listing 2: Creating CS Index and Summing CSindicator1 Values.

```
for i in range(CS_index.shape[0]):
for j in range(CS_index.shape[1]):
# Calculate the sum of the
   CSindicator1 values up to the
   current column.
CS_index[i, j] = jags_input_s1_pre['
   CSindicator1'].iloc[i, :j+1].sum()
CS_index_df = pd.DataFrame(CS_index,
   index=jags_input_s1_pre['
```

```
          CSindicator1'].index, columns=
          jags_input_s1_pre['CSindicator1'].
          columns)
      jags_input_s1_pre['CS_index'] =
          CS_index_df
```

2. **Extract CS Perception:** Select the `Per_size` column for rows where `stimulus` is "CS+", add a `trials` column, and convert the data to wide format. The `CS_per_s1` data is reordered by participant, and the first column is removed.

Listing 3: Extracting CS Perception and Preparing Wide Format Data.

```
      # Extract CS perception.
      CS_per_s1 = data_s1[data_s1['stimulus
          '] == 'CS+'][['participant', '
          Per_size']].copy()
      # Add trials column as a cumulative
          count based on the participant.
      CS_per_s1['trials'] = CS_per_s1.
          groupby('participant').cumcount()
          + 1
      # Convert data from long to wide
          format.
      CS_per_s1 = CS_per_s1.sort_values(by
          =['participant', 'trials'])
      CS_per_s1_wide = CS_per_s1.pivot(
          index='participant', columns='
          trials', values='Per_size').
          reset_index()
      # Convert all columns to numeric,
          coercing non-numeric values to NaN
          .
      CS_per_s1 = CS_per_s1.apply(pd.
          to_numeric, errors='coerce')
      CS_per_s1_wide.set_index('participant
          ', inplace=True)
```

3. **Compute Moving Average for CS Perception:** Loop through the rows and columns of the `CS_per_s1` data, and calculate the sum of the `CS_per_s1` values up to the current column, excluding NA values.

Listing 4: Computing Moving Average for CS Perception.

```
      # Compute moving average for CS
          perception.
```

```
CS_per_updatemean_s1 = pd.DataFrame(
    index=CS_per_s1_wide.index,
    columns=CS_per_s1_wide.columns)

for i in range(len(CS_per_s1_wide)):
for j in range(1, len(CS_per_s1_wide.
    columns) + 1):
# Calculate the sum of the CS_per_s1
    values up to the current column,
    excluding NA values.
CS_per_updatemean_s1.iloc[i, j - 1] =
     CS_per_s1_wide.iloc[i, :j].dropna
    ().sum() / j
```

4. **Create Empty Matrices for Perceptual and Physical Distance Data:** Initialize matrices to store perceptual and physical distance data for the analysis. The calculations for d-per and d-phy are as follows:

   - ***d*-per: Perceptual Distance:** This represents the difference between the participant's perception of the current stimulus and their perception of the CS+.

     $$d\text{-per}[i,j] = |S\text{-per}[i,j] - \text{CS\_per\_mean}[i,k]|$$

     where:

     $$\text{CS\_per\_mean}[i,k] = \frac{1}{m}\sum_{n=1}^{m} \text{CS\_per}[i,n]$$

     - m: The number of CS+ trials up to trial k. - k: The index of the most recent CS+ trial. - CS_per_mean$[i,k]$: Represents the moving average of CS+ perceptual sizes up to trial k. This moving average is chosen to capture cumulative learning and generalization mechanisms.

   - ***d*-phy: Physical Distance:** This represents the difference in size between the current stimulus and the CS+ size.

     $$d\text{-phy}[i,j] = S\text{-phy}[i,j] - \text{CS-phy}$$

     - CS-phy: A fixed value representing the size of the CS+.

   The Python implementation is as follows:

Listing 5: Calculating d-per and d-phy.

```
# Create empty matrices for perceptual and physical
    distances.
```

```
d_per = np.zeros((40, 188))
d_phy = np.zeros((40, 188))

for i in range(len(d_per)):
for j in range(len(d_per[i])):
# Find the most recent CS+ trial (k) for participant
   i.
k = jags_input_s1_pre['CS_index'].iloc[i, :j+1].max()

# Calculate CS_per_mean up to trial k.
cs_per_mean = CS_per_updatemean_s1.iloc[i, k - 1] if
   k > 0 else 0

# Calculate d-per.
d_per[i, j] = abs(jags_input_s1_pre['Sper'].iloc[i, j
   ] - cs_per_mean)

# Calculate d-phy.
d_phy[i, j] = jags_input_s1_pre['Sphy'].iloc[i, j] -
   jags_input_s1_pre['CSphy'].iloc[i, 0]

# Store d-per and d-phy in the dictionary.
d_list_s1 = {'d_per': d_per, 'd_phy': d_phy}
```

5. **Compute Perceptual and Physical Distances:** Perceptual (d-per) and physical (d-phy) distances are calculated by looping through the rows and columns of the data. These distances are computed as the absolute value of the difference between:

   - d-**per**$[i, j]$: The perceptual size (S-per) and the cumulative mean of CS perception (CS_per_mean$[i, k]$) up to the most recent CS+ trial (k).
   - d-**phy**$[i, j]$: The physical size (S-phy) and the CS+ physical size (stimulus_size[6]).

   The Python implementation is as follows:

Listing 6: Computing Perceptual and Physical Distances.

```
for i in range(1, 41):
for j in range(1, 189):
cs_idx = jags_input_s1_pre['CS_index'].iloc[i - 1, j
   - 1]
s_per = jags_input_s1_pre['Sper'].iloc[i - 1, j - 1]
if pd.isna(s_per):
d_per[i - 1, j - 1] = np.nan
else:
d_per[i - 1, j - 1] = round(abs(s_per -
   CS_per_updatemean_s1.iloc[i - 1, cs_idx - 1]), 2)

s_phy = jags_input_s1_pre['Sphy'].iloc[i - 1, j - 1]
d_phy[i - 1, j - 1] = np.round(abs(s_phy -
   stimulus_size[6]), 2)

d_list_s1['d_per'] = d_per
d_list_s1['d_phy'] = d_phy
```

6. **Merge Distance Data with `data_s1`:** The computed distance data is merged with the original `data_s1` dataset. This is achieved by reshaping the perceptual (*d*-per) and physical (*d*-phy) distances into a long format and aligning them with the original dataset by participant and trials.

Listing 7: Merging Distance Data with Original Dataset.

```
        reshaped = {}
        for key in ['d_per', 'd_phy']:
        ar = d_list_s1[key].flatten()
        reshaped[key] = pd.DataFrame({
                "participant": np.repeat(range(1, d_per.shape
                   [0] + 1), d_per.shape[1]),
                "trials": np.tile(range(1, d_per.shape[1] +
                   1), d_per.shape[0]),
                key: ar,
        })
        # reshaped[key]['participant'] = reshaped[key]['
           participant'].astype(str)

        reshaped[key] = reshaped[key].sort_values(by=["trials
           ", "participant"]).reset_index(drop=True)
        data_s1 = data_s1.merge(reshaped[key], on=['
           participant', 'trials'])
```

1.4 JAGS (PYMC) Input File

Defining the Input Function. The `data_input_s1()` function is defined to create a list of data to be used as input for PYMC. The function takes two arguments:

- `L`: Indicates the type of learning (e.g., continuous or non-continuous learning).
- `indicator`: Specifies the choice of CS indicators for `CSindicator1` or `CSindicator2`.

The function:

1. Creates a list of lists containing the names and values of the elements to be included in the data list.
2. Includes additional elements if `L = 2`.
3. Returns the data list.

Listing 8: Defining the Input Function for PYMC.

```
def data_input_s1(L, indicator=None):
data = {
        'Nparticipants': jags_input_s1_pre['y'].
           shape[0],
        'Ntrials': jags_input_s1_pre['y'].shape
           [1],
        'Nactrials': 14,
        'd_per': [d_list_s1['d_per'][:, 14:188],
           d_list_s1['d_per']][L-1],
        'd_phy': [d_list_s1['d_phy'][:, 14:188],
           d_list_s1['d_phy']][L-1],
        'y': np.array([jags_input_s1_pre['y'].
           iloc[:, 14:188], jags_input_s1_pre['y'
           ]][L-1])
}

if L == 2:
additional_data = {
        'r': np.array(jags_input_s1_pre['shock'])
           ,
        'k': np.array([jags_input_s1_pre['
           CSindicator1'], jags_input_s1_pre['
           CSindicator2']][indicator-1])
}
data.update(additional_data)

return data
```

Creating Input Datasets. Using the `data_input_s1()` function, three datasets are created with different values of `L` and `indicator`:

Listing 9: Creating Input Datasets for PYMC.

```
# Input data without learning trials
```

```
Data1_JAGSinput_G = data_input_s1(1)
# Input data with an assumption of non-continuous
    learning
Data1_JAGSinput_LG = data_input_s1(2, 2)
# Input data with an assumption of continuous
   learning
Data1_JAGSinput_CLG = data_input_s1(2, 1)
```

Saving the Data as Pickle Files. The datasets are saved as pickle files for later use with Python's PYMC sampling. This includes:

- `Data1_JAGSinput_G.pkl`: Input data without learning trials.
- `Data1_JAGSinput_LG.pkl`: Input data with an assumption of non-continuous learning.
- `Data1_JAGSinput_CLG.pkl`: Input data with an assumption of continuous learning.
- `data_s1.pkl`: Processed dataset.

Listing 10: Saving Data to Pickle Files.

```
def save_data_as_pickle(data, filename):
with open(filename, 'wb') as file:
pickle.dump(data, file)

save_data_as_pickle(Data1_JAGSinput_G, f'{
   root_folder}Data/res_py/Data1_JAGSinput_G.pkl')
save_data_as_pickle(Data1_JAGSinput_LG, f'{
   root_folder}Data/res_py/Data1_JAGSinput_LG.pkl'
   )
save_data_as_pickle(Data1_JAGSinput_CLG, f'{
   root_folder}Data/res_py/Data1_JAGSinput_CLG.pkl
   ')
save_data_as_pickle(data_s1, f'{root_folder}Data/
   res_py/Data_s1.pkl')
```

Validation of Python Outputs. To ensure accuracy, the output datasets generated in Python were compared to those generated in R. This comparison was performed using the following Python function:

Listing 11: Comparing Datasets Generated by Python and R.

```
def compare_excel_files(file1, file2, sheet_name=
   None, tolerance=0.001):
```

```
df1 = pd.read_csv(file1)
df2 = pd.read_csv(file2)

if df1.shape != df2.shape:
raise ValueError("The two files do not have the
   same dimensions.")

differences = []

for row in range(df1.shape[0]):
for col in range(df1.shape[1]):
val1 = df1.iat[row, col]
val2 = df2.iat[row, col]

# Check if both values are numeric
if pd.api.types.is_numeric_dtype(type(val1)) and
   pd.api.types.is_numeric_dtype(type(val2)):
if abs(val1 - val2) > tolerance:
differences.append((row + 1, col + 1, val1, val2)
   )  # 1-based index for rows and columns
else:
if val1 != val2:
differences.append((row + 1, col + 1, val1, val2)
   )

if differences:
print("Differences found:")
for diff in differences:
print(f"Row {diff[0] + 1}, Column {diff[1] }:
   File1 = {diff[2] }, File2 = {diff[3] }")
else:
print("No differences found.")
```

Comparison Results. The following datasets were compared:

- `Data1_JAGSinput_G`
- `Data1_JAGSinput_LG`
- `Data1_JAGSinput_CLG`
- `data_s1`

The outputs generated by Python matched exactly with those generated by R.

2.1.5 Dataset 2: Differential Conditioning

2.1 Pre-process. The preprocessing of Dataset 2 involves loading the data, creating new variables, and filtering it for further analysis.

Creating a Data List for Data Loading.

Listing 12: Loading Dataset 2.

```
participants_2 = list(range(41, 81))
data_s2_list = [f"{root_folder}Data/Experiment_2
   /{p}/{p}_results.txt" for p in participants_2]
data_s2 = pd.concat([pd.read_csv(file, sep="\t")
   for file in data_s2_list], keys=range(1, 41),
   names=['participant']).reset_index()
```

Processing the Data.

Listing 13: Data Processing Steps for Dataset 2.

```
conditions = [
((data_s2['C1'] == 1) & (data_s2['group'] == 1))
   | ((data_s2['C10'] == 1) & (data_s2['group'] ==
    2)),
((data_s2['C2'] == 1) & (data_s2['group'] == 1))
   | ((data_s2['C9'] == 1) & (data_s2['group'] ==
   2)),
# Continue for all conditions...
]
choices = ["CS+", "S2", "S3", "S4", "S5", "S6", "
   S7", "S8", "S9", "CS-"]

# Create stimulus column; Change 999 to NA
data_s2['stimulus'] = np.select(conditions,
   choices, default='ITI')
data_s2['Size'] = data_s2['Size'].replace(999, np
   .nan)
data_s2['US_expect'] = data_s2['US_expect'].
   replace(999, 0)

data_s2['stimulus_phy'] = np.select(conditions, [
   "S1", "S2", "S3", "S4", "S5", "S6", "S7", "S8",
    "S9"], default='S10')

# Remove practice trials and ITI trials
```

```
data_s2 = data_s2[(data_s2['block number'] != 2)
   & (data_s2['stimulus'] != 'ITI')]
data_s2['trials'] = data_s2.groupby('participant'
   ).cumcount() + 1
```

—

2.2 Long-wide Format (for PYMC). The data is converted to a long-wide format, suitable for input into Bayesian models.

Listing 14: Converting Dataset 2 to Long-Wide Format.

```
variable_list_s2 = {
        'y': 'US_expect',
        'Sphy': 'Phy_size',
        'Sper': 'Per_size',
        'CSphy_p': 'CS_phy_p',
        'CSphy_m': 'CS_phy_m',
        'CSindicator1_p': 'CSptrials',
        'CSindicator2_p': 'CSptrials',
        'CSindicator1_m': 'CSmtrials',
        'CSindicator2_m': 'CSmtrials',
        'US_p': 'USp',
        'US_m': 'USm'
}

# Convert the data from long to wide format
jags_input_s2_pre = {}
for key, value in variable_list_s2.items():
wide_df = data_s2.pivot(index='participant',
   columns='trials', values=value)
wide_df.sort_index(inplace=True)
jags_input_s2_pre[key] = wide_df
```

—

2.3 Compute Distance to CS. The perceptual and physical distances to the CS are computed.

Listing 15: Computing Perceptual and Physical Distances.

```
# Compute CS index
jags_input_s2_pre['CSp_index'] = np.zeros_like(
   jags_input_s2_pre['CSindicator1_p'])
jags_input_s2_pre['CSm_index'] = np.zeros_like(
   jags_input_s2_pre['CSindicator1_m'])
```

```
for i in range(jags_input_s2_pre['CSindicator1_p'
    ].shape[0]):
for j in range(jags_input_s2_pre['CSindicator1_p'
    ].shape[1]):
jags_input_s2_pre['CSp_index'][i, j] = np.sum(
    jags_input_s2_pre['CSindicator1_p'].iloc[i, :j
    +1])
for i in range(jags_input_s2_pre['CSindicator1_m'
    ].shape[0]):
for j in range(jags_input_s2_pre['CSindicator1_m'
    ].shape[1]):
jags_input_s2_pre['CSm_index'][i, j] = np.sum(
    jags_input_s2_pre['CSindicator1_m'].iloc[i, :j
    +1])
```

—

2.4 PYMC Input File. The data is formatted and saved for use with PYMC.

Listing 16: Preparing PYMC Input Data.

```
def data_input_s2(L, indicator=None):
data = {
        'Nparticipants': jags_input_s2_pre['y'].
            shape[0],
        'Ntrials': jags_input_s2_pre['y'].shape
            [1],
        'Nactrials': 24,
        'd_p_per': d_list_s2[0].iloc[:, 24:180]
            if L == 1 else d_list_s2[0],
        'd_m_per': d_list_s2[1].iloc[:, 24:180]
            if L == 1 else d_list_s2[1],
        'd_p_phy': d_list_s2[2].iloc[:, 24:180]
            if L == 1 else d_list_s2[2],
        'd_m_phy': d_list_s2[3].iloc[:, 24:180]
            if L == 1 else d_list_s2[3],
        'y': jags_input_s2_pre['y'].iloc[:,
            24:180] if L == 1 else
            jags_input_s2_pre['y'],
}

if L == 2:
additional_data = {
        'r_plus': jags_input_s2_pre['US_p'],
        'r_minus': jags_input_s2_pre['US_m'],
        'k_plus': jags_input_s2_pre['
            CSindicator1_p' if indicator == 1 else
            'CSindicator2_p'],
        'k_minus': jags_input_s2_pre['
            CSindicator1_m' if indicator == 1 else
            'CSindicator2_m']
}
data.update(additional_data)

return data

Data2_JAGSinput_G = data_input_s2(1)
Data2_JAGSinput_LG = data_input_s2(2, 2)
Data2_JAGSinput_CLG = data_input_s2(2, 1)

save_data_as_pickle(Data2_JAGSinput_G, f"{
   root_folder}Data/res_py/Data2_JAGSinput_G.pkl")
save_data_as_pickle(Data2_JAGSinput_LG, f"{
   root_folder}Data/res_py/Data2_JAGSinput_LG.pkl"
   )
save_data_as_pickle(Data2_JAGSinput_CLG, f"{
   root_folder}Data/res_py/Data2_JAGSinput_CLG.pkl
   ")
save_data_as_pickle(data_s2, f"{root_folder}Data/
   res_py/Data_s2.pkl")
```

2.1.6 Data Visualization and Plots

We use the Python libraries `matplotlib` and `seaborn` to generate visualizations and save the resulting plots in the folder `Plots/py/1_DataCleanPlots`.

2.1.7 Prepare for Visualization

Before plotting, we clean and prepare the datasets:

Listing 17: Preparing Data for Visualization.

```
        data_s1_dropped = data_s1.dropna(axis='index',
           how='all', subset=['US_expect'])
        data_s2_dropped = data_s2.dropna(axis='index',
           how='all', subset=['US_expect'])
```

```
data_list = [data_s1_dropped, data_s2_dropped]
Nactrials = [Nactrials_1, Nactrials_2]  #
   Assuming these values are defined
plot_list = []
colors = ['red', 'blue']
root_folder = "../../"
plts_folder = root_folder + "Plots/py/1
   _DataCleanPlots/"
```

2.1.8 Group Learning

The plots illustrate:

- **Group Average:** Average US expectancy ratings over acquisition trials for each stimulus (e.g., CS+ and CS-). Error bars represent the standard deviation of ratings.
- **What it Demonstrates:** Captures the general acquisition trend for different stimuli, such as CS+ ratings increasing while CS- ratings remain flat.
- **Individual Participant Lines:** Grey lines represent individual participant ratings, overlaid on group averages.
- **What it Demonstrates:** Highlights inter-individual variability in acquisition learning, showcasing differences between participants.

Listing 18: Generating Learning Plots.

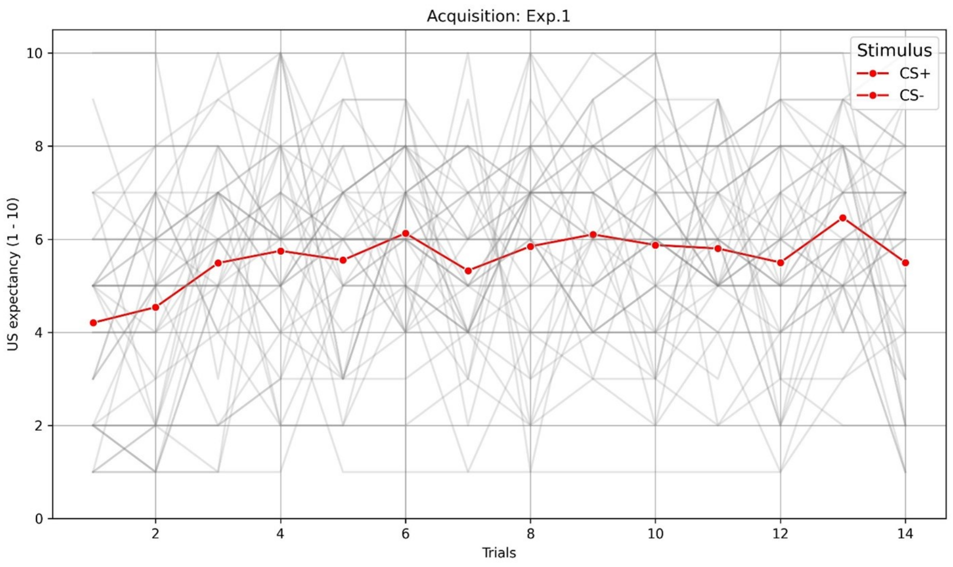
```
for i, data in enumerate(data_list):
# Filter data to include only the first Nactrials
   [i] trials
filtered_data = data[data['trials'] <= Nactrials[
   i]]

# Group by 'trials' and 'stimulus', and calculate
    mean and standard deviation
summarized_data = filtered_data.groupby(['trials'
   , 'stimulus'], observed=False).agg(
mean_ac=('US_expect', 'mean'),
sd_ac=('US_expect', 'std')
).reset_index()

# Merge back with filtered_data to calculate
   individual participant metrics
```

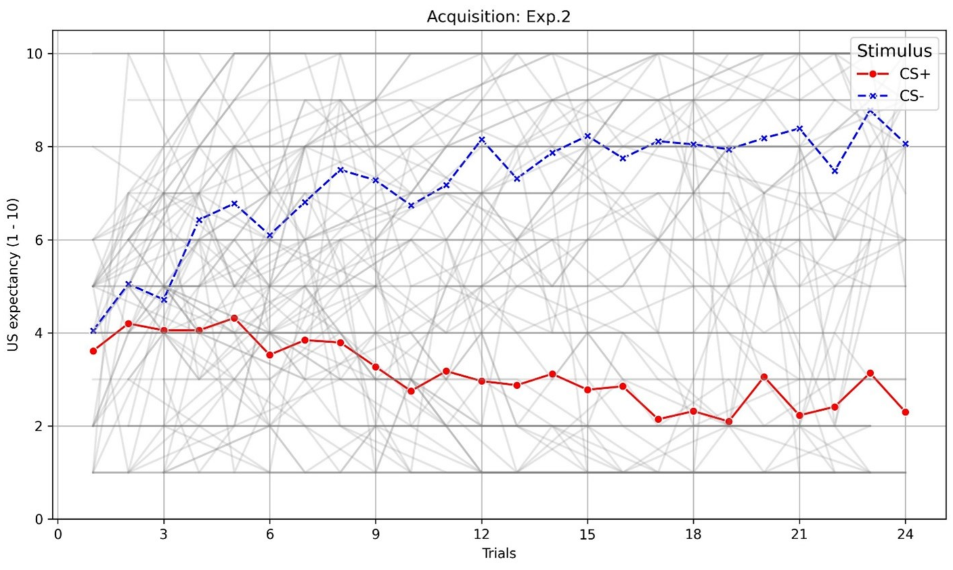
```
summarized_data = pd.merge(filtered_data,
   summarized_data, on=['trials', 'stimulus'])
summarized_data['mean_ac_indi'] = summarized_data
   .groupby(['trials', 'stimulus', 'participant'],
    observed=False)['US_expect'].transform('mean')
summarized_data['sd_ac'] = summarized_data.
   groupby(['trials', 'stimulus', 'participant'],
   observed=False)['US_expect'].transform('std')

# Plotting
plt.figure(figsize=(10, 6))

ax = sns.lineplot(data=summarized_data, x='trials
   ', y='mean_ac', hue='stimulus', style='stimulus
   ', markers=True, err_style="bars", errorbar='sd
   ', palette=colors)

# Adding individual participant lines
for key, grp in summarized_data.groupby(['
   participant', 'stimulus'], observed=False):
ax = grp.plot(ax=ax, x='trials', y='mean_ac_indi'
   , label='_nolegend_', color='grey', alpha=0.2)

# Setting plot title and labels
ax.set_title(f"Acquisition: Exp.{i+1}")
ax.set_xlabel("Trials")
ax.set_ylabel("US expectancy (1 - 10)")

# Setting axis limits and ticks
ax.set_ylim(0, 10.5)
ax.xaxis.set_major_locator(MaxNLocator(integer=
   True))

plt.legend(title='Stimulus', labels=['CS+', 'CS-'
   ], title_fontsize='13', fontsize='11', loc='
   upper right')
plt.grid(True)
plt.tight_layout()

plt.savefig(f"{plts_folder}1_Acquisition_Exp_{i
   +1}.jpg", format='jpg', dpi=300)
# Store the plot in the list
plot_list.append(ax.get_figure())
```

Description of the Visualization:

- **X-Axis:** Number of trials.
- **Y-Axis:** US expectancy ratings on a scale from 1 to 10.
- **Title:** "Acquisition: Exp.n" where n is the experiment number (1 or 2).
- **Legend:** Stimulus types (e.g., CS+ and CS-).
- **Individual Participant Lines:** Shown in grey to highlight variability.

The resulting plots are saved as high-resolution images in the `Plots/py/1_DataCleanPlots/` directory for further analysis and reporting.





Individual Participant Analysis.

Facet grid plots show the mean US expectancy across acquisition trials for individual participants, separated by stimulus.

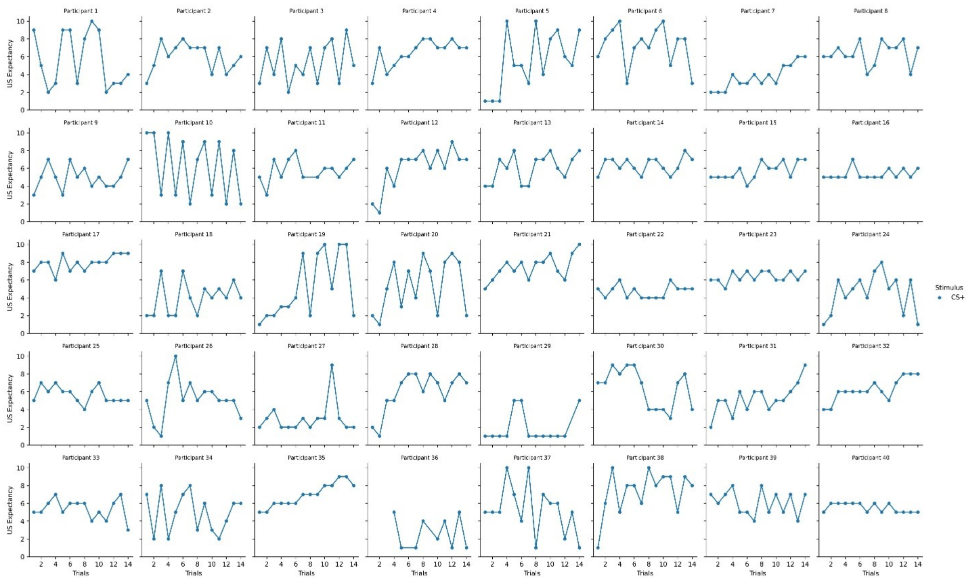
What it Demonstrates:

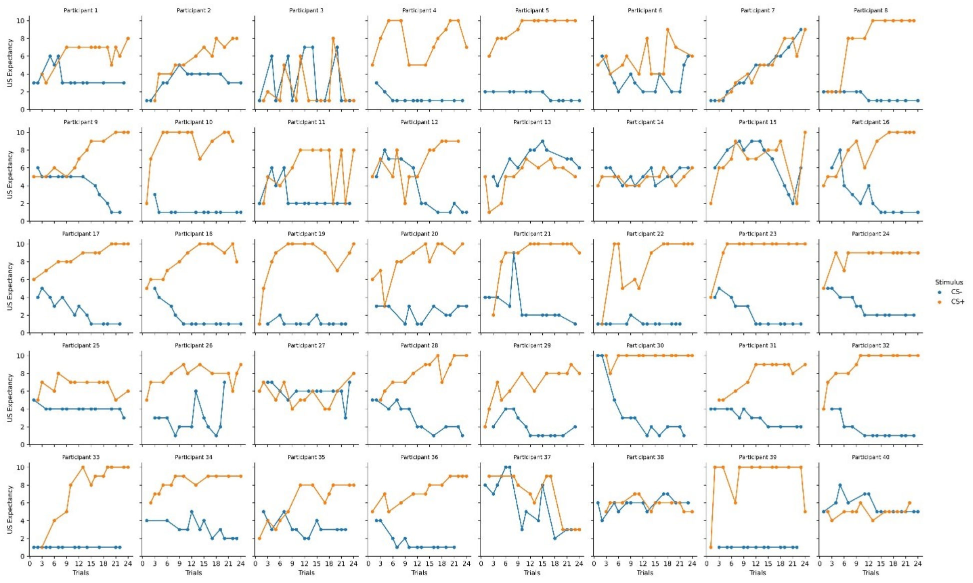
- Provides a detailed view of how each participant learns during acquisition.
- Helps identify outliers or inconsistent patterns in the learning process.

Listing 19: Generating Facet Grid Plots for Individual Participants.

```
for i, data in enumerate(data_list):
# Filter data to include only the first Nactrials[i]
   trials
filtered_data = data[data['trials'] <= Nactrials[i]]

# Calculate mean of US_expect by participant, trial, and
   stimulus
summarized_data = filtered_data.groupby(['participant', '
   trials', 'stimulus'], observed=False).agg(
mean_ac=('US_expect', 'mean')
).reset_index()

# Plotting with Seaborn and Matplotlib
g = sns.FacetGrid(summarized_data, col='participant',
   col_wrap=8, hue='stimulus', height=2.5, aspect=1)
g.map_dataframe(sns.lineplot, x='trials', y='mean_ac')
g.map_dataframe(sns.scatterplot, x='trials', y='mean_ac',
    s=30)
```

```
# Customizing plots
g.set_titles("Participant {col_name}")
g.set_axis_labels("Trials", "US Expectancy")
g.add_legend(title="Stimulus")
for ax in g.axes.flatten():
ax.set_title(ax.get_title(), fontsize='small')
ax.set_ylim(0, 10.5)
ax.xaxis.set_major_locator(plt.MaxNLocator(integer=True))
     # Ensures integer values on x-axis

# Saving each plot in the list
g.savefig(f"{plts_folder}2_Experiment_{i+1}.jpg", format=
   'jpg', dpi=300)

plot_list.append(g)
g.figure.suptitle(f"Experiment {i+1}", y=1.02)
g.figure.tight_layout()
g.figure.show()
```


Participant 1
Participant 2
Participant 3
Participant 4
Participant 5
Participant 6
Participant 7
Participant 8
Participant 9
Participant 10
Participant 11
Participant 12
Participant 13
Participant 14
Participant 15
Participant 16
Participant 17
Participant 18
Participant 19
Participant 20
Participant 21
Participant 22
Participant 23
Participant 24
Participant 25
Participant 26
Participant 27
Participant 28
Participant 29
Participant 30
Participant 31
Participant 32
Participant 33
Participant 34
Participant 35
Participant 36
Participant 37
Participant 38
Participant 39
Participant 40
US Expectancy
Trials
Stimulus
CS+


Participant 1
Participant 2
Participant 3
Participant 4
Participant 5
Participant 6
Participant 7
Participant 8
Participant 9
Participant 10
Participant 11
Participant 12
Participant 13
Participant 14
Participant 15
Participant 16
Participant 17
Participant 18
Participant 19
Participant 20
Participant 21
Participant 22
Participant 23
Participant 24
Participant 25
Participant 26
Participant 27
Participant 28
Participant 29
Participant 30
Participant 31
Participant 32
Participant 33
Participant 34
Participant 35
Participant 36
Participant 37
Participant 38
Participant 39
Participant 40
US Expectancy
Trials
Stimulus
CS-
CS+

2.1.9 Generalization: Group Analysis

The line plot shows US expectancy ratings for generalization stimuli, including individual participant ratings and group averages.

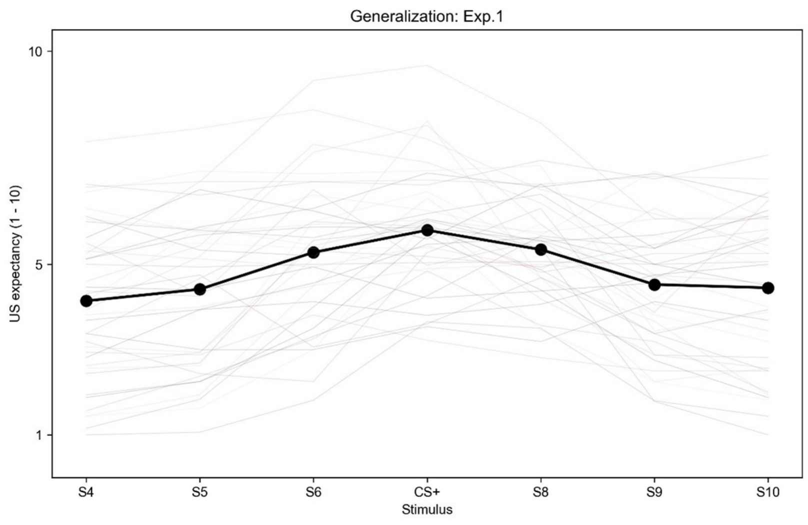
What it Demonstrates:

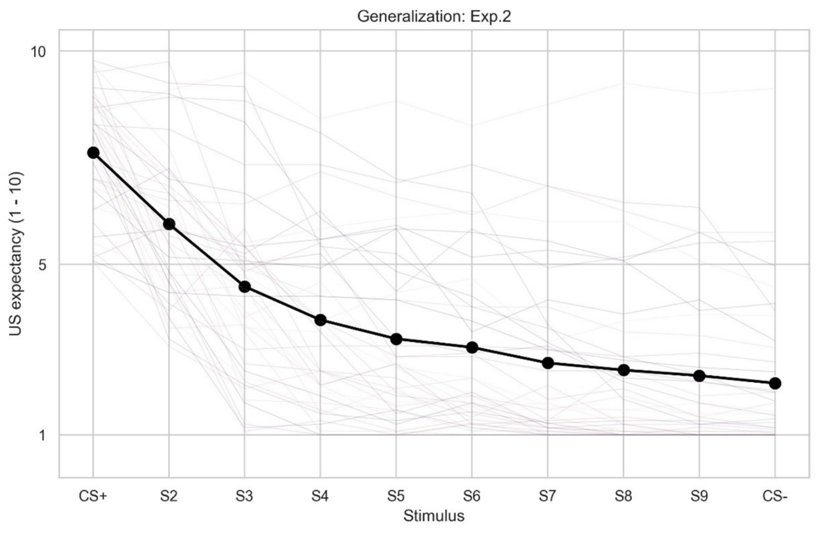
- Captures the generalization gradient, reflecting how responses to non-trained stimuli relate to the trained stimuli (e.g., CS+ generalization to nearby stimuli).

Listing 20: Creating Generalization Plots for Group Analysis.

```
def create_generalization(data, trials_threshold, title,
   stimulus_order, imgTitle):
filtered_data = data[data['trials'] > trials_threshold]
filtered_data['stimulus'] = pd.Categorical(filtered_data[
   'stimulus'], categories=stimulus_order, ordered=True)

# Calculate mean and standard deviation of US_expect by
   stimulus and participant
summarized_data_indi = filtered_data.groupby(['stimulus',
    'participant'], as_index=False, observed=False).agg(
mean_ge_indi=pd.NamedAgg(column='US_expect', aggfunc='
   mean'),
sd_ge_indi=pd.NamedAgg(column='US_expect', aggfunc='std')
)

# Calculate mean and standard deviation of US_expect by
   stimulus
summarized_data = filtered_data.groupby('stimulus',
   as_index=False, observed=False).agg(
mean_ge=pd.NamedAgg(column='US_expect', aggfunc='mean'),
sd_ge=pd.NamedAgg(column='US_expect', aggfunc='std')
)

# Create the plot
plt.figure(figsize=(10, 6))

# Plot individual participant lines
sns.lineplot(data=summarized_data_indi, x='stimulus', y='
   mean_ge_indi', hue='participant',
alpha=0.2, linewidth=0.5, legend=False)

# Plot summary points
```

```
        sns.scatterplot(data=summarized_data, x='stimulus', y='
           mean_ge', s=100, color='black')

        # Plot summary line
        sns.lineplot(data=summarized_data, x='stimulus', y='
           mean_ge', linewidth=2, color='black')

        # Add labels and title
        plt.xlabel("Stimulus")
        plt.ylabel("US expectancy (1 - 10)")
        plt.title(title)

        # Set limits and breaks for the y-axis
        plt.ylim(0, 10.5)
        plt.yticks([1, 5, 10])
        sns.set_theme(style="whitegrid")
        plt.savefig(f"{plts_folder}3_{imgTitle}", format='jpg',
           dpi=300)
        plot_list.append(plt.gcf())  # Store the current figure
           object

        create_generalization(data_list[0], Nactrials[0], "
           Generalization: Exp.1", ['S4','S5','S6', 'CS+','S8','S9
           ','S10'], 'group_generalization_exp1.jpg')
        create_generalization(data_list[1], Nactrials[1], "
           Generalization: Exp.2", ['CS+','S2','S3','S4','S5','S6'
           , 'S7','S8','S9','CS-'], 'group_generalization_exp2.jpg
           ')
```





Individual: Generalization Analysis

Facet grid plots show US expectancy across generalization stimuli for individual participants.

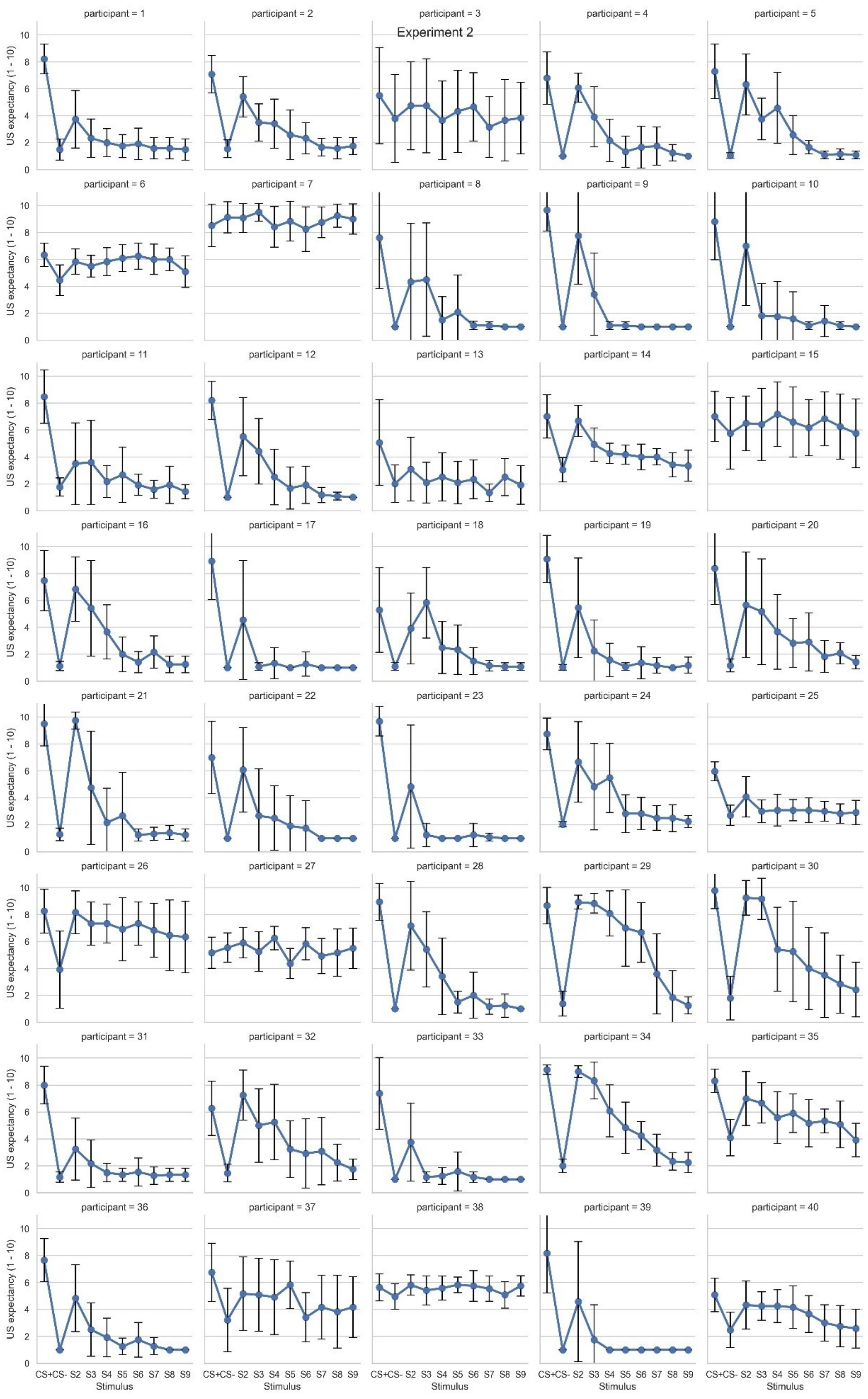
What it Demonstrates:

- Captures participant-specific generalization patterns.
- Highlights how individual differences affect generalization.

Listing 21: Creating Facet Grid Plots for Individual Generalization Analysis.

```
def create_Individual(data, trials_threshold, title,
    imgTitle):
# Filter data to include only trials after the given
    threshold
filtered_data = data[data['trials'] > trials_threshold]
# Calculate mean and standard deviation of US_expect by
    stimulus and participant
summarized_data = filtered_data.groupby(['stimulus', '
    participant'], as_index=False).agg(
mean_ge=pd.NamedAgg(column='US_expect', aggfunc='mean'),
sd_ge=pd.NamedAgg(column='US_expect', aggfunc='std')
)
# Create the plot
g = sns.FacetGrid(summarized_data, col="participant",
    col_wrap=5, sharey=True, sharex=True)
g.map_dataframe(sns.pointplot, x='stimulus', y='mean_ge')
g.map_dataframe(sns.lineplot, x='stimulus', y='mean_ge')
```

```
    g.map(plt.errorbar, 'stimulus', 'mean_ge', 'sd_ge', fmt='
       o', capsize=5, capthick=1, ecolor='black')

    # Add labels and title
    g.set_axis_labels("Stimulus", "US expectancy (1 - 10)")
    g.figure.suptitle(title, fontsize=16)
    g.set(ylim=(0, 11))
    plt.savefig(f"{plts_folder}4_{imgTitle}", format='jpg',
       dpi=300)

    plt.subplots_adjust(top=0.9)
    plot_list.append(plt.gcf())
    plt.show()

    create_Individual(data_list[0], Nactrials_1, "Experiment
       1", "individual_generalization_exp1.jpg")
    create_Individual(data_list[1], Nactrial_2, "Experiment 2
       ", "individual_generalization_exp2.jpg")
```



2.1.10 Perception: Group Analysis

Violin plots with jittered points show the perceived size distribution for each stimulus in the group.

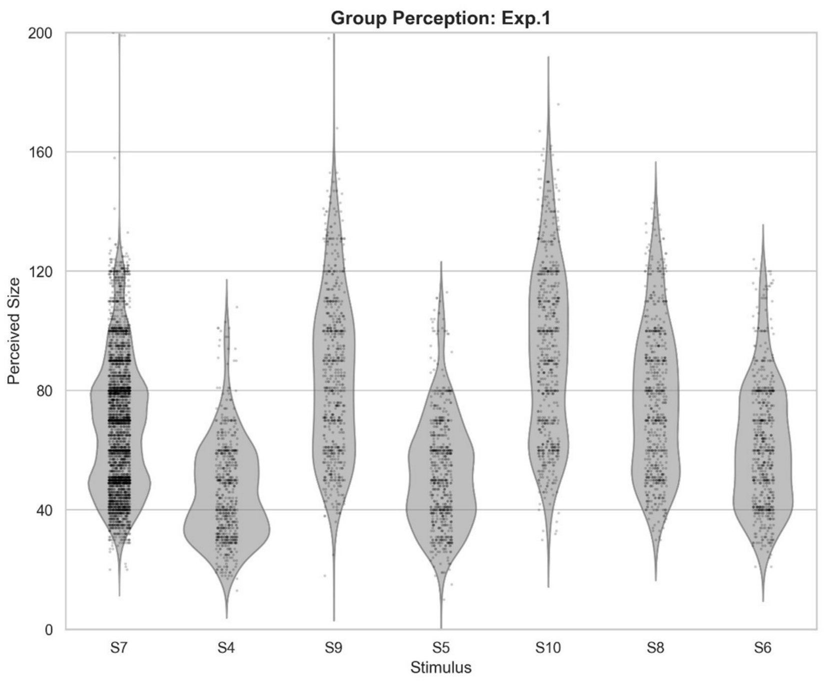
What it Demonstrates:

- Highlights systematic distortions in perceived stimulus size.
- Shows variability in perception among participants.

Listing 22: Creating Group Perception Plots.

```
def create_perception_group_plot(data, title, img_title):
sns.set_theme(style="whitegrid")

# Create the plot
plt.figure(figsize=(10, 8))
ax = sns.violinplot(
x="stimulus_phy",
y="Per_size",
data=data,
inner=None,
color="gray",
alpha=0.5
)

# Add points for individual data
sns.stripplot(
x="stimulus_phy",
y="Per_size",
data=data,
jitter=True,
size=2,
alpha=0.2,
color="black",
dodge=True
)

# Add the title and labels
plt.title(title, fontsize=14, fontweight='bold')
plt.xlabel("Stimulus", fontsize=12)
plt.ylabel("Perceived Size", fontsize=12)

# Customize y-axis
ax.yaxis.set_major_locator(MaxNLocator(integer=True))
plt.ylim(0, 200)
plt.yticks(range(0, 201, 40))

# Save the plot
plt.savefig(f"{plts_folder}6_{img_title}", format='jpg',
   dpi=300)
```

```
        plt.show()

        create_perception_group_plot(data_list[0], "Group
           Perception: Exp.1", "group_perception_exp1.jpg")
        create_perception_group_plot(data_list[1], "Group
           Perception: Exp.2", "group_perception_exp2.jpg")
```



Individual: Perception Analysis

Boxplots display perceived sizes for each stimulus by participant.

What it Demonstrates:

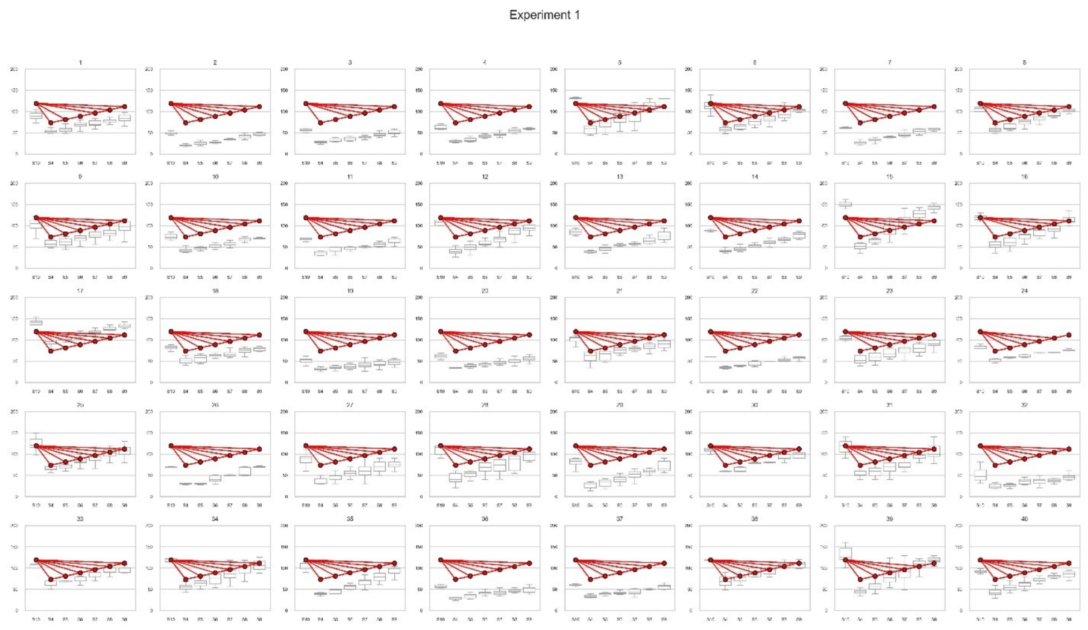
- Examines individual differences in size perception accuracy and consistency.
- Shows alignment (or misalignment) with physical reality.

Listing 23: Generating Individual Perception Analysis Boxplots.

```
        P_in_plot = []
        titles = ["Experiment 1", "Experiment 2"]
        for data in data_list:
        data['stimulus_phy'] = data['stimulus_phy'].astype('
           category')
```

```
P_in_plot = []
idx = 0
for x, data in enumerate(data_list):
fig, axes = plt.subplots(5, 8, figsize=(20, 12))  #
   Adjust size for better spacing
axes = axes.flatten()
idx += 1
for i, participant_id in enumerate(sorted(data['
   participant'].unique())):
ax = axes[i]

# Filter data for each participant
participant_data = data[data['participant'] ==
   participant_id]

# Boxplot
sns.boxplot(
data=participant_data,
x='stimulus_phy',
y='Per_size',
ax=ax,
color="white",
fliersize=0,  # Hide outliers
linewidth=0.8
)

# Add red line for Phy_size
ax.plot(
participant_data['stimulus_phy'].cat.codes,
participant_data['Phy_size'],
color='red',
linewidth=1.2
)

# Add red dots for Phy_size
ax.scatter(
participant_data['stimulus_phy'].cat.codes,
participant_data['Phy_size'],
color='red',
edgecolor='black',
s=30,
zorder=5
```

```
        )

        # Adjust axis labels
        ax.set_title(str(participant_id), fontsize=8)
        ax.set_xlabel("")
        ax.set_ylabel("")
        ax.set_ylim(0, 200)
        ax.set_yticks([0, 50, 100, 150, 200])
        ax.tick_params(axis='both', which='major', labelsize=6)

        # Remove extra axes
        for ax in axes[len(data['participant'].unique()):]:
        ax.remove()

        # Overall title for the experiment
        fig.suptitle(titles[x], fontsize=16)
        plt.tight_layout(rect=[0, 0.03, 1, 0.95])
        plt.savefig(f"{plts_folder}7_group_perception_exp{idx}.
           jpg", format='jpg', dpi=300)
        P_in_plot.append(fig)

        plt.show()
```



2.2 Data Analysis

2.2.1 Goal of the Model

Group participants: into latent categories (*gp groups*) based on their fear generalization behavior:

- **Non-learners:** no learning occurs.
- **Overgeneralizers:** generalize fear broadly without sensitivity to distance metrics.
- **Physical generalizers:** base fear generalization on physical distances.
- **Perceptual generalizers:** base fear generalization on perceptual distances.

Model fear generalization responses using parameters that capture:

- **Learning:** α
- **Generalization:** λ
- **Baseline response:** w_0
- **Response variability:** σ

Provide predictions ($y_{\mathbf{pre}}$) and calculate likelihoods (*loglik*) for new data based on the inferred parameters.

2.2.2 Model Components

Let us remember that:

- $d_{\text{per}}[i, j]$: Perceptual distance between CS+ and the stimulus in trial j for participant i.
- $d_{\text{phy}}[i, j]$: Physical distance between CS+ and the stimulus in trial j for participant i.
- $y[i, j]$ (*US expectation*): Observed response (shock expectation) (0–10).
- $r[i, j]$: (*Shock presence*): (0,1) whether the shock was actually delivered or not.

2.2.3 Likelihood ($y \sim \mathcal{N}$)

In Bayesian modeling, the likelihood represents the probability of observing the data given the model parameters:

$$y[i,j] \sim \mathcal{N}(\theta[i,j], \sigma[\text{zn}[i]]^2)$$

This means that the observed response y, is assumed to follow a normal distribution with:

- **Theta ($\theta[i,j]$):** Predicted response for participant i at trial j calculated below.
- **Sigma ($\sigma[\mathbf{zn}[i]]$):** Noise in the response, dependent on the participant group zn[i]:
  - zn[i] = 1: smaller noise for learners (groups 2–4).
  - zn[i] = 2: larger noise for non-learners (group 1).

2.2.4 Predicted Response ($\theta[i,j]$)

The predicted response $\theta[i,j]$ is modeled using a logistic function:

$$\theta[i,j] = A + \frac{K - A}{1 + \exp\left(-(w_0[i] + w_1[i] \cdot g[i,j])\right)}$$

- **Why logistic transformation?** Because we want to map predictions to a bounded range ($A = 1$, $K = 10$).
- $w_0[i]$: Baseline response for participant i.
- $w_1[i]$: Scaling factor, determining how much the generalization signal $g[i,j]$ influences the predicted response.
- $g[i,j]$: Generalization signal, computed as:

$$g[i,j] = v[i,j] \cdot s[i,j]$$

2.2.5 Hyperparameters

- $v[i,j]$: Learned association strength between the stimulus and the shock.
- $s[i,j]$: Distance-sensitive scaling factor.
- $s[i,j]$: Distance sensitivity.

- $w_0[i]$: Again, it is the baseline response for a participant in the absence of a generalization signal. It is sampled from a prior:

$$w_0[i] \sim \mathcal{N}(w_{0,\mu}, w_{0,\sigma}^2)$$

- $w_1[i]$: Group-dependent. $w_1[i] = 0$ for non-learners. Otherwise, it is sampled from a gamma distribution:

$$w_1[i] \sim \text{Gamma}(w_{1,a}, w_{1,b})$$

How $v[i,j]$ is Calculated

It is updated trial-by-trial using a learning rule.

$$v[i, j+1] = \begin{cases} v[i,j] + \alpha[i] \cdot (r[i,j] - v[i,j]) & \text{if } k[i,j] = 1 \\ v[i,j] & \text{if } k[i,j] = 0 \\ 0 & \text{if } gp[i] = 1 \end{cases}$$

k[i,j] Whether the trial involved CS+

How $s[i,j]$ is Calculated?

$$s[i,j] = \begin{cases} \exp\left(-\lambda[i] \cdot d_{\text{type}}[i,j]\right) & \text{if } v[i,j] > 0 \text{ and } gp[i] > 1 \\ 1 & \text{otherwise} \end{cases}$$

Generalization rate (sampled from prior) indicating how sharply fear decays with distance

$$d_{\text{type}}[i,j] = \begin{cases} d_{\text{per}}[i,j] & \text{for perceptual generalizers } gp[i] = 4 \\ d_{\text{phy}}[i,j] & \text{for physical generalizers } gp[i] = 3 \end{cases}$$

Priors

The model uses the following priors:

- **Learning rate** (α): $\alpha[i] \sim \text{Beta}(\alpha_a, \alpha_b)$
- **Generalization rate** (λ): $\lambda[i] \sim \mathcal{N}(\lambda_\mu, \lambda_\sigma^2)$

The Code

For the code, we are sampling the models in `PYMC` (Python) instead of `JAGS` (R). We are trying to replicate the original model as closely as possible to match the one defined in the file `1v_LG2D.txt`. Below is the Python code used for the translation:

Listing 24: Translated Model Code in PYMC

```
d_list = {
        "e1_clg": load_data_from_pickle("../Data/res_py/
           Data1_JAGSinput_CLG.pkl"),
}

data = d_list['e1_clg']

Nparticipants = data['Nparticipants']
Ntrials = data['Ntrials']
Nactrials = data['Nactrials']

d_per = data['d_per']
d_phy = data['d_phy']

d_per = (d_per - np.mean(d_per)) / np.std(d_per)
d_phy = (d_phy - np.mean(d_phy)) / np.std(d_phy)


y = data['y']
r = data['r']
k = data['k']
A = 1
K = 10

with pm.Model() as model:

pi = pm.Dirichlet("pi", a=np.ones(4) , shape=4)
gp = pm.Categorical("gp", p=pi, shape=Nparticipants)

# Priors
lambda_mu = pm.TruncatedNormal("lambda_mu", mu=0.1, sigma
   =1, lower=0)

lambda_sigma = pm.Uniform("lambda_sigma", lower=1e-9,
   upper=1)
```

```
alpha_raw = pm.Beta("alpha_raw", alpha=1, beta=1)
alpha_mu = pm.Deterministic("alpha_mu", pm.math.clip(
   alpha_raw, 1e-9, 1 - 1e-9))

alpha_kappa = pm.Uniform("alpha_kappa", lower=1, upper
   =10)
alpha_a = alpha_mu * alpha_kappa
alpha_b = (1 - alpha_mu) * alpha_kappa

w0_mu = pm.Normal("w0_mu", mu=0, sigma=10)
w0_sigma = pm.Gamma("w0_sigma", alpha=2, beta=1)

w1_a_raw = pm.Gamma("w1_a_raw", alpha=2, beta=1)
w1_b_raw = pm.Gamma("w1_b_raw", alpha=2, beta=1)
w1_a = pm.Deterministic("w1_a", pm.math.maximum(w1_a_raw,
    1e-9))
w1_b = pm.Deterministic("w1_b", pm.math.maximum(w1_b_raw,
    1e-9))



sigma1 = pm.Uniform("sigma1", lower=1e-9, upper=1.5)  #
   Noise for learners
sigma2 = pm.Uniform("sigma2", lower=1.5, upper=3)    #
   Noise for non-learners
sigma1_broadcasted = at.full((Nparticipants, Ntrials),
   sigma1)
sigma2_broadcasted = at.full((Nparticipants, Ntrials),
   sigma2)




# --- Participant-level Parameters ---
w0 = pm.Normal("w0", mu=w0_mu, sigma=w0_sigma, shape=
   Nparticipants)

w1_1_raw = pm.Gamma("w1_1_raw", alpha=w1_a, beta=w1_b,
   shape=Nparticipants)
w1_1 = pm.Deterministic("w1_1", pm.math.maximum(w1_1_raw,
    1))
w1 = pm.Deterministic("w1", pm.math.switch( at.eq(gp, 1),
    0, w1_1))


alpha_1_raw = pm.Beta("alpha_1_raw", alpha=alpha_a, beta=
   alpha_b, shape=Nparticipants)
alpha_1 = pm.Deterministic("alpha_1", pm.math.clip(
   alpha_1_raw, 1e-9, 1 - 1e-9))  # Apply truncation to
   enforce bounds
alpha = pm.Deterministic("alpha", pm.math.switch(at.eq(gp
   , 1), 0, alpha_1))




lambda_1_raw = pm.Normal("lambda_1_raw", mu=lambda_mu,
   sigma=lambda_sigma, shape=Nparticipants)
lambda_1 = pm.Deterministic("lambda_1", pm.math.maximum(
   lambda_1_raw, 0.0052))

lambda_2_raw = pm.Normal("lambda_2_raw", mu=lambda_mu,
   sigma=lambda_sigma, shape=Nparticipants)
lambda_2 = pm.Deterministic("lambda_2", pm.math.minimum(
   lambda_2_raw, 0.0052))



lambda_ = pm.Deterministic(
"lambda",
at.switch(
at.eq(gp, 1),
0,
at.switch(at.eq(gp,2), lambda_2, lambda_1)
)
)


d_sigma_raw = pm.Gamma("d_sigma_raw", alpha=2, beta=1,
   shape=Nparticipants)
d_sigma = pm.Deterministic("d_sigma", pm.math.maximum(
   d_sigma_raw, 1e-9))


zn = pm.math.switch(at.eq(gp, 1), 2, 1)
zn = at.reshape(zn, (Nparticipants,))
zn_broadcasted = at.broadcast_to(zn[:, None], (
    Nparticipants, Ntrials))



gp_broadcasted = at.broadcast_to((gp > 1)[:, None], d_per
    .shape)
s = pm.Deterministic("s", pm.math.switch(
(gp_broadcasted & (d_per > 0)),
pm.math.exp(-at.broadcast_to(lambda_[:, None], d_per.
    shape) * pm.math.switch(at.eq(at.broadcast_to(gp[:,
    None], d_per.shape), 4), d_per, d_phy)),
1
))


k_t = at.as_tensor_variable(k.T)
r_t = at.as_tensor_variable(r.T)
alpha_sym = at.as_tensor_variable(alpha)
gp_sym = at.as_tensor_variable(gp)
v0 = at.zeros((Nparticipants,))

def v_update(v_prev, r_t, k_t, gp_i,alpha_i):
v_next = at.switch(
at.neq(gp_i, 1),
at.switch( at.eq( k_t, 1), v_prev + alpha_i * (r_t -
    v_prev), v_prev),
0
)
return v_next

v_updates, _ = pytensor.scan(
fn=v_update,
sequences=[r_t, k_t],
outputs_info=v0,
non_sequences=[gp_sym, alpha_sym]
)

v_updates_t = v_updates.T
v = pm.Deterministic("v", v_updates_t)
g = pm.Deterministic("g", v * s)
```

```
w0_broadcasted = at.broadcast_to(w0[:, None], g.shape)
w1_broadcasted = at.broadcast_to(w1[:, None], g.shape)

theta = pm.Deterministic(
"theta",
A + (K - A) / (1 + pm.math.exp(-(w0_broadcasted +
   w1_broadcasted * g)))
)

zn_eq_1 = at.eq(zn_broadcasted, 1)
sigma_broadcasted = at.switch(
at.broadcast_to(zn_eq_1, sigma1_broadcasted.shape),
sigma1_broadcasted,
sigma2_broadcasted
)




# Observed data
y_obs = pm.Normal("y_obs", mu=theta, sigma=
   sigma_broadcasted, observed=y)


intermediate_switch = at.switch(zn_eq_1,
   sigma1_broadcasted, sigma2_broadcasted)


y_pre = pm.Normal("y_pre",mu=theta[:, Nactrials:],sigma=
   sigma_broadcasted[:, Nactrials:],shape=(Nparticipants,
   Ntrials - Nactrials))



loglik = pm.logp(pm.Normal.dist(mu=theta[:, Nactrials:],
   sigma=sigma_broadcasted[:, Nactrials:]), y[:, Nactrials
   :])


print("Shape of theta[:, Nactrials:]:", theta[:,
   Nactrials:].shape)
```

```
        print("Shape of sigma_broadcasted[:, Nactrials:]:",
           sigma_broadcasted[:, Nactrials:].shape)
        print("Shape of y[:, Nactrials:]:", y[:, Nactrials:].
           shape)


        trace = pm.sample(
        draws=25000,
        tune=75000,
        chains=4,
        cores=4,
        target_accept=0.9,
        return_inferencedata=True
        )
        with open('./fitting_res_py/Result_Study1_CLG2D_trace.pkl
           ', 'wb') as f:
        pm.save_trace(trace, f)


        with open('./fitting_res_py/Result_Study1_CLG2D_new.pkl',
            'wb') as f:
        pickle.dump(trace, f)
```

This Python code serves as the implementation of the Bayesian model in PYMC, closely mirroring the JAGS model defined in 1v_LG2D.txt.

Mathematical Representation of the Model

1. Notation

- $N_{\text{participants}}$: Number of participants.
- N_{trials}: Number of trials.
- N_{actrials}: Number of actual trials.
- d_{per}: Perceptual distance (standardized).
- d_{phy}: Physical distance (standardized).
- y: Observed data.
- r, k: Stimulus-response variables.

- A, K: Parameters for response scaling.
- $\lambda, \alpha, w_0, w_1$: Parameters inferred from the model.

2. Priors

Group-Level Parameters

- **Mixture proportions:** $\pi \sim \text{Dirichlet}(1)$
- **Group assignment:** $g_p \sim \text{Categorical}(\pi), \quad p = 1, \ldots, N_{\text{participants}}$

Latent Group-Specific Priors

- $\lambda_\mu \sim \text{TruncatedNormal}(\mu = 0.1, \sigma = 1, \text{lower} = 0)$
- $\lambda_\sigma \sim \text{Uniform}(0, 1)$
- **Beta Distribution for α:**

$$\alpha_\mu \sim \text{Beta}(1, 1) \quad \text{Clipped to } [1 \times 10^{-9}, 1 - 1 \times 10^{-9}]$$

Weights and Noise Terms

- $w_0 \sim \text{Normal}(0, 10), \quad w_1 \sim \text{Gamma}(\alpha = 2, \beta = 1)$
- **Noise levels:**

$$\sigma_1 \sim \text{Uniform}(1 \times 10^{-9}, 1.5), \quad \sigma_2 \sim \text{Uniform}(1.5, 3)$$

3. Participant-Level Parameters

For participant p:

- **Weight priors:**

$$w_0(p) \sim \text{Normal}(\mu = w_0, \sigma = w_0)$$

$$w_1(p) = \begin{cases} 0 & \text{if } g_p = 1 \\ \text{Gamma}(\alpha, \beta) & \text{otherwise} \end{cases}$$

- **Learning rate:**

$$\alpha(p) = \begin{cases} 0 & \text{if } g_p = 1 \\ \text{Beta}(\alpha_a, \alpha_b) & \text{otherwise} \end{cases}$$

- **Scaling coefficients:**

$$\lambda(p) = \begin{cases} 0 & \text{if } g_p = 1 \\ \lambda_1 & \text{if } g_p = 2 \\ \lambda_2 & \text{if } g_p = 3 \text{ or } 4 \end{cases}$$

4. Deterministic Relationships

Stimulus Effect s:

Based on distances and group membership:

$$s(p,t) = \begin{cases} 1 & \text{if } d_{\text{per}} \leq 0 \\ e^{-\lambda(p)\cdot d_{\text{phy}}} & \text{otherwise, for } g_p = 4 \end{cases}$$

Value Update Rule (v):

Recursive update based on responses:

$$v(p,t+1) = \begin{cases} v(p,t) & \text{if } k_t \neq 1 \\ v(p,t) + \alpha(p)\cdot(r_t - v(p,t)) & \text{if } g_p \neq 1 \end{cases}$$

Latent Signal g:

$$g(p,t) = v(p,t)\cdot s(p,t)$$

Response Probability (θ):

$$\theta(p,t) = A + \frac{K - A}{1 + e^{-(w_0(p) + w_1(p)\cdot g(p,t))}}$$

5. Likelihood

Observed responses:

$$y(p,t) \sim \text{Normal}(\mu = \theta(p,t), \sigma = \sigma(p))$$

Predictive distribution:

$$y_{\text{pre}}(p,t) \sim \text{Normal}(\mu = \theta(p,t), \sigma = \sigma(p))$$

2.3 Simulation

The simulation study has three main purposes:

1. To understand how different combinations of parameter values affect response patterns.
2. To evaluate the model's ability to accurately recover the true parameter values.
3. To investigate the impact of incorrect assumptions about latent group allocation on model performance.

The simulation will be divided into two parts. In the first part, we will examine the influence of individual parameters on model performance. In the second part, we will create four groups of simulated participants with different parameter combinations and use Bayesian analysis to estimate the simulated parameter values.

2.3.1 Part-1 Generative Process: Simulate Experimental Data

For a differential learning experiment, there are two CSs for participants to learn. CS+ represents danger, and CS- represents safety. In this simulation, we have 2 CSs and 8 TSs.

- Prepare data:

```
size = np.round(np.arange(50.80, 119.42 +
   7.624, 7.624), 2)
CSp_size = size[4]
Nactrials = 24
Ngetrials = 76
```

```
                Ntrials = Nactrials + Ngetrials

                s_size = np.concatenate((np.repeat(CSp_size,
                   Nactrials), np.random.choice(size,
                   Ngetrials, replace=True)))
                d = np.abs(np.round(s_size - CSp_size, 0))
                stimulus = np.where(d == 0, "CS+", "TS")
                r = np.random.choice([1, 0], size=len(s_size)
                   , p=[0.8, 0.2])
                k = np.where(d == 0, 1, 0)
```

Model: The generative model for simulated data is defined as follows:

```
        def lg_fun(alpha, _lambda, w1=10, w0=-5, sigma=0, trials=
           Ntrials, K=10, A=1, persd=0):
        y = np.zeros(trials)
        theta = np.zeros(trials)
        s_per = np.zeros(trials)
        s_phy = np.zeros(trials)
        perd = np.zeros(trials)
        phyd = d
        v = np.zeros(trials + 1)
        g = np.zeros(trials)
        perd = np.zeros(trials)
        color = ["red", "#56B4E9"]

        for t in range(trials):
        perd[t] = abs(np.random.normal(phyd[t], persd))
        v[t + 1] = v[t] + alpha * (r[t] - v[t]) if k[t] == 1 else
            v[t]
        s_per[t] = np.exp(-_lambda * perd[t])
        s_phy[t] = np.exp(-_lambda * phyd[t])
        g[t] = v[t] * s_per[t]
        theta[t] = A + (K - A) / (1 + np.exp(-(w0 + w1 * g[t])))
        y[t] = np.random.normal(theta[t], sigma)

        data = pd.DataFrame({
                'theta': theta,
                's_per': s_per,
                's_phy': s_phy,
                'v': v[:Ntrials],
                'g': g,
                'stim': stimulus,
                'y': y,
                'trials': np.arange(1, Ntrials + 1),
                'phyd': phyd,
                'perd': perd,
                'r': r,
                's_size': s_size
        })
        data['percept'] = np.where(data['perd'] < 5, '1',
        np.where((data['perd'] > 5) & (data['perd'] < 15), '2', '
           3'))
        return data
```

The Data:

- **Theta:** Predicted response, participant's behavioral response in a model.
- **s_per:** Perceptual generalization signal.
- **s_phy:** Physical generalization signal.
- **v:** Associative strength, representing the learned associative value of a stimulus updated trial-by-trial.
- **g:** Generalized response or signal, calculated as the product of v (associative strength) and s (generalization signal).
- **stim:** The presented stimulus.
- **y:** The observed response.
- **phyd:** Physical distance between the presented stimulus and CS+.
- **perd:** Perceptual distance between the presented stimulus and CS+.
- **r:** Indicates whether reinforcement (reward or punishment) was given in a trial. Typically binary (e.g., 1 = reward, 0 = no reward).
- **s_size:** Stimulus size.

Associative Strength: Think of associative strength as a numerical measure of "how well" an organism has learned that one thing predicts another. For instance:

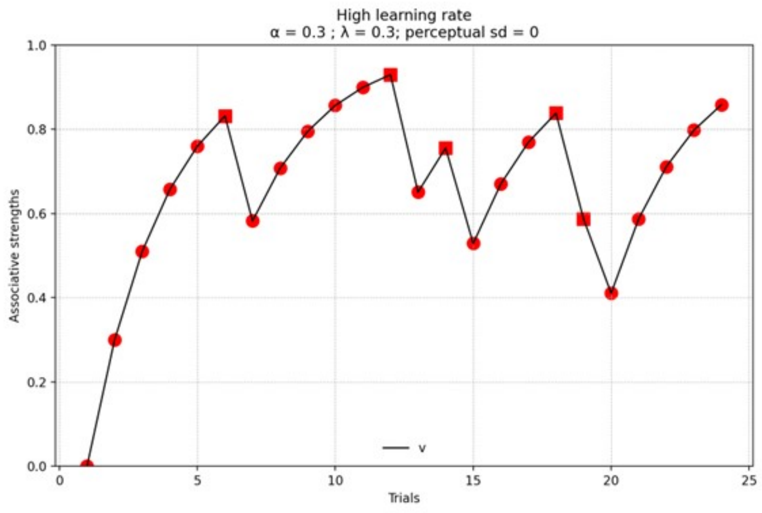
- If a dog learns that a bell ringing (CS) always leads to food (US), the associative strength between the bell and food will be high.
- If the bell stops predicting food, the associative strength will decrease.

Simulation Settings: the code is

```
K = 10
A = 1
w1 = 10
w0 = -5
trials = 100

alpha_high = 0.3
alpha_low = 0.01
lambda_ = 0.3

# Alter learning rate:
L_sim_high = lg_fun(alpha_high, lambda_, w1=10)
L_sim_high_plot = do_plt1(L_sim_high, "High learning rate
   ", alpha_high, lambda_)
L_sim_low = lg_fun(alpha_low, lambda_, w1=10)
L_sim_low_plot = do_plt1(L_sim_low, "Low learning rate",
   alpha_low, lambda_)

L_sim_high_plot.savefig(f"{plts_folder}/
   ChangeLearning_High.png", dpi=300)
L_sim_low_plot.savefig(f"{plts_folder}/ChangeLearning_Low
   .png", dpi=300)
```

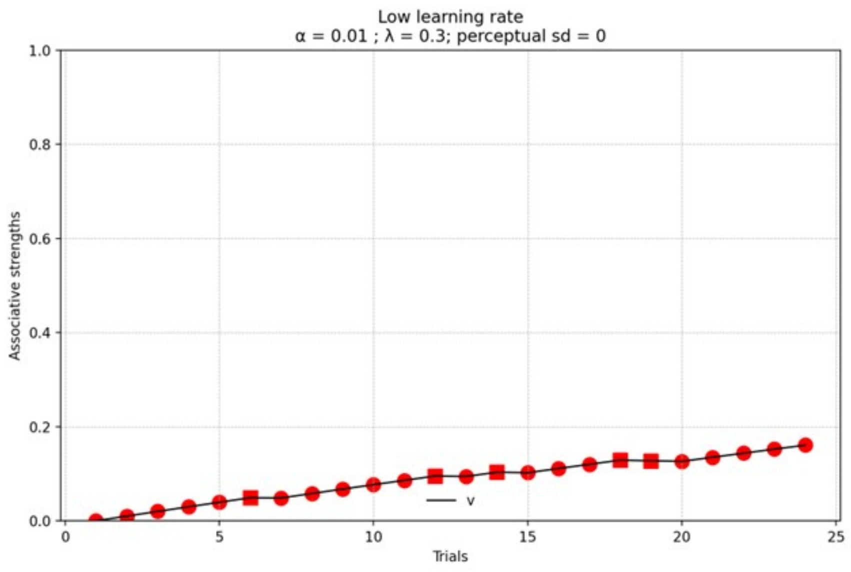
Example Simulation Results:

- **High Learning Rate (α = 0.3):** Higher associative strength, indicating faster learning. Associative strength increases rapidly over trials. The high learning rate



  enables the model to quickly update associations after receiving feedback (presence or absence of reinforcement).

- **Low Learning Rate (α = 0.01):** Slower associative strength updates, indicating slower learning.



Observations:

- Associative strength increases much more gradually over trials.
- The curve appears smoother, with a linear-like increase over time.
- Even after multiple trials, the associative strength does not reach the maximum levels observed in the high learning rate condition.

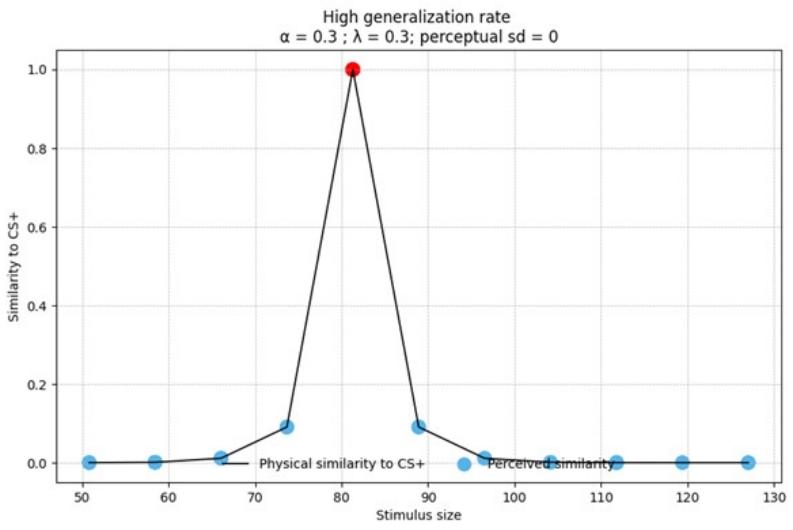
Alter Generalization Rate: the code is

```
alpha = 0.1
lambda_high = 0.3
lambda_low = 0.01

G_sim_high = lg_fun(alpha, lambda_high, w1
    =10)
G_sim_high_plot = do_plt2(G_sim_high, "High
    generalization rate", alpha, lambda_high)
G_sim_low = lg_fun(alpha, lambda_low, w1=10)
G_sim_low_plot = do_plt2(G_sim_low, "Low
    generalization rate", alpha, lambda_low)

G_sim_high_plot.savefig(f"{plts_folder}/
    ChangeGeneralization_High.png", dpi=300)
G_sim_low_plot.savefig(f"{plts_folder}/
    ChangeGeneralization_Low.png", dpi=300)
```

High Generalization Rate ($\lambda = 0.3$):

Observations:

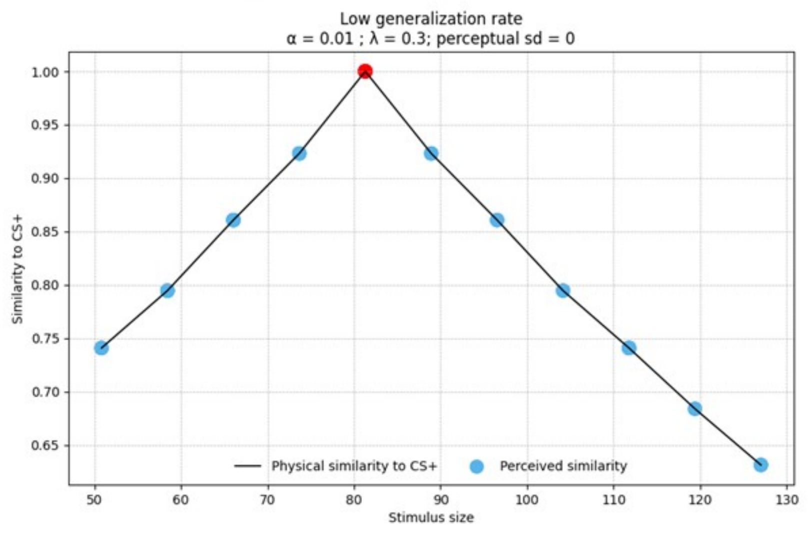
- The similarity to CS+ peaks sharply at the CS+ size (red point).
- Test stimuli (TS, blue points) further from CS+ exhibit almost no similarity.
- The gradient is steep, indicating that the subject discriminates sharply between CS+ and TS.



Low Generalization Rate ($\lambda = 0.01$):

Observations:

- The similarity to CS+ peaks at the CS+ size (red point) but decreases more slowly compared to the high λ condition.
- Test stimuli (TS, blue points) exhibit higher similarity values even at sizes far from CS+.
- The gradient is broader and less steep.



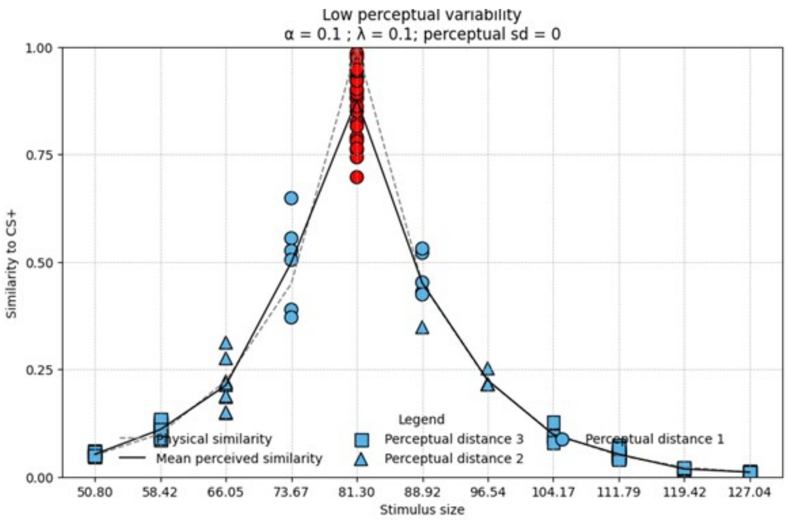
Alter Perceptual Variability (persd)

Listing 25: Python Code to Alter Perceptual Variability

```
        pers_d_low = 2
        pers_d_high = 20

        P_sim_high = lg_fun(0.1, 0.1, w1=10, persd=pers_d_high )
        P_sim_high_plot = do_plt3(P_sim_high, "High perceptual
           variability",0.1,0.1 )
        P_sim_high_plot.savefig(f"{plts_folder}/ChangePersd_High.
           png", dpi=300)

        P_sim_low = lg_fun(0.1,0.1, w1=10, persd=pers_d_low)
        P_sim_low_plot = do_plt3(P_sim_low, "Low perceptual
           variability",0.1,0.1 )
        P_sim_low_plot.savefig(f"{plts_folder}/ChangePersd_Low.
           png", dpi=300)
```

Low Perceptual Variability (perceptual sd=2):

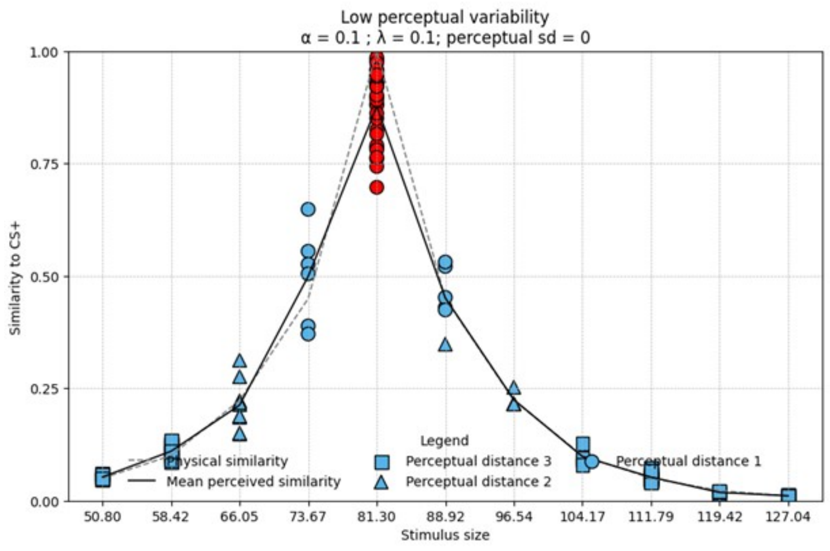
- With low perceptual variability, perceived distances from the CS+ are highly consistent and closely match the actual stimulus size.
- The generalization gradient is smooth and aligns with the expected physical distance to the CS+.
- Similarity to the CS+ decreases symmetrically as the stimulus size deviates from the CS+ size.



- The gradient is narrow and centered around the CS+ size (peak at stimulus size ∼81).
- Perceptual distance categories (small, medium, big) are distinct and show less variability.

High Perceptual Variability (perceptual sd=20):

- With high perceptual variability, perceived distances from the CS+ deviate significantly due to random noise in perception.
- The generalization gradient is irregular, with significant variability in similarity scores across stimuli.
- The similarity to CS+ is less predictable, with a noisier pattern of responses across stimulus sizes.
- The perceived distance categories (small, medium, big) overlap considerably, and responses for some stimuli are inconsistent.
- Peaks and troughs in the gradient reflect noise introduced by high variability.
- High perceptual variability reflects noisy perceptual processing, where perceived stimuli often deviate from physical stimuli.
- This condition models environments or systems with unreliable sensory input or high cognitive uncertainty.



Logit Function

The following Python code implements the logit function to calculate probabilities for various parameter pairs:

```
def logit(w01, w02, w03, w04, w05, w06, w11, w12, w13,
   w14, w15, w16):
# Generate a grid of g values (less dense for clarity)
g = np.linspace(-1, 1, 10)  # Using fewer points for
   better clarity
p_matrices = []

# Calculate probabilities for each pair of w0 and w1
weights = [(w01, w11), (w02, w12), (w03, w13), (w04, w14)
   , (w05, w15), (w06, w16)]
for w0, w1 in weights:
p = 1 + 9 / (1 + np.exp(-(w0 + w1 * g)))  # Compute theta
    values
p_matrices.append(p)

# Plot all probabilities with clearer style
plt.figure(figsize=(8, 6))
colors = ["red", "blue", "pink", "green", "purple", "
   brown"]
labels = [
f"w0 = {w01}, w1 = {w11}", f"w0 = {w02}, w1 = {w12}", f"
   w0 = {w03}, w1 = {w13}",
f"w0 = {w04}, w1 = {w14}", f"w0 = {w05}, w1 = {w15}", f"
   w0 = {w06}, w1 = {w16}"
]
```

```
    # Loop through each set of probabilities, using clean
       lines and small markers
    for p, color, label in zip(p_matrices, colors, labels):
    plt.plot(g, p, color=color, marker='o', markersize=5,
       label=label, linestyle='-', linewidth=1.5)

    plt.xlabel("g")
    plt.ylabel("theta")
    plt.ylim(1, 10)
    plt.legend(title="Parameter pairs", loc='upper left',
       fontsize=9)
    plt.grid(True, which='both', linestyle='--', linewidth
       =0.5)

    plt.savefig(f"{plts_folder}/ChangeW.png", dpi=300)
    plt.show()

    # Test the function
    logit(w01=3, w02=3, w03=0, w04=0, w05=-3, w06=-3, w11=5,
       w12=20, w13=5, w14=20, w15=5, w16=20)
```

Description of the Code

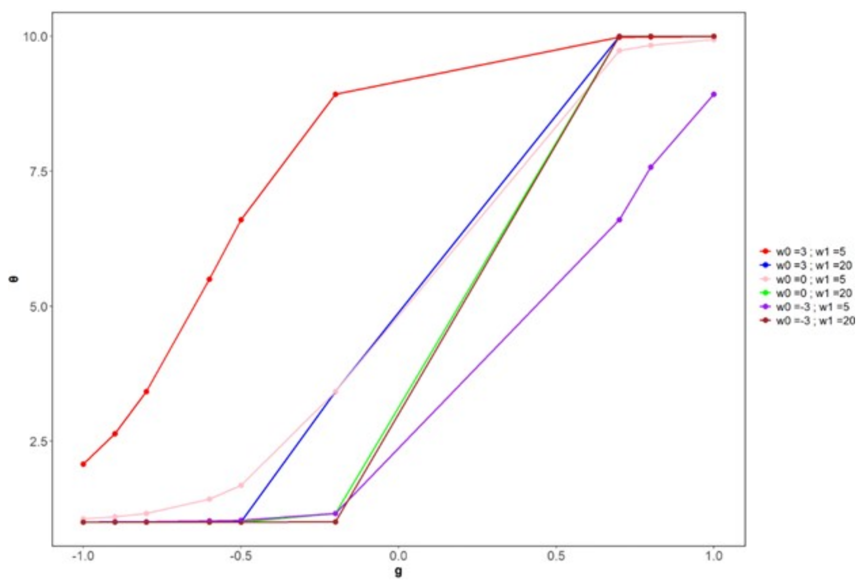
- The function generates a grid of g values ranging from -1 to 1 with 10 points.
- For each pair of w_0 and w_1, the function computes θ values using the formula:

$$\theta = 1 + \frac{9}{1 + \exp -(w_0 + w_1 \cdot g)}$$

- The probabilities for each parameter pair are stored in a list `p_matrices`.
- The function creates a plot for each parameter pair, displaying θ as a function of g with distinct colors and markers.
- The function saves the resulting plot as `ChangeW.png` in the specified directory.

Visualization

The function generates a plot to visualize how changes in w_0 and w_1 affect the logistic function output (θ) for a range of g values. The plot clearly distinguishes different parameter pairs and highlights their effect on the response.



Purpose

- The plot demonstrates how variations in w_0 and w_1 influence the relationship between associative strength (g) and response strength (θ).

Findings

- Increasing w_0 raises the overall response strength, independent of g.
- Increasing w_1 sharpens the transition of θ, making the system more sensitive to small changes in g.
- Combined effects of w_0 and w_1 allow for fine-tuning of the system's behavior, ranging from gradual to steep transitions.

2.3.2 Part-2: 200 Hypothesized Participants

Load Data

The following code snippet is used to load and preprocess the data for a single participant:

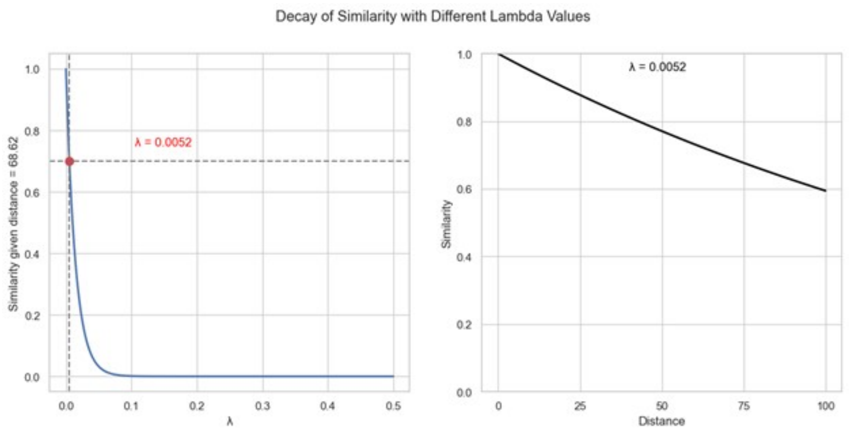
Listing 26: Loading Data from Pickle File

```
def load_data_from_pickle(filename):
with open(filename, 'rb') as file:
data = pickle.load(file)
return data
```

```
        data = load_data_from_pickle("../../Data/res_py/Data_s2.
           pkl")
        data = data[data['participant'] == 1]
        data = data.sort_values(by=['participant', 'trials'])
```

Lambda Value

The code below creates and visualizes two subplots, demonstrating the decay of similarity with varying λ values and distance. We observe from the plots that: These plots



demonstrate how the decay of similarity is influenced by both λ and distance. This relationship is critical in models of generalization, where λ determines how quickly similarity diminishes as stimuli deviate from a reference point.

Listing 27: Decay of Similarity for Different λ Values

```
        max_d = data['d_phy_p'].max()
        min_d = sorted(data['d_phy_p'].unique())[1]

        # Create decay values table for different lambda values
        lambda_values = np.arange(0, 0.501, 0.001)
        sim_decay = pd.DataFrame({
                'lambda': lambda_values,
                'sim_max': np.exp(-lambda_values * max_d),
                'sim_min': np.exp(-lambda_values * min_d)
        })

        # Define similarity thresholds
        min_perc = 0.7
        max_perc = 0.95

        # Calculate the minimum and maximum lambda values
        min_lam = round(-(np.log(min_perc) / max_d), 4)
```

```
max_lam = round(-(np.log(max_perc) / min_d), 4)

# Create decay table for different distances with min and
    max lambda values
d_values = np.arange(0, 101, 1)
sim_decay_2 = pd.DataFrame({
        'd': d_values,
        'sim_max': np.exp(-min_lam * d_values),
        'sim_min': np.exp(-max_lam * d_values),
        'sim_min_og': np.exp(0 * d_values)
})

# Plot decay of similarity for different lambda values
fig, axes = plt.subplots(1, 2, figsize=(14, 6))

# Plot for sim_decay lambda values
sns.lineplot(data=sim_decay, x='lambda', y='sim_max', ax=
   axes[0], linewidth=2)
axes[0].axvline(min_lam, linestyle='--', color='grey')
axes[0].axhline(min_perc, linestyle='--', color='grey')
axes[0].plot(min_lam, min_perc, 'ro', markersize=8)
axes[0].set_xlabel("$\lambda$")
axes[0].set_ylabel(f"Similarity given distance = {max_d}"
   )
axes[0].text(min_lam + 0.1, min_perc + 0.05, f"$\lambda$
   = {min_lam}", color='red')

# Plot for sim_decay_2 values by distance
sns.lineplot(data=sim_decay_2, x='d', y='sim_max', ax=
   axes[1], color='black', linewidth=2)
axes[1].text(40, 0.95, f"$\lambda$ = {min_lam}", color='
   black')
axes[1].set_ylim(0, 1)
axes[1].set_xticks(np.arange(0, 101, 25))
axes[1].set_xlabel("Distance")
axes[1].set_ylabel("Similarity")

# Combine and save plots
plt.suptitle("Decay of Similarity with Different Lambda
   Values")
plt.savefig(f"{plts_folder}/Sim_lambda.pdf", dpi=300,
   format="pdf")
plt.show()
```

Parameter Settings

The following code simulates participants within 4 latent groups:

Listing 28: Parameter Settings for Groups

```
# Define the number of participants in each group
Ngroup1 = 50
Ngroup2 = 50
Ngroup3 = 50
Ngroup4 = 50

# Calculate the total number of participants
Nparticipants = Ngroup1 + Ngroup2 + Ngroup3 + Ngroup4

# Get the number of trials from the data
Ntrials = int(data['trials'].max())

# Set the number of actual trials (excluding warm-up and
   cool-down trials)
Nactrials = 24

# Define the group names and colors for plots
group_nm = ["Non-Learners", "Overgeneralizers", "Physical
    Generalizers", "Perceptual Generalizers", "Unknown"]
color_gp = ["#CC79A7", "#F0E442", "#56B4E9", "#D55E00"]
```

2.3.3 2. Extract Variables and Simulation

Extract Variables

The following code extracts variables from the dataset and organizes them into structured data frames for further analysis:

Listing 29: Extract Variables

```
def extract_fun(var_key, var_val):
_x = {}
for x in range(Ntrials):
col_name = f"{var_key}.{x + 1}"
_col_data = []
for nc in range(Nparticipants):
_col_data.append(data[var_val].iloc[x])
_x[col_name] = _col_data
```

```
return pd.DataFrame(_x)

# Create a dictionary mapping variable names to dataset
   columns
variable_name = {
        "d_phy_p": "d_phy_p",
        "d_phy_m": "d_phy_m",
        "kplus": "CSptrials",
        "kminus": "CSmtrials",
        "rplus": "USp",
        "rminus": "USm"
}

# Apply `extract_fun` to each variable and combine into a
    single DataFrame
variables = pd.concat([extract_fun(var, variable_name[var
   ]) for var in variable_name], axis=1)

# Function to extract column values starting with a
   specific prefix
def getValuesOfCols(col_start):
_cols = [col for col in variables.columns if col.
   startswith(col_start)]
vals = pd.concat([variables[col] for col in _cols],
   ignore_index=True).to_frame(name=col_start)
return vals

# Extract specific variable values
d_phy_p_vals = getValuesOfCols("d_phy_p")
d_phy_m_vals = getValuesOfCols("d_phy_m")
rplus_vals = getValuesOfCols("rplus")

# Initialize variables for simulation
vars = {
        "rplus": np.empty((Nparticipants, Ntrials)),
        "kplus": np.empty((Nparticipants, Ntrials)),
        "kminus": np.empty((Nparticipants, Ntrials)),
        "rminus": np.empty((Nparticipants, Ntrials)),
        "d_phy_p": np.empty((Nparticipants, Ntrials)),
        "d_phy_m": np.empty((Nparticipants, Ntrials)),
}

# Populate variables with extracted values
```

```
for i in range(Nparticipants):
for j in range(Ntrials):
vars["rplus"][i, j] = variables[f'rplus.{j + 1}'][i]
vars["kplus"][i, j] = variables[f'kplus.{j + 1}'][i]
vars["kminus"][i, j] = variables[f'kminus.{j + 1}'][i]
vars["rminus"][i, j] = variables[f'rminus.{j + 1}'][i]
vars["d_phy_p"][i, j] = variables[f'd_phy_p.{j + 1}'][i]
vars["d_phy_m"][i, j] = variables[f'd_phy_m.{j + 1}'][i]
```

Simulation

The simulation function models the learning process for participants, incorporating various parameters such as associative strength, generalization, and response similarity:

Listing 30: Simulation Process

```
v_plus = np.zeros((Nparticipants, Ntrials + 1))
alpha = pd.read_csv('../../vars/randoms/alpha.csv').
   values.tolist()
alpha = [it[0] for it in alpha]
group = np.repeat([1, 2, 3, 4], Nparticipants // 4)
v_minus = np.zeros((Nparticipants, Ntrials + 1))

# Update associative strength (v_plus and v_minus)
for i in range(Nparticipants):
for j in range(Ntrials):
v_plus[i, j + 1] = (
v_plus[i, j] + alpha[i] * (vars['rplus'][i, j] - v_plus[i
   , j])
if group[i] != 1 and vars['kplus'][i, j] == 1
else v_plus[i, j]
if group[i] != 1
else 0
)
v_minus[i, j + 1] = (
v_minus[i, j] + alpha[i] * (vars['rminus'][i, j] -
   v_minus[i, j])
if group[i] != 1 and vars['kminus'][i, j] == 1
else v_minus[i, j]
if group[i] != 1
else 0
)
```

```
        v_plus = pd.DataFrame(v_plus)
        v_plus = pd.melt(v_plus.iloc[:, :180])['value']
```

Learning Function: `lg_fun()`

The `lg_fun()` function simulates a learning process. It accepts three parameters: `persd`, `A`, and `K`. The parameter `persd` represents perceived distance, while `A` and `K` are constants used in response scaling. The function is structured as follows:

1. Initialize matrices and arrays to store intermediate and final results.
2. Update associative strengths (`v_plus`, `v_minus`) based on trial outcomes.
3. Calculate similarity metrics (`s_plus`, `s_minus`).
4. Compute generalization signals (`g`) and predicted responses (`theta`).
5. Generate participant responses (`y`) based on `theta` and noise levels.

Below is the Python code implementation:

Listing 31: Learning Function Implementation

```
        def lg_fun(persd=30, A=1, K=10):
        # Initialize matrices and arrays
        y = np.zeros((Nparticipants, Ntrials))
        theta = np.zeros((Nparticipants, Ntrials))
        s_plus = np.zeros((Nparticipants, Ntrials))
        s_minus = np.zeros((Nparticipants, Ntrials))
        v_plus = np.zeros((Nparticipants, Ntrials + 1))
        v_minus = np.zeros((Nparticipants, Ntrials + 1))
        g = np.zeros((Nparticipants, Ntrials))
        d_per_p = np.zeros((Nparticipants, Ntrials))
        d_per_m = np.zeros((Nparticipants, Ntrials))
        group = np.repeat([1, 2, 3, 4], Nparticipants // 4)

        # Initialize participant-specific parameters
        alpha = np.zeros(Nparticipants)
        lambd = np.zeros(Nparticipants)
        w0 = np.zeros(Nparticipants)
        w1 = np.zeros(Nparticipants)
        sigma = np.zeros(Nparticipants)
        for i in range(Nparticipants):
```

```
alpha[i] = 0 if group[i] == 1 else np.random.beta(1, 1)
if group[i] == 1:
lambd[i] = 0
elif group[i] == 2:
lambd[i] = max(0, min(0.0052, np.random.normal(0.0026,
   0.001)))
else:
lambd[i] = max(0.0052, min(0.3022, np.random.normal
   ((0.0052 + 0.3022) / 2, 0.1)))
w0[i] = np.random.normal(0, 5) if group[i] < 3 else np.
   random.normal(-2, 1)
w1[i] = np.random.gamma(10, 1)
sigma[i] = 2.5 if group[i] == 1 else 0.5

for i in range(Nparticipants):
for j in range(Ntrials):
# Learning
v_plus[i, j + 1] = (
v_plus[i, j] + alpha[i] * (vars['rplus'][i, j] - v_plus[i
   , j])
if group[i] != 1 and vars['kplus'][i, j] == 1
else v_plus[i, j]
)
v_minus[i, j + 1] = (
v_minus[i, j] + alpha[i] * (vars['rminus'][i, j] -
   v_minus[i, j])
if group[i] != 1 and vars['kminus'][i, j] == 1
else v_minus[i, j]
)

# Similarity
d_per_p[i, j] = max(0, np.random.normal(vars['d_phy_p'][i
   , j], persd))
d_per_m[i, j] = max(0, np.random.normal(vars['d_phy_m'][i
   , j], persd))

s_plus[i, j] = (
np.exp(-lambd[i] * d_per_p[i, j])
if v_plus[i, j] > 0 and group[i] > 1 and group[i] == 4
else np.exp(-lambd[i] * vars['d_phy_p'][i, j])
if v_plus[i, j] > 0 and group[i] > 1
else 1
)
```

```
s_minus[i, j] = (
np.exp(-lambd[i] * d_per_m[i, j])
if abs(v_minus[i, j]) > 0 and group[i] > 1 and group[i]
   == 4
else np.exp(-lambd[i] * vars['d_phy_m'][i, j])
if abs(v_minus[i, j]) > 0 and group[i] > 1
else 1
)

# Generalization
g[i, j] = v_plus[i, j] * s_plus[i, j] + v_minus[i, j] *
   s_minus[i, j]
theta[i, j] = A + (K - A) / (1 + np.exp(-(w0[i] + w1[i] *
    g[i, j])))

# Response
y[i, j] = np.random.normal(theta[i, j], sigma[i])

rplus = [it[0] for it in rplus_vals.values]
d_phy_p = [it[0] for it in d_phy_p_vals.values]
d_phy_m = [it[0] for it in d_phy_m_vals.values]

data = pd.DataFrame({
        "participant": np.repeat(np.arange(1,
           Nparticipants + 1), Ntrials),
        "trials": np.tile(np.arange(1, Ntrials + 1),
           Nparticipants),
        "group": np.repeat(group, Ntrials),
        "alpha": np.repeat(alpha, Ntrials),
        "lambda": np.repeat(lambd, Ntrials),
        "w0": np.repeat(w0, Ntrials),
        "sigma": np.repeat(sigma, Ntrials),
        "w1": np.repeat(w1, Ntrials),
        "y": y.flatten(),
        "g": g.flatten(),
        "theta": theta.flatten(),
        "v_plus": v_plus[:, :Ntrials].flatten(),
        "v_minus": v_minus[:, :Ntrials].flatten(),
        "s_plus": s_plus.flatten(),
        "s_minus": s_minus.flatten(),
        "d_per_p": d_per_p.flatten(),
        "d_per_m": d_per_m.flatten(),
        "d_phy_p": d_phy_p,
                "d_phy_m": d_phy_m,
                "rplus": rplus
        })

        # Add stimulus labels
        data["stim"] = np.where(
        data["d_phy_p"] == 0, "CS+",
        np.where(data["d_phy_m"] == 0, "CS-", "TS")
        )

        return data

        result = lg_fun()
        group_labels = ["Non-Learners", "Overgeneralizers", "
           Physical Generalizers", "Perceptual Generalizers"]
        result['group'] = pd.Categorical(result['group'],
           categories=[1, 2, 3, 4], ordered=True).
           rename_categories(group_labels)
        result = result2.merge(data[['trials', 'stimulus']], on='
           trials', how='left')
        result['stimulus'] = pd.Categorical(result['stimulus'],
           categories=["CS+", "S2", "S3", "S4", "S5", "S6", "S7",
           "S8", "S9", "CS-"], ordered=True)
```

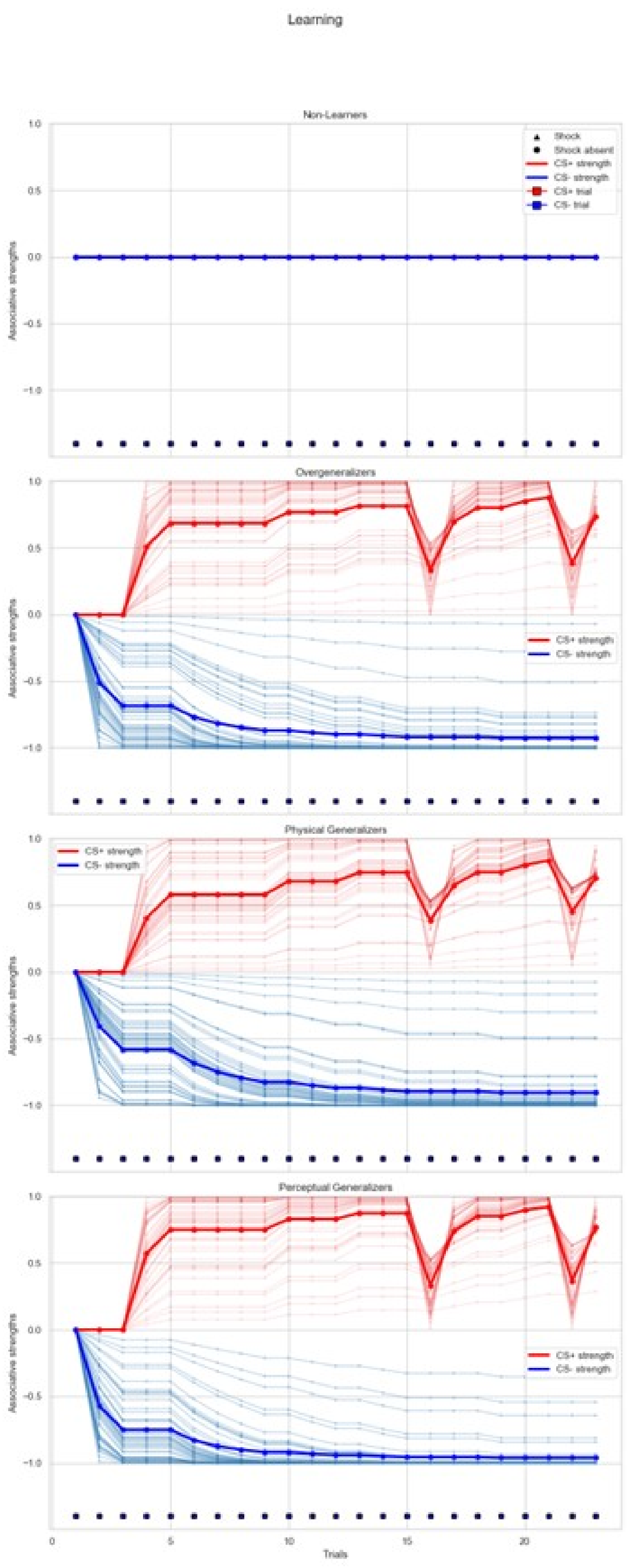
Visualization: 1- Learning

The following Python function, `create_learning_plt()`, generates a visualization of the learning process by plotting associative strengths (`v_plus` and `v_minus`) for each participant and group. It also highlights group-level mean associative strengths and stimulus properties across trials.

Listing 32: Learning Visualization

```
        def create_learning_plt():
        # Filter and group the data
        L = (result[result['trials'] < 24]
        .groupby(['trials', 'group'])
        .agg(mean_vp=('v_plus', 'mean'), mean_vm=('v_minus', '
           mean'))
        .reset_index())

        # Set up the facet grid
        unique_groups = result['group'].unique()
```

```
num_groups = len(unique_groups)
fig, axes = plt.subplots(num_groups, 1, figsize=(10, 6 *
   num_groups), sharex=True, sharey=True)

for i, (group, ax) in enumerate(zip(unique_groups, axes))
   :
# Filter data for the current group
group_data = result[(result['group'] == group) & (result[
   'trials'] < 24)]
group_means = L[L['group'] == group]

# Plot individual lines and points for v_plus
sns.lineplot(data=group_data, x='trials', y='v_plus', hue
   ='participant',
palette=["#e41a1c"], ax=ax, alpha=0.1, legend=None)
sns.scatterplot(data=group_data, x='trials', y='v_plus',
   hue='participant',
palette=["#e41a1c"], ax=ax, alpha=0.1, s=10, legend=None)

# Plot individual lines and points for v_minus
sns.lineplot(data=group_data, x='trials', y='v_minus',
   hue='participant',
palette=["#377eb8"], ax=ax, alpha=0.3, legend=None)
sns.scatterplot(data=group_data, x='trials', y='v_minus',
    hue='participant',
palette=["#377eb8"], ax=ax, alpha=0.3, s=10, legend=None)

# Plot mean lines and points for v_plus (mean_vp) and
   v_minus (mean_vm)
sns.lineplot(data=group_means, x='trials', y='mean_vp',
   color='red', linewidth=3, label='CS+ strength', ax=ax)
sns.scatterplot(data=group_means, x='trials', y='mean_vp'
   , color='red', s=50, legend=None, ax=ax)
sns.lineplot(data=group_means, x='trials', y='mean_vm',
   color='blue', linewidth=3, label='CS- strength', ax=ax)
sns.scatterplot(data=group_means, x='trials', y='mean_vm'
   , color='blue', s=50, legend=None, ax=ax)

# Shape and color for 'rplus' and 'stim'
for _, row in group_data.iterrows():
# Bottom layer for 'stim' - adjusting y position to -1.4
   as in R code
ax.scatter(row['trials'], -1.4, color='red' if row['stim'
   ] == 'CS+' else 'blue',
s=25, marker='s', edgecolor='black')
# Customize labels and title
ax.set_title(group, loc='center')
ax.set_ylabel("Associative strengths")
# Shared x-axis label
axes[-1].set_xlabel("Trials")

# Set y-axis limits and breaks
plt.setp(axes, ylim=(-1.5, 1), yticks=[-1, -0.5, 0, 0.5,
   1]
# Custom legend for shape encoding
legend_elements = [
Line2D([0], [0], marker='^', color='w', label='Shock',
   markerfacecolor='black', markersize=8),
Line2D([0], [0], marker='o', color='w', label='Shock
   absent', markerfacecolor='black', markersize=8),
Line2D([0], [0], color='red', lw=3, label='CS+ strength')
   ,
Line2D([0], [0], color='blue', lw=3, label='CS- strength'
   ),
Line2D([0], [0], marker='s', color='red', label='CS+
   trial', markersize=8, markerfacecolor='red',
   markeredgecolor='black'),
Line2D([0], [0], marker='s', color='blue', label='CS-
   trial', markersize=8, markerfacecolor='blue',
   markeredgecolor='black')
]
axes[0].legend(handles=legend_elements, loc='upper right'
   )
# Apply theme and styling
sns.set(style="whitegrid", rc={"axes.spines.right": False
   , "axes.spines.top": False})
# Overall title
plt.suptitle("Learning", y=1.02, fontsize=16)
plt.tight_layout(rect=[0, 0, 1, 0.98])
plt.savefig("../../Plots/py/2_SimulationPlots/
   Hypothesized_Learning.png", dpi=300, bbox_inches="tight
   ")
plt.show()
return plt
```



The plot is divided into four sections based on participant group: Non-Learners, Overgeneralizers, Physical Generalizers, and Perceptual Generalizers. Each group exhibits distinct learning and generalization behavior:

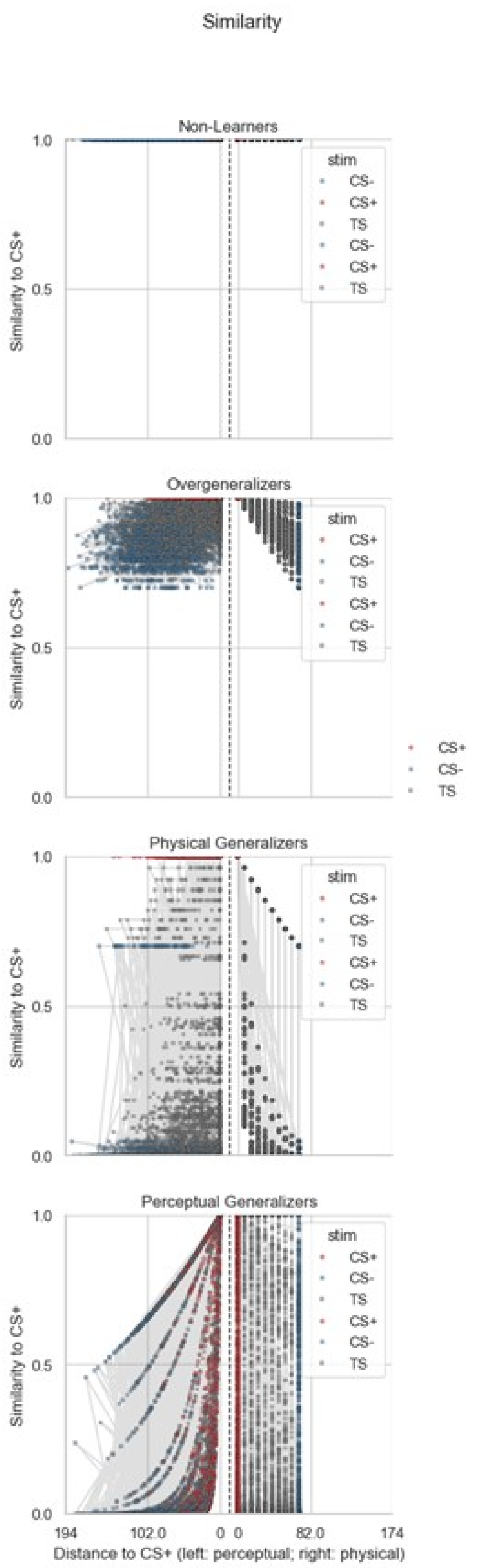
- **Non-Learners:** The associative strengths remain constant (close to zero) across trials, indicating no learning.
- **Overgeneralizers:** High CS+ (red) and CS- (blue) associative strengths, showing that these participants generalize their fear broadly.
- **Physical Generalizers:** A moderate level of generalization, with a clear distinction between CS+ and CS- strengths. Generalization appears limited to stimuli physically similar to CS+.
- **Perceptual Generalizers:** Behavior similar to physical generalizers, but generalization depends more on perceptual similarity (e.g., subjective perception rather than physical resemblance).

2.3.4 Similarity

Listing 33: Python code for similarity analysis

```
def do_sim_similarity_plt():
# Calculate max value for ‘d_per_p‘ and add 10
max_d = result['d_per_p'].max() + 10

# Filter rows where group is "Non-Learners" or (
    v_plus > 0.1 and v_minus < 0.1)
filtered_result = result[(result['group'] == "Non
    -Learners") | ((result['v_plus'] > 0.1) & (
    result['v_minus'] < 0.1))]

# Set up the plot
sns.set(style="whitegrid")
g = sns.FacetGrid(filtered_result, col="group",
    col_wrap=1, height=4, sharey=True)

# Plot each group
for ax, (group, group_data) in zip(g.axes.flat,
    filtered_result.groupby("group")):
# Plot grey lines for s_plus as a function of ‘
    d_phy_p + 10‘ and ‘-d_per_p - 10‘
ax.plot(group_data['d_phy_p'] + 10, group_data['
    s_plus'], color="grey", alpha=0.25)
ax.plot(-group_data['d_per_p'] - 10, group_data['
    s_plus'], color="grey", alpha=0.25)

# Plot scatter points colored and filled by ‘stim
    ‘
sns.scatterplot(data=group_data, x=group_data['
    d_phy_p'] + 10, y="s_plus", hue="stim",
style="stim", palette={"CS+": "#e41a1c", "CS-": "
    #377eb8", "TS": "grey"},
edgecolor="black", s=10, ax=ax, alpha=0.5)
sns.scatterplot(data=group_data, x=-group_data['
    d_per_p'] - 10, y="s_plus", hue="stim",
style="stim", palette={"CS+": "#e41a1c", "CS-": "
    #377eb8", "TS": "grey"},
edgecolor="black", s=10, ax=ax, alpha=0.5)

# Add vertical line at x=0
```

```
ax.axvline(0, color="black", linestyle="--",
   linewidth=1)
ax.set_title(group)
ax.set_ylim(0, 1)
ax.set_yticks([0, 0.5, 1])

# Customize x-axis limits and ticks
g.set(xlim=(-max_d, max_d))
x_ticks = [-max_d, -max_d/2, -10, 10, max_d/2,
   max_d]
x_labels = [round(max_d)+10, round(max_d)/2+10,
   0, 0, round(max_d)/2-10, round(max_d)-10]
for ax in g.axes.flat:
ax.set_xticks(x_ticks)
ax.set_xticklabels(x_labels)

# Customize labels and title
g.set_axis_labels("Distance to CS+ (left:
   perceptual; right: physical)", "Similarity to
   CS+")
plt.subplots_adjust(top=0.9)
g.fig.suptitle("Similarity")

# Customize legend
handles, labels = ax.get_legend_handles_labels()
legend_labels = ["CS+", "CS-", "TS"]
g.add_legend(title="", handles=handles[:3],
   labels=legend_labels, bbox_to_anchor=(1, 0.5),
   loc="center right", frameon=False)

# Save the plot
plt.savefig("../../Plots/py/2_SimulationPlots/
   Sim_Similarity_plus.png", dpi=300, bbox_inches=
   "tight")
plt.show()
```



Observations:

- **Non-Learners:** Consistently high similarity to CS+ across all distances, indicating no meaningful differentiation between stimuli.
- **Overgeneralizers:** High similarity across a broad range of distances, reflecting a lack of specificity and extensive generalization.

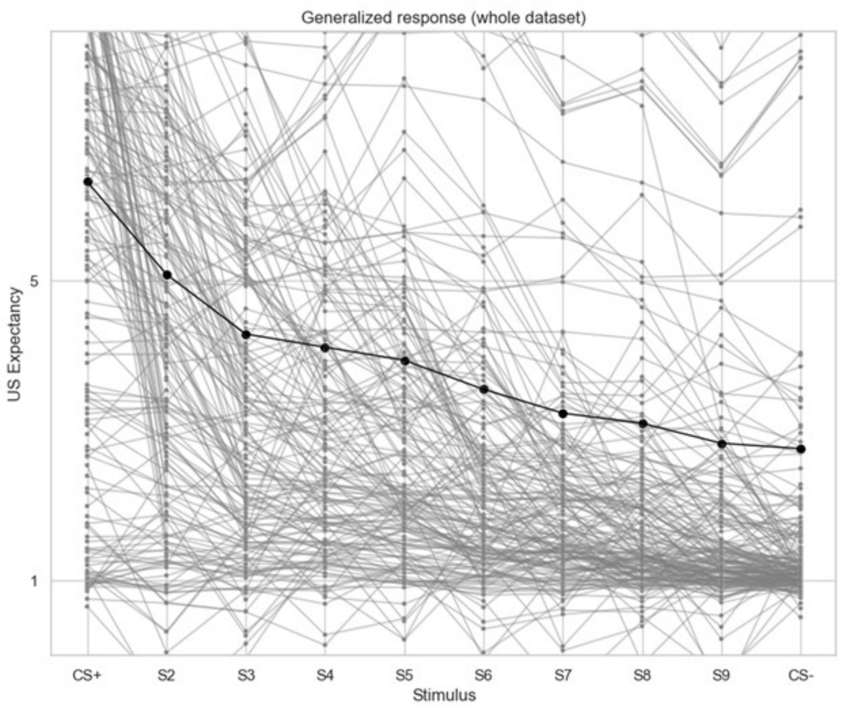
- **Physical Generalizers:** Similarity to CS+ decays primarily as a function of physical distance, showing sensitivity to physical attributes of stimuli.
- **Perceptual Generalizers:** Similarity is modulated more by perceptual distance than physical distance, with a gradual decay reflecting perceptual-based generalization.

2.3.5 Generalized Response (Whole Dataset)

Listing 34: Python Code for Generalized Response (Whole Dataset)

```python
def do_generalization_all_plt():
G_all_data = result[result['trials'] > 24].copy()

# Group by 'stimulus' and calculate mean of 'y'
    for each group
G_all_data['mean_res'] = G_all_data.groupby('
    stimulus')['y'].transform('mean')

# Group by 'participant' and 'stimulus' to
    calculate individual mean 'y' for each group
G_all_data['indi_res'] = G_all_data.groupby(['
    participant', 'stimulus'])['y'].transform('mean
    ')

# Plot
plt.figure(figsize=(10, 8))
sns.lineplot(data=G_all_data, x='stimulus', y='
    indi_res', hue='participant',
palette=["grey"], alpha=0.3, linewidth=1, legend=
    None)
sns.scatterplot(data=G_all_data, x='stimulus', y=
    'indi_res', hue='participant',
palette=["grey"], alpha=0.3, s=10, legend=None)
sns.lineplot(data=G_all_data, x='stimulus', y='
    mean_res', color="black", linewidth=1)
sns.scatterplot(data=G_all_data, x='stimulus', y=
    'mean_res', color="black", s=50)

# Customize plot
plt.xlabel("Stimulus")
plt.ylabel("US Expectancy")
plt.title("Generalized response (whole dataset)")
plt.yticks([1, 5, 10])
```

```
                plt.ylim(0, max(G_all_data['mean_res']) + 2)
                plt.grid(True)
                plt.savefig(f"{plts_folder}Sim_Generalization_all
                    .png", dpi=300, bbox_inches="tight")

                return plt
                plt3 = do_generalization_all_plt()
```



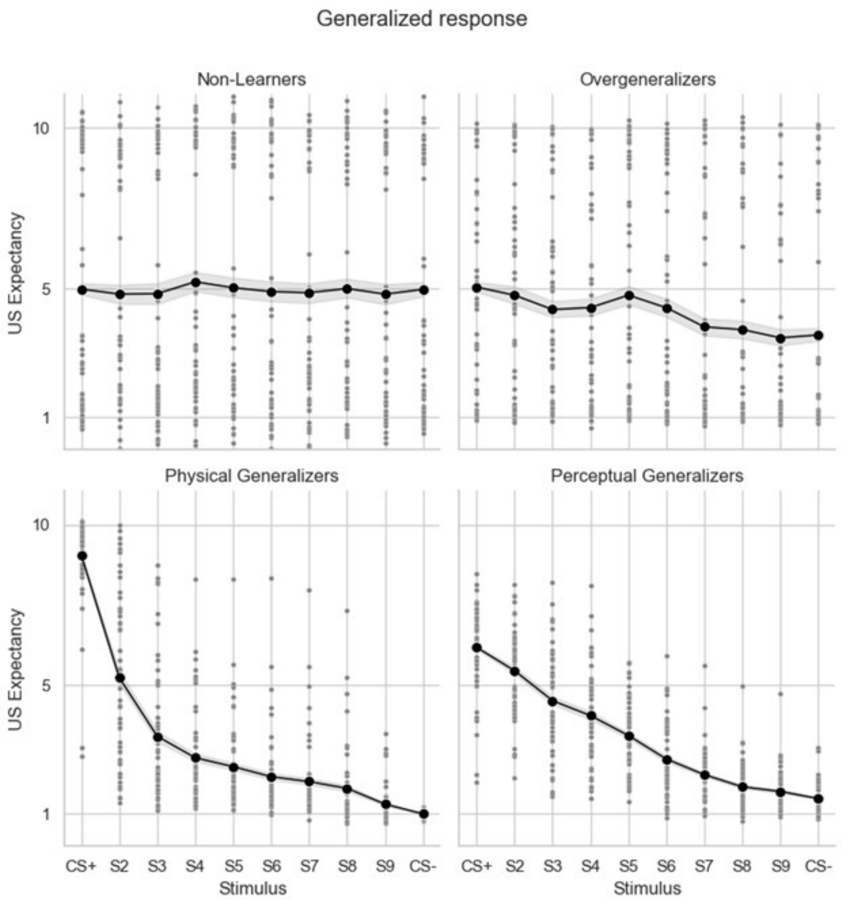
Observations: The black line and dots represent the mean US expectancy across participants for each stimulus.

- A gradual decrease in US expectancy is observed as the stimulus changes from CS+ (conditioned stimulus) to CS- (non-conditioned stimulus).
- This indicates reduced generalization to stimuli less similar to CS+.

2.3.6 Generalized response

Group Observations:

- **Non-Learners:** No change in US expectancy across stimuli, indicating no learning or generalization.



- **Overgeneralizers:** Minimal distinction between stimuli, with a relatively flat curve, suggesting fear or response generalization to all stimuli, regardless of similarity to CS+.
- **Physical Generalizers:** Sharp decline in US expectancy as the stimuli become less similar to CS+, indicating generalization based on physical similarity.
- **Perceptual Generalizers:** Gradual decline in US expectancy, showing generalization based on perceptual distance rather than physical similarity.

Listing 35: Generalized Response by Groups

```
def do_sim_generalization_plt():
# Filter rows where trials > 24
G_data = result[result['trials'] > 24].copy()

# Group by 'group' and 'stimulus' and calculate
    mean of 'y' for each group
G_data['mean_res'] = G_data.groupby(['group', '
    stimulus'])['y'].transform('mean')

# Group by 'participant' and 'stimulus' to
    calculate individual mean 'y' for each group
```

```
G_data['indi_res'] = G_data.groupby(['participant
    ', 'stimulus'])['y'].transform('mean')

# Faceted Plot
g = sns.FacetGrid(G_data, col='group', col_wrap
    =2, height=4, sharey=True)
g.map_dataframe(sns.lineplot, x='stimulus', y='
    indi_res', color='grey', alpha=0.3, linewidth
    =1)
g.map_dataframe(sns.scatterplot, x='stimulus', y=
    'indi_res', color='grey', alpha=0.3, s=10)
g.map_dataframe(sns.lineplot, x='stimulus', y='
    mean_res', color="black", linewidth=1)
g.map_dataframe(sns.scatterplot, x='stimulus', y=
    'mean_res', color="black", s=50)

# Customize plot
g.set_axis_labels("Stimulus", "US Expectancy")
g.set_titles("{col_name}")
g.set(yticks=[1, 5, 10])
g.fig.suptitle("Generalized response", y=1.05)
g.set(ylim=(0, max(G_data['mean_res']) + 2))

# Remove legend and adjust layout
plt.savefig("../../Plots/py/2_SimulationPlots/
    Sim_Generalization.png", dpi=300, bbox_inches="
    tight")

return plt

plt4 = do_sim_generalization_plt()
```

2.3.7 PYMC Implementation

Data Preparation

Listing 36: Prepare Data for PYMC

```
root_folder= "../../"
data_folder= root_folder + "Data/"

def load_data_from_pickle(filename):
with open(filename, 'rb') as file:
```

```
data = pickle.load(file)
return data

data = load_data_from_pickle(f"{data_folder}
   res_py/Data_s2.pkl")
data = data[data['participant'] == 1]
data = data.sort_values(by=['participant', '
   trials'])

# Participant and Trial Configurations
Ngroup1, Ngroup2, Ngroup3, Ngroup4 = 50, 50, 50,
   50
Nparticipants = Ngroup1 + Ngroup2 + Ngroup3 +
   Ngroup4
Ntrials = int(data['trials'].max())
Nactrials = 24

# Save Data
def save_data_as_pickle(data, filename):
with open(filename, 'wb') as file:
pickle.dump(data, file)

result = pd.read_csv(f"{data_folder}res_py/
   result_3_simulation.csv")
q = np.reshape(result['d_per_p'], (Nparticipants,
    -1), order='F')[:,]
q2= np.reshape(result['d_per_p'], (Nparticipants,
    -1), order='F')

# Save to File
save_data_as_pickle(PYMCData(1), f"{data_folder}/
   res_py/Sim_PYMCinput_CLG.pkl")
save_data_as_pickle(PYMCData(0), f"{data_folder}
   res_py/Sim_PYMCinput_G.pkl")
```

Run PYMC

Listing 37: Run PYMC Analysis

```
import pandas as pd
import numpy as np
import pickle
import pymc as pm
import pytensor.tensor as at
import pytensor

def load_data_from_pickle(filename):
with open(filename, 'rb') as file:
data = pickle.load(file)
return data

# Load Data
d_list = {"lg": load_data_from_pickle("../Data/
   res_py/Sim_PYMCinput_CLG.pkl")}
data = d_list['lg']

Nparticipants = data['Nparticipants']
Ntrials = data['Ntrials']
Nactrials = data['Nactrials']

# Data Standardization
d_p_per = (data['d_p_per'] - np.mean(data['
   d_p_per'])) / np.std(data['d_p_per'])
d_p_phy = (data['d_p_phy'] - np.mean(data['
   d_p_phy'])) / np.std(data['d_p_phy'])
d_m_per = (data['d_m_per'] - np.mean(data['
   d_m_per'])) / np.std(data['d_m_per'])
d_m_phy = (data['d_m_phy'] - np.mean(data['
   d_m_phy'])) / np.std(data['d_m_phy'])

y = data['y']
r_plus = data['r_plus']
k_plus = data['k_plus']
r_minus = data['r_minus']
k_minus = data['k_minus']
A, K = 1, 10

# PYMC Model
with pm.Model() as model:
# Define Priors
pi = pm.Dirichlet("pi", a=np.ones(4), shape=4)
gp = pm.Categorical("gp", p=pi, shape=
   Nparticipants)
...
trace = pm.sample(
draws=25000,
tune=75000,
chains=4,
cores=4,
target_accept=0.9,
return_inferencedata=True
)

# Save Results
with open('./fitting_res_py/
   Result_sim_CLG2D_TRACE.pkl', 'wb') as f:
pm.save_trace(trace, f)
with open('./fitting_res_py/Result_sim_CLG2D_new.
   pkl', 'wb') as f:
pickle.dump(trace, f)
```

3 Summary and Conclusions

This document comprehensively addresses the study of fear generalization through theoretical, computational, and empirical lenses. The study aims to replicate existing models of fear generalization and extend their applicability by utilizing modern computational tools such as Python and PyMC.

3.1 Key Accomplishments

1. **Theoretical Foundations:**
   - Provided a detailed overview of fear generalization mechanisms, including associative learning, generalization gradients, extinction learning, and individual differences.
   - Highlighted the clinical relevance of fear generalization for understanding anxiety-related disorders.
2. **Methodological Advancements:**
   - Transitioned from R/JAGS to Python/PyMC for Bayesian modeling, improving accessibility and computational efficiency.
   - Defined a probabilistic graphical model to encapsulate the relationships between latent traits and observed fear responses.
3. **Data Analysis and Visualization:**

    - Processed two datasets (simple and differential conditioning) to extract meaningful insights into fear learning and generalization.
    - Employed advanced visualization techniques to illustrate group and individual differences in fear responses.

4. **Model Implementation:**
    - Successfully implemented a Bayesian model using PyMC to estimate latent variables, group differences, and response variability.
    - Validated Python-generated datasets against their R counterparts, ensuring accuracy and consistency.

5. **Simulation Studies:**
    - Conducted simulation studies to evaluate model performance and explore the effects of individual differences on fear generalization patterns.

3.2 Future Directions

- Explore the model's applicability to other datasets or experimental setups to test its generalizability.
- Investigate potential extensions of the model, such as incorporating additional latent variables or alternative learning paradigms.
- Use the insights from the model to develop targeted therapeutic interventions for anxiety-related disorders.

This work represents a significant step forward in understanding fear generalization, leveraging modern computational methods to enhance the precision and applicability of existing models. The findings provide a robust framework for future research and clinical applications in the field of anxiety disorders.