This MATLAB function, `align\_photom\_foraging`, is designed to align fiber photometry data (likely dopamine-related signals, based on your description) to specific behavioral events (water drinking) in a mouse foraging experiment. The function processes data from multiple trials, aligning the photometry signal to the onset of reward delivery (either left or right reward). Below is a detailed interpretation of the code:

---

### \*\*Function Inputs\*\*

1. \*\*`SessionData`\*\*: A structure containing experimental data, including:

- `TrialSettings`: Contains settings like the sampling rate (`NidaqSamplingRate`).

- `RawData`: Contains trial-specific data, such as:

- `OriginalStateData`: State information for each trial (e.g., state 5 for left reward, state 6 for right reward).

- `OriginalStateTimestamps`: Timestamps corresponding to state changes.

2. \*\*`Photo`\*\*: A cell array where each cell contains the photometry signal (e.g., fluorescence data) for a specific trial.

3. \*\*`Time`\*\*: A cell array where each cell contains the timestamps corresponding to the photometry signal for a specific trial.

---

### \*\*Function Outputs\*\*

- \*\*`all`\*\*: A matrix where each row corresponds to a trial, and each column represents a time point in the aligned photometry signal. The signal is aligned to the onset of reward delivery (left or right). If alignment fails for a trial, the row is filled with `NaN`.

---

### \*\*Code Breakdown\*\*

#### 1. \*\*Initialization\*\*

```matlab

sampleRate = SessionData.TrialSettings(1).GUI.NidaqSamplingRate;

DecimateFactor = 100;

decSR = sampleRate ./ DecimateFactor;

```

- \*\*`sampleRate`\*\*: Retrieves the sampling rate of the photometry data from the session settings.

- \*\*`DecimateFactor`\*\*: A factor (100) used to downsample the data, reducing the effective sampling rate.

- \*\*`decSR`\*\*: The downsampled sampling rate (`sampleRate / DecimateFactor`).

#### 2. \*\*Alignment Window Setup\*\*

```matlab

points = 2 \* decSR;

all = NaN(length(Photo), (2.5 \* points) + 1);

```

- \*\*`points`\*\*: Defines the number of time points corresponding to 2 seconds at the downsampled rate (`2 \* decSR`).

- \*\*`all`\*\*: Initializes the output matrix with `NaN` values:

- Rows: Number of trials (`length(Photo)`).

- Columns: The alignment window spans from `-points` to `+1.5\*points` relative to the reward onset, resulting in `(2.5 \* points) + 1` columns.

- The window covers:

- \*\*Pre-onset\*\*: 2 seconds before reward delivery (`-points`).

- \*\*Post-onset\*\*: 3 seconds after reward delivery (`+1.5\*points`).

---

#### 3. \*\*Main Loop: Aligning Photometry Data to Reward Onset\*\*

```matlab

for i = 1:length(Photo)

try

LeftReward = find(SessionData.RawData.OriginalStateData{1, i} == 5);

RightReward = find(SessionData.RawData.OriginalStateData{1, i} == 6);

```

- The loop iterates over each trial (`i`).

- \*\*`LeftReward`\*\*: Finds indices in the state data where state 5 occurs (left reward delivery).

- \*\*`RightReward`\*\*: Finds indices in the state data where state 6 occurs (right reward delivery).

- The code checks for either left or right reward delivery to align the photometry signal.

---

#### 4. \*\*Processing Left Reward Trials\*\*

```matlab

if ~isempty(LeftReward)

try

OnsetTime(i) = SessionData.RawData.OriginalStateTimestamps{1, i}(LeftReward);

PhotoOnset(i) = find(Time{i} > OnsetTime(i), 1);

all(i, :) = Photo{i}((PhotoOnset(i) - points):(PhotoOnset(i) + 1.5 \* points));

catch

all(i, :) = NaN;

end

```

- If a left reward is detected (`~isempty(LeftReward)`):

- \*\*`OnsetTime(i)`\*\*: Retrieves the timestamp of the left reward delivery.

- \*\*`PhotoOnset(i)`\*\*: Finds the index in the photometry time vector (`Time{i}`) that corresponds to the reward onset (first time point after `OnsetTime(i)`).

- \*\*`all(i, :)`\*\*: Extracts the photometry signal (`Photo{i}`) from `-points` to `+1.5\*points` relative to `PhotoOnset(i)` and stores it in the `all` matrix.

- If an error occurs (e.g., indexing out of bounds), the row is filled with `NaN`.

---

#### 5. \*\*Processing Right Reward Trials\*\*

```matlab

elseif ~isempty(RightReward)

try

OnsetTime(i) = SessionData.RawData.OriginalStateTimestamps{1, i}(RightReward);

PhotoOnset(i) = find(Time{i} > OnsetTime(i), 1);

all(i, :) = Photo{i}((PhotoOnset(i) - points):(PhotoOnset(i) + 1.5 \* points));

catch

all(i, :) = NaN;

end

end

```

- If a right reward is detected (`~isempty(RightReward)`), the process is identical to the left reward case:

- Extracts the reward onset time and aligns the photometry signal.

- Handles errors by filling the row with `NaN`.

---

#### 6. \*\*Error Handling for the Entire Trial\*\*

```matlab

catch

all(i, :) = NaN;

end

```

- If an error occurs at the trial level (e.g., missing state data), the row for that trial is filled with `NaN`.

---

### \*\*Key Observations\*\*

1. \*\*Alignment Window\*\*:

- The photometry signal is aligned to the reward onset, with a window spanning:

- \*\*Pre-onset\*\*: 2 seconds before reward delivery.

- \*\*Post-onset\*\*: 3 seconds after reward delivery.

- This allows analysis of dopamine signals before, during, and after water drinking.

2. \*\*Error Handling\*\*:

- The code uses nested `try-catch` blocks to handle errors gracefully.

- If alignment fails (e.g., due to missing data or indexing issues), the trial is marked as `NaN`.

3. \*\*Downsampling\*\*:

- The `DecimateFactor` (100) suggests that the photometry data is downsampled to reduce computational load.

- The alignment window is defined in terms of the downsampled rate (`decSR`).

4. \*\*Commented Code\*\*:

- There are commented sections (e.g., normalization by baseline mean) that suggest additional processing steps were considered but not implemented:

```matlab

% ./ mean(signal{i}((PhotoOnset(i) - points):(PhotoOnset(i) - points)))

```

- This could be used to normalize the signal (e.g., for ΔF/F calculation), but it is currently disabled.

5. \*\*Assumptions\*\*:

- The code assumes that states 5 and 6 correspond to left and right reward delivery, respectively.

- It assumes that the photometry signal (`Photo`) and timestamps (`Time`) are synchronized and properly formatted.

---

### \*\*Purpose in Fiber Photometry Analysis\*\*

- This function aligns the photometry signal (likely fluorescence from dopamine sensors) to the onset of reward delivery (water drinking).

- The aligned signals can be used to:

- Analyze dopamine dynamics before, during, and after reward consumption.

- Compute ΔF/F (change in fluorescence relative to baseline) for each trial.

- Average signals across trials to study reward-related dopamine activity.

---

### \*\*Potential Improvements\*\*

1. \*\*Normalization\*\*:

- Uncomment and implement the normalization step (e.g., ΔF/F) to account for baseline fluorescence variability.

- Example: Normalize by the mean fluorescence in the pre-onset window.

2. \*\*Multiple Rewards\*\*:

- The current code only handles the first reward event in a trial (e.g., `LeftReward` or `RightReward`). If multiple rewards occur, they are ignored.

- Modify the code to handle multiple reward events per trial if needed.

3. \*\*Validation\*\*:

- Add checks to ensure that `Photo` and `Time` are properly synchronized and have sufficient length for alignment.

- Validate that states 5 and 6 indeed correspond to reward delivery.

4. \*\*Output Metadata\*\*:

- Add outputs for `OnsetTime` and `PhotoOnset` to track alignment details.

- Include trial type (left vs. right reward) for downstream analysis.

5. \*\*Visualization\*\*:

- Add plotting functionality to visualize aligned signals for quality control.

---

### \*\*Summary\*\*

This function aligns fiber photometry signals to reward delivery events (left or right) in a mouse foraging task. It extracts a time window around the reward onset, allowing analysis of dopamine-related fluorescence changes. The code is robust with error handling but could benefit from normalization and handling of multiple reward events. The aligned signals can be used to study dopamine dynamics during reward consumption.

Model 2

The provided MATLAB function `align\_photom\_foraging` aligns fiber photometry data around the time a mouse receives a water reward (left or right) during a foraging task. Here's a breakdown of its purpose and operation:

### \*\*Key Components\*\*

1. \*\*Inputs\*\*:

- `SessionData`: Contains trial-specific information, including reward timestamps.

- `Photo`: Photometry data (likely preprocessed, possibly ΔF/F).

- `Time`: Timestamps for photometry data points.

2. \*\*Output\*\*:

- `all`: A matrix where each row represents a trial, and columns are photometry samples aligned to reward delivery. Trials with alignment errors are filled with `NaN`.

---

### \*\*Workflow\*\*

1. \*\*Setup\*\*:

- \*\*Sampling Rate\*\*: Extracts the original NIDAQ sampling rate from `SessionData`.

- \*\*Decimation\*\*: Reduces data resolution by a factor of 100 (`DecimateFactor=100`), resulting in a decimated sampling rate (`decSR`). This lowers the sampling frequency (e.g., 1000 Hz → 10 Hz) to simplify analysis.

2. \*\*Window Parameters\*\*:

- `points = 2 \* decSR`: Defines a 2-second baseline window (pre-reward).

- Total window: \*\*2 seconds before\*\* to \*\*3 seconds after\*\* reward (5 seconds total), stored in a matrix `all` with dimensions `[trials × 2.5\*points + 1]`.

3. \*\*Trial Processing\*\*:

- For each trial, the code identifies when the mouse drank water (states 5 or 6 for left/right rewards).

- \*\*Reward Timing\*\*:

- Extracts the timestamp (`OnsetTime`) of reward delivery.

- Finds the corresponding photometry sample index (`PhotoOnset`) using `Time`.

- \*\*Data Extraction\*\*:

- Aligns photometry data around `PhotoOnset`, spanning `points` before (2 seconds) and `1.5\*points` after (3 seconds).

- Handles edge cases (e.g., incomplete data) with `try/catch`, filling invalid trials with `NaN`.

---

### \*\*Critical Observations\*\*

- \*\*DFF Normalization\*\*: The code includes a commented-out line for ΔF/F calculation (`Photo{i}./mean(baseline)`), suggesting normalization might occur elsewhere or was omitted.

- \*\*Decimation\*\*: The code assumes `Photo` data is already decimated, as no explicit downsampling is performed here.

- \*\*Error Handling\*\*: Robust `try/catch` blocks ensure trials with missing rewards or alignment issues are excluded gracefully.

---

### \*\*Purpose\*\*

This function \*\*aligns preprocessed photometry traces\*\* (e.g., dopamine signals) to reward events, enabling analysis of neural activity relative to behavior (water drinking). The output (`all`) can be averaged across trials to visualize population-level responses.

This MATLAB function, `AP\_Demodulation`, is designed to demodulate an amplitude-modulated (AM) signal (`rawData`) using a reference signal (`refData`) in the context of fiber photometry analysis. The demodulation is performed in quadrature, which involves generating a 90-degree phase-shifted version of the reference signal and using both signals to extract the original modulated signal. The function also applies a low-pass Butterworth filter to remove high-frequency noise. Below is a detailed interpretation of the code:

---

### \*\*Function Inputs\*\*

1. \*\*`rawData`\*\*: The amplitude-modulated input signal (e.g., fluorescence data from a photometry experiment).

2. \*\*`refData`\*\*: The reference signal used for demodulation (e.g., the modulation carrier signal).

3. \*\*`sampleRate`\*\*: The sampling rate of the data (in Hz).

4. \*\*`modAmp`\*\*: The amplitude of the modulation signal (used for amplitude correction).

5. \*\*`modFreq`\*\*: The frequency of the modulation signal (in Hz).

6. \*\*`lowCutoff`\*\*: The corner frequency (in Hz) for a 5-pole Butterworth low-pass filter. If not provided, it defaults to an empty value (`[]`).

---

### \*\*Function Outputs\*\*

- \*\*`demodData`\*\*: The demodulated signal, corrected for the modulation amplitude and filtered to remove high-frequency noise.

---

### \*\*Code Breakdown\*\*

#### 1. \*\*Default Input Handling\*\*

```matlab

if nargin < 5

lowCutoff = [];

end

```

- If fewer than 5 input arguments are provided, `lowCutoff` is set to an empty array (`[]`).

- This allows the function to proceed without a low-pass filter cutoff if not specified.

---

#### 2. \*\*Prepare Reference Data\*\*

```matlab

refData = refData(1:length(rawData), 1); % adjust length of refData to rawData

refData = refData - mean(refData); % suppress DC offset

```

- \*\*`refData` Adjustment\*\*:

- The reference signal (`refData`) is truncated or padded to match the length of `rawData`.

- Only the first column of `refData` is used (assuming it is a matrix).

- \*\*DC Offset Removal\*\*:

- The mean of `refData` is subtracted to remove any DC (direct current) offset, ensuring the reference signal is centered around zero.

- This step is critical for accurate demodulation, as DC offsets can introduce artifacts.

---

#### 3. \*\*Generate 90-Degree Phase-Shifted Reference Signal\*\*

```matlab

samplesPerPeriod = (1/modFreq) / (1/sampleRate);

quarterPeriod = round(samplesPerPeriod/4);

refData90 = circshift(refData, [-quarterPeriod]); % update Michi (7/10/19)

```

- \*\*`samplesPerPeriod`\*\*:

- Calculates the number of samples per modulation period: `1/modFreq` gives the period in seconds, and dividing by `1/sampleRate` (the sampling interval) converts it to samples.

- \*\*`quarterPeriod`\*\*:

- Calculates the number of samples corresponding to a 90-degree phase shift (one-quarter of the modulation period).

- Rounded to the nearest integer using `round`.

- \*\*`refData90`\*\*:

- Creates a 90-degree phase-shifted version of `refData` by circularly shifting it by `-quarterPeriod` samples (i.e., shifting backward in time).

- The comment "update Michi (7/10/19)" suggests this is a corrected version of an earlier implementation (previously `circshift(refData, [1 quarterPeriod])`).

- The negative shift (`-quarterPeriod`) ensures the correct phase shift direction for quadrature demodulation.

---

#### 4. \*\*Quadrature Demodulation\*\*

```matlab

processedData\_0 = rawData .\* refData;

processedData\_90 = rawData .\* refData90;

```

- \*\*In-Phase Component (`processedData\_0`)\*\*:

- Multiplies the raw signal (`rawData`) by the original reference signal (`refData`).

- This extracts the in-phase (0-degree) component of the modulated signal.

- \*\*Quadrature Component (`processedData\_90`)\*\*:

- Multiplies the raw signal (`rawData`) by the 90-degree phase-shifted reference signal (`refData90`).

- This extracts the quadrature (90-degree) component of the modulated signal.

- \*\*Purpose\*\*:

- Quadrature demodulation uses both in-phase and quadrature components to fully reconstruct the amplitude of the modulated signal, regardless of its phase relative to the reference.

---

#### 5. \*\*Low-Pass Filtering\*\*

```matlab

lowCutoff = lowCutoff / (sampleRate/2); % normalized CutOff by half SampRate (see doc)

[b, a] = butter(5, lowCutoff, 'low');

```

- \*\*Normalization\*\*:

- The cutoff frequency (`lowCutoff`) is normalized by the Nyquist frequency (`sampleRate/2`) to meet the requirements of MATLAB's filter design functions.

- \*\*Butterworth Filter Design\*\*:

- Designs a 5th-order Butterworth low-pass filter using `butter`.

- The filter coefficients (`b` and `a`) are computed for the normalized cutoff frequency.

---

#### 6. \*\*Padding and Filtering\*\*

```matlab

pad = 1;

if pad

paddedData\_0 = processedData\_0(1:sampleRate, 1);

paddedData\_90 = processedData\_0(1:sampleRate, 1);

demodDataFilt\_0 = filtfilt(b, a, [paddedData\_0; processedData\_0]);

demodDataFilt\_90 = filtfilt(b, a, [paddedData\_90; processedData\_90]);

processedData\_0 = demodDataFilt\_0(sampleRate + 1: end, 1);

processedData\_90 = demodDataFilt\_90(sampleRate + 1: end, 1);

else

processedData\_0 = filtfilt(b, a, processedData\_0);

processedData\_90 = filtfilt(b, a, processedData\_90);

end

```

- \*\*Padding Option (`pad = 1`)\*\*:

- If padding is enabled (`pad = 1`), the data is padded to reduce edge effects during filtering:

- `paddedData\_0` and `paddedData\_90` are created by duplicating the first `sampleRate` samples of `processedData\_0`.

- The padded data is concatenated with the original data (`[paddedData\_0; processedData\_0]`).

- The padded signals are filtered using `filtfilt` (zero-phase filtering) with the Butterworth filter coefficients.

- After filtering, the padding is removed by selecting only the portion corresponding to the original data (`sampleRate + 1: end`).

- \*\*No Padding (`pad = 0`)\*\*:

- If padding is disabled, the signals are filtered directly using `filtfilt` without padding.

- \*\*Purpose\*\*:

- The low-pass filter removes high-frequency noise introduced during demodulation (e.g., harmonics from the multiplication process).

- Padding helps mitigate edge effects (transients) at the start and end of the filtered signal.

---

#### 7. \*\*Reconstruct Demodulated Signal\*\*

```matlab

demodData = (processedData\_0 .^2 + processedData\_90 .^2) .^(1/2);

```

- \*\*Quadrature Reconstruction\*\*:

- Combines the filtered in-phase (`processedData\_0`) and quadrature (`processedData\_90`) components using the formula for the magnitude of a complex signal:

- `demodData = sqrt(processedData\_0^2 + processedData\_90^2)`.

- This step reconstructs the amplitude envelope of the modulated signal, independent of its phase.

---

#### 8. \*\*Amplitude Correction\*\*

```matlab

demodData = demodData \* 2 / modAmp;

```

- \*\*Correction Factor\*\*:

- The demodulated signal is scaled by `2 / modAmp` to correct for the amplitude of the reference signal.

- The factor of 2 accounts for the fact that the demodulation process (multiplication and filtering) typically halves the amplitude of the original signal.

- Dividing by `modAmp` normalizes the signal to the modulation amplitude, ensuring accurate scaling.

---

### \*\*Key Observations\*\*

1. \*\*Quadrature Demodulation\*\*:

- The function uses quadrature demodulation, which involves multiplying the raw signal by both the reference signal and its 90-degree phase-shifted version.

- This approach ensures robust demodulation, regardless of the phase difference between the raw signal and the reference.

2. \*\*Low-Pass Filtering\*\*:

- A 5th-order Butterworth low-pass filter is applied to remove high-frequency noise introduced during demodulation.

- Padding is used to reduce edge effects, improving filter performance.

3. \*\*Phase Shift Implementation\*\*:

- The 90-degree phase shift is implemented using `circshift`, with a negative shift (`-quarterPeriod`) to ensure correct alignment.

- The comment "update Michi (7/10/19)" suggests this is a corrected version of an earlier implementation.

4. \*\*Amplitude Correction\*\*:

- The final scaling (`2 / modAmp`) ensures the demodulated signal accurately reflects the original modulation amplitude.

5. \*\*Assumptions\*\*:

- The function assumes that `rawData` and `refData` are synchronized and sampled at the same rate.

- It assumes that `modFreq` and `modAmp` accurately describe the modulation characteristics.

---

### \*\*Purpose in Fiber Photometry Analysis\*\*

- This function is used to demodulate amplitude-modulated photometry signals, such as fluorescence data from dopamine sensors.

- In fiber photometry, signals are often modulated at a specific frequency (`modFreq`) to separate them from noise or other signals (e.g., isosbestic control signals).

- The demodulation process extracts the fluorescence signal of interest, which can then be used to compute ΔF/F or analyze neural activity.

---

### \*\*Potential Improvements\*\*

1. \*\*Input Validation\*\*:

- Add checks to ensure that `rawData` and `refData` have the same length and sampling rate.

- Validate that `modFreq` is within the Nyquist limit (`< sampleRate/2`).

2. \*\*Filter Cutoff Handling\*\*:

- If `lowCutoff` is empty, the function proceeds without filtering, which may leave noise in the signal. Consider adding a default cutoff or warning.

- Validate that `lowCutoff` is within a reasonable range.

3. \*\*Padding Optimization\*\*:

- The current padding uses the first `sampleRate` samples, which may not be optimal for all signals. Consider using mirrored padding or other methods to reduce edge effects.

4. \*\*Phase Shift Accuracy\*\*:

- The phase shift (`quarterPeriod`) is rounded to the nearest integer, which may introduce small errors. Consider interpolating the signal for more precise phase shifts.

5. \*\*Output Metadata\*\*:

- Add outputs for intermediate signals (e.g., `processedData\_0`, `processedData\_90`) for debugging or visualization.

- Include filter parameters or modulation details in the output for documentation.

6. \*\*Visualization\*\*:

- Add plotting functionality to visualize the raw, reference, and demodulated signals for quality control.

---

### \*\*Summary\*\*

This function performs quadrature demodulation of an amplitude-modulated photometry signal using a reference signal. It generates a 90-degree phase-shifted reference, multiplies both signals with the raw data, applies a low-pass Butterworth filter, and reconstructs the amplitude envelope. The demodulated signal is corrected for the modulation amplitude, making it suitable for downstream analysis (e.g., ΔF/F calculation). The function is robust but could benefit from input validation, optimized padding, and additional output options. It is a critical step in fiber photometry analysis for extracting fluorescence signals from modulated data.

This is a MATLAB function named `decimate` that is used to resample a signal at a lower rate after lowpass filtering. Here's a breakdown of the code:

\*\*Function Signature\*\*

The function takes four input arguments:

\* `idata`: the input signal to be decimated

\* `r`: the decimation factor (i.e., the ratio of the original sampling rate to the desired sampling rate)

\* `nfilt`: the order of the filter to be used (optional)

\* `option`: a string indicating whether to use an IIR (Infinite Impulse Response) filter or an FIR (Finite Impulse Response) filter (optional)

The function returns one output argument:

\* `odata`: the decimated signal

\*\*Input Validation\*\*

The function first checks the number of input arguments using `narginchk`. It then checks the output arguments using `nargoutchk`. If the input arguments are invalid, it throws an error.

The function also validates the input signal `idata` and the decimation factor `r` using the `validateinput` function. If either of these inputs is invalid, it throws an error.

\*\*Filtering and Decimation\*\*

The function then determines whether to use an IIR filter or an FIR filter based on the `option` input argument. If `option` is not provided, it defaults to using an IIR filter.

If an IIR filter is used, the function designs a Chebyshev Type I lowpass filter with the specified order `nfilt` and cutoff frequency `0.8/r`. It then filters the input signal in both directions using `filtfilt` to ensure that the filtered signal has zero phase.

If an FIR filter is used, the function designs a FIR filter with the specified order `nfilt` and cutoff frequency `1/r`. It then filters the input signal using `filter` and eliminates transients by prepending and appending the signal with its mirror image.

In both cases, the function then selects every `r`-th point from the filtered signal to produce the decimated signal `odata`.

\*\*Special Cases\*\*

The function handles several special cases:

\* If the decimation factor `r` is 1, the function simply returns the original input signal.

\* If the filter order `nfilt` is greater than 13, the function issues a warning about the potential unreliability of the results.

\* If the input signal is empty or sparse, or if the decimation factor is non-integer or non-positive, the function throws an error.

\*\*Helper Functions\*\*

The function uses several helper functions:

\* `filtmag\_db`: calculates the magnitude response of a filter in decibels at a given frequency.

\* `validateinput`: validates the input signal and decimation factor.

\* `cheby1`: designs a Chebyshev Type I lowpass filter.

\* `fir1`: designs a FIR filter.

\* `filter`: filters a signal using a filter.

\* `filtfilt`: filters a signal in both directions using a filter.

This MATLAB function, `demod\_and\_baseline`, appears to be designed to process data from a scientific experiment, likely involving optical imaging or photometry. The function takes a `SessionData` structure as input and returns two cell arrays, `Photo` and `Time`, which contain the processed data.

Here's a breakdown of the code:

\*\*Initialization\*\*

The function initializes several variables:

\* `lowCutoff`: a low-pass filter cutoff frequency (15 Hz)

\* `trials\_to\_skip`: the number of trials to skip (0)

\* `sampleRate`: the sampling rate of the data (extracted from `SessionData`)

\* `modAmpgreen` and `modFreqgreen`: the amplitude and frequency of the modulation signal (extracted from `SessionData`)

\* `DecimateFactor`: a decimation factor (100)

\*\*Data Processing\*\*

The function loops through each trial in the `SessionData` structure, skipping the first `trials\_to\_skip` trials. For each trial, it:

1. Extracts the raw data and reference data from the `SessionData` structure.

2. Demodulates the raw data using the `AP\_Demodulation` function (not shown in this code snippet).

3. Decimates the demodulated data using the `decimate` function.

4. Adjusts the first few samples of the demodulated data to avoid artifacts.

5. Fits a double exponential function to the demodulated data to account for bleaching.

6. Subtracts the fitted exponential from the demodulated data.

\*\*Signal Processing\*\*

The function applies additional signal processing steps to the demodulated data:

1. Applies a Savitzky-Golay filter to the data using `sgolayfilt`.

2. Fits another double exponential function to the filtered data to account for bleaching.

3. Subtracts the fitted exponential from the filtered data.

4. Applies a moving average filter to the data using `sgolayfilt` with a longer frame length.

5. Removes outliers from the data using `filloutliers`.

\*\*Gamma Normalization\*\*

The function applies gamma normalization to the processed data:

1. Fits a gamma distribution to the processed data using `fitdist`.

2. Discretizes the data into bins using `discretize`.

3. Applies the inverse cumulative distribution function (CDF) of the gamma distribution to the binned data using `icdf`.

\*\*Output\*\*

The function constructs the `Photo` and `Time` cell arrays by concatenating the processed data from each trial. The `Photo` array contains the normalized, filtered data, while the `Time` array contains the corresponding time values.

Overall, this function appears to be designed to process and normalize data from an optical imaging experiment, taking into account various sources of noise and artifacts.

This MATLAB function, `demod\_and\_baseline`, is designed for processing photometry data, specifically for analyzing dopamine signals. It performs demodulation, baseline correction, and signal normalization. Below is a detailed interpretation of the code, breaking it down into key sections and explaining its purpose in the context of photometry analysis.

---

### \*\*Overview\*\*

Photometry is a technique used to measure fluorescence signals, often from genetically encoded indicators like GCaMP or dLight, to study neural activity (e.g., dopamine signaling). This function processes raw photometry data, likely collected using a National Instruments DAQ (Nidaq), to extract meaningful signals. The steps include:

1. Demodulation of the raw signal to isolate the fluorescence component.

2. Correction for photobleaching (slow decay of fluorescence over time).

3. Baseline correction using filtering techniques.

4. Normalization of the signal using a gamma distribution.

5. Structuring the processed data back into trials for further analysis.

---

### \*\*Function Inputs and Outputs\*\*

- \*\*Input\*\*: `SessionData` - A structure containing trial data, including:

- `NidaqData`: Raw photometry data for each trial (likely two channels: signal and reference).

- `TrialSettings`: Settings such as sampling rate, LED modulation amplitude, and frequency.

- \*\*Outputs\*\*:

- `Photo`: Cell array containing processed photometry signals for each trial.

- `Time`: Cell array containing corresponding time vectors for each trial.

---

### \*\*Key Parameters\*\*

- `lowCutoff = 15`: Low-pass filter cutoff frequency (in Hz) for demodulation.

- `trials\_to\_skip = 0`: Number of initial trials to skip (can be adjusted if early trials are unreliable).

- `sampleRate`: Sampling rate of the DAQ (from `SessionData.TrialSettings`).

- `modAmpgreen` and `modFreqgreen`: Modulation amplitude and frequency of the LED used for excitation (likely for the green channel, e.g., 470 nm for dopamine sensors).

- `DecimateFactor = 100`: Factor by which the signal is downsampled to reduce computational load.

---

### \*\*Step-by-Step Interpretation\*\*

#### \*\*1. Initialization\*\*

- The function extracts the sampling rate, LED modulation amplitude, and frequency from `SessionData`.

- The decimated sampling rate (`decSR`) is calculated as `sampleRate / DecimateFactor`.

- Empty arrays (`signal`, `demodData`) are initialized to store processed data.

---

#### \*\*2. Trial-by-Trial Processing\*\*

The function loops through all trials in `SessionData.NidaqData`. For each trial:

##### \*\*a. Trials to Process (i > trials\_to\_skip)\*\*

- \*\*Demodulation\*\*:

- `rawDatagreen = SessionData.NidaqData{1,i}(:,1)`: Extracts the raw signal (likely the fluorescence channel).

- `refDatagreen = SessionData.NidaqData{1,i}(:,2)`: Extracts the reference signal (likely an isosbestic or control channel).

- `AP\_Demodulation`: A custom function (not shown) that demodulates the raw signal using the reference signal, sampling rate, LED modulation amplitude, frequency, and low-pass cutoff. This step isolates the fluorescence signal from noise and artifacts.

- The demodulated signal is downsampled using `decimate` by the `DecimateFactor`.

- \*\*Initial Artifact Correction\*\*:

- The first 5 samples of the demodulated signal are replaced with the mean of samples 4–7 (ignoring NaNs). This likely corrects for initial artifacts or instability in the demodulation process.

- \*\*Photobleaching Correction\*\*:

- A time vector (`xdata`) is created using `linspace(0, duration, ExpectedSize)`, where `duration = length(demodData{i}) / decSR`.

- Outliers in the demodulated signal (`ydata`) are removed using `rmoutliers`, and the signal is mean-subtracted.

- A single exponential function `F = @(x,xdata)x(1)\*exp(-x(2)\*xdata)` is fitted to the data using `lsqcurvefit` to model photobleaching (slow decay of fluorescence).

- Initial guess (`x0`) is set based on the maximum value of the outlier-removed signal and a decay rate of 0.005.

- The fitted exponential is subtracted from the signal to correct for photobleaching: `ydata = (ydata - F(xunc, xdata))`.

- \*\*Signal Concatenation\*\*:

- The corrected signal (`ydata`) is appended to the `signal` array for further processing.

##### \*\*b. Skipped Trials (i <= trials\_to\_skip)\*\*

- For skipped trials, the raw and reference data are still extracted, but the demodulated signal is replaced with zeros (same size as the processed signal).

- No signal is appended to `signal` for these trials.

---

#### \*\*3. Post-Processing of Concatenated Signal\*\*

After processing all trials, the concatenated signal (`signal`) is further refined:

- \*\*Smoothing\*\*:

- A Savitzky-Golay filter (`sgolayfilt`) is applied with polynomial order 2 and frame length 7 to smooth the signal and reduce noise.

- \*\*Global Photobleaching Correction\*\*:

- Similar to the trial-level correction, outliers are removed, and the signal is mean-subtracted.

- A single exponential is fitted to the smoothed signal to correct for any remaining photobleaching trends.

- The fitted exponential is subtracted from the signal.

- \*\*Baseline Correction\*\*:

- A Savitzky-Golay filter (`sgolayfilt`) with polynomial order 1 and a long frame length (`framelen = 0.5 \* decSR \* 60 + 1`, corresponding to 30 seconds) is applied to estimate the baseline.

- The baseline is subtracted from the signal: `ydata\_filt = ydata - sgf`.

- Outliers in the filtered signal are replaced using linear interpolation (`filloutliers`) with percentiles [0.1, 99.99].

- \*\*Gamma Normalization\*\*:

- A gamma distribution is fitted to the baseline-corrected signal (shifted to start at 0) using `fitdist`.

- If the shape parameter (`dist.a`) of the gamma distribution is greater than 25, it is printed (likely for debugging).

- The signal is discretized into bins based on quantiles (`discretize`).

- The inverse cumulative distribution function (CDF) of a standard gamma distribution (`pd = makedist('Gamma', 1, 1)`) is applied to normalize the signal: `ydata\_filt = pd.icdf(quants(dig\_bins))`.

- This step transforms the signal into a standardized distribution, likely to facilitate comparison across sessions or conditions.

---

#### \*\*4. Structuring Output\*\*

- The processed signal (`ydata\_filt`) is segmented back into trials:

- For trials to process (`i > trials\_to\_skip`):

- The corresponding segment of `ydata\_filt` is assigned to `Photo{i}`.

- A time vector (`Time{i}`) is created using `linspace(0, duration, ExpectedSize)`.

- For skipped trials (`i <= trials\_to\_skip`):

- `Photo{i} = NaN` and `Time{i} = 0` are assigned.

- The `begin` index is updated to track the start of each trial segment in `ydata\_filt`.

---

### \*\*Key Concepts and Context\*\*

1. \*\*Demodulation\*\*:

- Photometry often uses frequency modulation to separate the fluorescence signal from noise (e.g., motion artifacts, ambient light). The `AP\_Demodulation` function likely implements lock-in amplification or a similar technique to extract the signal at the LED modulation frequency.

2. \*\*Photobleaching Correction\*\*:

- Fluorescence signals decay over time due to photobleaching. The exponential fit corrects for this slow drift, ensuring that changes in the signal reflect neural activity rather than optical artifacts.

3. \*\*Baseline Correction\*\*:

- The Savitzky-Golay filter with a long frame length estimates the slow baseline drift, which is subtracted to isolate transient dopamine signals.

4. \*\*Gamma Normalization\*\*:

- The gamma distribution is used to normalize the signal, likely to account for variability in signal amplitude and noise across sessions. This step ensures that the processed signal is comparable across experiments.

5. \*\*Outlier Handling\*\*:

- Outliers are removed or replaced at multiple stages to improve robustness, as photometry data can be noisy due to motion, electrical interference, or other artifacts.

---

### \*\*Potential Improvements or Considerations\*\*

1. \*\*Custom Demodulation Function\*\*:

- The `AP\_Demodulation` function is not provided, so its implementation is unclear. Ensure it is optimized for your specific hardware and signal characteristics.

2. \*\*Photobleaching Model\*\*:

- The code uses a single exponential for photobleaching correction, but some systems may require a double exponential or other models. The commented-out double exponential (`F = @(x,xdata)x(1)\*exp(-x(2)\*xdata) + x(3)\*exp(-x(4)\*xdata)`) suggests this was considered.

3. \*\*Baseline Filter Length\*\*:

- The 30-second frame length for baseline correction may need adjustment depending on the dynamics of your dopamine signals. A longer frame (e.g., 200 seconds, as commented) could be used for slower baselines.

4. \*\*Gamma Normalization\*\*:

- The gamma normalization step is specific and may not be suitable for all analyses. Consider validating this step against other normalization methods (e.g., z-scoring, ΔF/F).

5. \*\*Error Handling\*\*:

- Add checks for missing or corrupted data in `SessionData.NidaqData`.

- Handle cases where the exponential fit fails or produces unrealistic parameters.

6. \*\*Performance\*\*:

- Downsampling by a factor of 100 (`DecimateFactor`) reduces computational load but may lose high-frequency information. Adjust this based on your signal dynamics and analysis needs.

---

### \*\*Conclusion\*\*

This function processes raw photometry data to extract clean dopamine signals by performing demodulation, photobleaching correction, baseline subtraction, and gamma normalization. It is designed for trial-based analysis and outputs processed signals and time vectors for each trial. The code is robust but may require customization depending on the specific experimental setup, signal characteristics, and analysis goals.

This MATLAB function, `MakeTrialEventsForaging`, is designed to process and refine trial event data for a foraging behavior experiment, likely involving animal subjects. It loads a pre-existing `TrialEvents.mat` file, removes unnecessary fields, and saves a cleaned version of the data as `TrialEventsAll.mat` (though the output file is named differently in the code). Below is a detailed interpretation of the code, explaining its purpose and functionality in the context of behavioral analysis.

---

### \*\*Overview\*\*

The function assumes that trial start timestamps have already been adjusted to align with the recording frame (e.g., synchronized with neural or photometry data). It processes the trial event data by:

1. Loading the `TrialEvents.mat` file generated by a previous function (`MakeTrialEventsNP`).

2. Removing unnecessary fields from the trial event structure to reduce file size and focus on relevant data.

3. Saving the cleaned data to a new file in a specified output directory.

The function does not create new fields as mentioned in the comments but instead focuses on cleaning and saving the data. The output is intended for further analysis, such as aligning behavioral events with neural recordings.

---

### \*\*Function Inputs and Outputs\*\*

- \*\*Inputs\*\*:

- `Directory`: Path to the directory containing the `TrialEvents.mat` file.

- `subj\_str`: A string identifier for the subject (e.g., animal ID), used to name the output file.

- `outdir`: Output directory where the cleaned data will be saved.

- \*\*Outputs\*\*:

- Saves a file named `<subj\_str>\_RecBehav.mat` in the `outdir` directory, containing the cleaned trial event structure (`TE\_save`).

---

### \*\*Step-by-Step Interpretation\*\*

#### \*\*1. Loading the Trial Event Data\*\*

```matlab

TE = load(Directory);

```

- The function loads the `TrialEvents.mat` file from the specified `Directory`.

- The loaded data is stored in the `TE` structure, which likely contains a field `SessionData` with trial event information.

#### \*\*2. Handling the Session Data\*\*

```matlab

try

TEbis = TE.SessionData;

catch

TEbis = SessionData;

end

```

- The function attempts to extract the `SessionData` field from `TE`.

- If `TE.SessionData` does not exist (e.g., due to a naming inconsistency), it falls back to a variable named `SessionData` (assumed to be in the workspace).

- This error handling suggests that the function is designed to be robust to variations in the input file structure, but relying on a workspace variable (`SessionData`) is risky and should be avoided in production code.

#### \*\*3. Truncating Trial Data\*\*

```matlab

nTrials = TEbis.nTrials - 1;

TEbis.LeftAmount = TEbis.LeftAmount(1:nTrials);

TEbis.RightAmount = TEbis.RightAmount(1:nTrials);

TEbis.TrialStartTimestamp = TEbis.TrialStartTimestamp(1:nTrials);

TEbis.TrialTypes = TEbis.TrialTypes(1:nTrials);

```

- The number of trials (`nTrials`) is reduced by 1, likely to exclude an incomplete or invalid last trial.

- Key fields are truncated to include only the first `nTrials` elements:

- `LeftAmount`: Likely the reward amount or probability associated with the left choice.

- `RightAmount`: Likely the reward amount or probability associated with the right choice.

- `TrialStartTimestamp`: Timestamps marking the start of each trial.

- `TrialTypes`: Likely a categorical variable indicating the type of trial (e.g., left vs. right reward, forced vs. free choice).

- Other fields in `TEbis` are not explicitly truncated, which could lead to inconsistencies if they have different lengths.

#### \*\*4. Removing Unnecessary Fields\*\*

```matlab

TE\_save = TEbis;

TE\_save = rmfield(TE\_save, 'RawEvents');

TE\_save = rmfield(TE\_save, 'NidaqData');

TE\_save = rmfield(TE\_save, 'Nidaq2Data');

TE\_save = rmfield(TE\_save, 'RawData');

TE\_save = rmfield(TE\_save, 'Settings');

TE\_save = rmfield(TE\_save, 'LeftValveTime');

TE\_save = rmfield(TE\_save, 'RightValveTime');

TE\_save = rmfield(TE\_save, 'TrialSettings');

TE\_save = rmfield(TE\_save, 'changebridge');

TE\_save = rmfield(TE\_save, 'subject');

TE\_save = rmfield(TE\_save, 'delays');

TE\_save = rmfield(TE\_save, 'bridgepos');

TE\_save = rmfield(TE\_save, 'nTrials');

TE\_save = rmfield(TE\_save, 'suprise');

TE\_save = rmfield(TE\_save, 'extraITI');

```

- A copy of `TEbis` is created as `TE\_save`.

- The function removes a list of fields that are deemed unnecessary for further analysis:

- `RawEvents`: Likely raw behavioral event data (e.g., lever presses, licks).

- `NidaqData` and `Nidaq2Data`: Likely raw data from National Instruments DAQ devices (e.g., photometry or neural recordings).

- `RawData`: Likely raw behavioral or experimental data.

- `Settings` and `TrialSettings`: Experiment configuration and trial-specific settings.

- `LeftValveTime` and `RightValveTime`: Likely timing data for reward delivery (e.g., valve opening times).

- `changebridge`, `bridgepos`, `suprise`, `extraITI`: Likely custom fields related to experimental design (e.g., inter-trial intervals, surprise trials, or positional data).

- `subject`: Subject identifier (redundant if included in the filename via `subj\_str`).

- `delays`: Likely delay times (e.g., between cue and reward).

- `nTrials`: Number of trials (redundant after truncation).

- Removing these fields reduces file size and focuses on essential trial-level data (e.g., `LeftAmount`, `RightAmount`, `TrialStartTimestamp`, `TrialTypes`).

#### \*\*5. Saving the Cleaned Data\*\*

```matlab

[status, msg, msgID] = mkdir(outdir);

save(fullfile(outdir, subj\_str + "\_RecBehav.mat"), '-struct', 'TE\_save')

```

- The function creates the output directory (`outdir`) if it does not exist using `mkdir`.

- The cleaned trial event structure (`TE\_save`) is saved as `<subj\_str>\_RecBehav.mat` in the `outdir` directory.

- The `-struct` option saves the fields of `TE\_save` directly as variables in the `.mat` file.

#### \*\*6. Display Message\*\*

```matlab

disp('Additional events created and Trial Event saved')

```

- A message is displayed to indicate that the process is complete.

- Note: The message is misleading because no additional events are created; the function only removes fields and saves the data.

---

### \*\*Key Concepts and Context\*\*

1. \*\*Purpose\*\*:

- The function is part of a pipeline for processing behavioral data from a foraging task, likely involving choices between left and right options with varying reward amounts or probabilities.

- It cleans the trial event data by removing raw and auxiliary fields, retaining only essential trial-level information for alignment with neural or photometry data.

2. \*\*Trial Truncation\*\*:

- Reducing `nTrials` by 1 suggests that the last trial may be incomplete or unreliable. This is common in behavioral experiments where the session ends prematurely.

3. \*\*Field Removal\*\*:

- The removed fields include raw data (`RawEvents`, `NidaqData`, etc.), experimental settings (`Settings`, `TrialSettings`), and custom fields (`changebridge`, `suprise`, etc.).

- This step reduces file size and focuses on processed trial-level data, but care must be taken to ensure that no critical information is lost.

4. \*\*Output File\*\*:

- The output file (`<subj\_str>\_RecBehav.mat`) is named to include the subject identifier, facilitating organization and analysis across multiple subjects.

---

### \*\*Potential Improvements or Considerations\*\*

1. \*\*Misleading Comments\*\*:

- The function comments and display message suggest that new fields are created, but the code only removes fields. Update the documentation to reflect the actual functionality.

2. \*\*Error Handling\*\*:

- The fallback to `SessionData` in the `try-catch` block is risky if `SessionData` is not defined in the workspace. Instead, validate the input file structure and provide clear error messages.

- Add checks for missing or inconsistent fields in `TEbis` (e.g., ensure all fields have the same length after truncation).

3. \*\*Trial Truncation\*\*:

- Truncating `nTrials` by 1 may not be appropriate for all experiments. Consider making this configurable or validating the last trial's completeness.

- Ensure that all relevant fields are truncated consistently, not just the four fields listed (`LeftAmount`, `RightAmount`, `TrialStartTimestamp`, `TrialTypes`).

4. \*\*Field Removal\*\*:

- Removing fields like `subject` and `nTrials` may cause issues if downstream analyses require this information. Consider retaining essential metadata.

- Add a configuration option to specify which fields to remove, allowing flexibility for different analyses.

5. \*\*Output File Naming\*\*:

- The output file is named `<subj\_str>\_RecBehav.mat`, but the comments mention `TrialEventsAll.mat`. Standardize the naming convention and update the documentation.

6. \*\*Validation\*\*:

- Validate the input `Directory` and `outdir` to ensure they exist and are accessible.

- Check that `TE\_save` contains the expected fields before saving.

7. \*\*Performance\*\*:

- For large datasets, consider optimizing memory usage by clearing unnecessary variables (e.g., `TE`, `TEbis`) after processing.

---

### \*\*Conclusion\*\*

This function processes trial event data for a foraging behavior experiment by loading `TrialEvents.mat`, truncating trials, removing unnecessary fields, and saving the cleaned data as `<subj\_str>\_RecBehav.mat`. It is designed to prepare behavioral data for alignment with neural or photometry recordings, but it has some limitations, such as misleading documentation, risky error handling, and potential loss of critical metadata. With minor improvements, it can be made more robust and flexible for various experimental designs.

This Python code is designed to process behavioral data from a foraging task, likely involving animal subjects, and organize it into a structured DataFrame for further analysis. The main function, `proc\_foraging\_df`, processes trial-by-trial data to track reward amounts, trial switches, and other behavioral metrics. Additional code handles file loading, sorting by datetime, and processing multiple sessions for multiple subjects. Below is a detailed interpretation of the code, explaining its purpose and functionality in the context of behavioral analysis.

---

### \*\*Overview\*\*

The code is part of a pipeline for analyzing behavioral data from a foraging task, where animals make choices (e.g., left vs. right) to obtain rewards. The main components are:

1. \*\*`proc\_foraging\_df`\*\*: Processes trial-by-trial behavioral data to track reward amounts, trial switches, and other metrics, adding new columns to the DataFrame.

2. \*\*`extract\_datetime`\*\*: Extracts and parses datetime information from file names for sorting purposes.

3. \*\*File Loading and Processing\*\*: Loads behavioral data files, sorts them by datetime, and processes each session using `proc\_foraging\_df`.

The code is designed for a specific experimental setup (e.g., IL6-KD condition) and handles multiple subjects and sessions, organizing the processed data into a list of DataFrames.

---

### \*\*Function: `proc\_foraging\_df`\*\*

This function processes a behavioral DataFrame (`behav\_df`) to extract trial-level metrics, such as trial switches, reward amounts, and probe pokes.

#### \*\*Inputs and Outputs\*\*

- \*\*Inputs\*\*:

- `behav\_df`: A pandas DataFrame containing trial-by-trial data with columns:

- `LeftAmount`: Reward amount or probability for the left choice.

- `RightAmount`: Reward amount or probability for the right choice.

- `subj\_name`: Subject identifier (e.g., animal ID).

- `condition`: Experimental condition (e.g., "IL6-KD").

- `session`: Session number.

- \*\*Output\*\*:

- Returns the modified `behav\_df` with additional columns for trial metrics.

#### \*\*Step-by-Step Interpretation\*\*

##### \*\*1. Initialization\*\*

```python

n\_trials = 0

n\_epochs = -1

switched = 0.0

n\_rewards = -1

trial\_number = []

reward\_number = []

epoch\_number = []

starting\_rsize = []

reward\_amount = []

probe\_poke = ["first"]

left = []

```

- Initialize counters and lists to track trial metrics:

- `n\_trials`: Number of trials (starts at 0).

- `n\_epochs`: Number of epochs (starts at -1, incremented later).

- `switched`: Flag indicating whether the trial switched sides (0.0 = no switch, 1.0 = switch).

- `n\_rewards`: Number of rewards (starts at -1, incremented later).

- Lists to store trial-level data:

- `trial\_number`: Trial index.

- `reward\_number`: Reward index.

- `epoch\_number`: Epoch index within a trial.

- `starting\_rsize`: Initial reward size for the current trial.

- `reward\_amount`: Actual reward amount for each epoch.

- `probe\_poke`: List to track probe pokes (starts with "first", later updated with numeric values).

- `left`: List to track switches (same as `switched`).

##### \*\*2. Determine Initial Side and Reward Size\*\*

```python

if ~np.isnan(behav\_df.iloc[0]['LeftAmount']) and ~np.isnan(behav\_df.iloc[1]['LeftAmount']):

current\_side = 'left'

rsize = behav\_df.iloc[0]['LeftAmount']

elif ~np.isnan(behav\_df.iloc[0]['RightAmount']) and ~np.isnan(behav\_df.iloc[1]['RightAmount']):

current\_side = 'right'

rsize = behav\_df.iloc[0]['RightAmount']

elif ~np.isnan(behav\_df.iloc[0]['LeftAmount']) and np.isnan(behav\_df.iloc[1]['LeftAmount']):

current\_side = 'left'

rsize = behav\_df.iloc[0]['LeftAmount']

switched = 1.0

elif ~np.isnan(behav\_df.iloc[0]['RightAmount']) and np.isnan(behav\_df.iloc[1]['RightAmount']):

current\_side = 'right'

rsize = behav\_df.iloc[0]['RightAmount']

switched = 1.0

```

- Determine the initial side (`current\_side`) and reward size (`rsize`) based on the first two rows of `behav\_df`:

- If `LeftAmount` is non-NaN for both rows, start on the left side.

- If `RightAmount` is non-NaN for both rows, start on the right side.

- If `LeftAmount` is non-NaN for the first row but NaN for the second, start on the left side and mark as switched.

- If `RightAmount` is non-NaN for the first row but NaN for the second, start on the right side and mark as switched.

- The `switched` flag is set to 1.0 if the second row indicates a switch (e.g., NaN on the current side).

##### \*\*3. Process Each Row (Epoch)\*\*

```python

for ind, row in behav\_df.iterrows():

left\_amt = row['LeftAmount']

right\_amt = row['RightAmount']

n\_epochs += 1

n\_rewards += 1

```

- Iterate through each row (epoch) in `behav\_df`.

- Increment `n\_epochs` and `n\_rewards` for each row.

##### \*\*4. Detect Trial Switches\*\*

```python

if np.isnan(left\_amt) and (current\_side == 'left'):

assert (np.isnan(right\_amt) == False)

rsize = right\_amt

current\_side = 'right'

switched = 1.0

n\_trials += 1

n\_epochs = 0

elif np.isnan(right\_amt) and (current\_side == 'right'):

assert (np.isnan(left\_amt) == False)

rsize = left\_amt

current\_side = 'left'

switched = 1.0

n\_trials += 1

n\_epochs = 0

else:

switched = 0.0

```

- Detect trial switches based on NaN values:

- If `left\_amt` is NaN and the current side is left, switch to the right side, update `rsize`, mark as switched, increment `n\_trials`, and reset `n\_epochs`.

- If `right\_amt` is NaN and the current side is right, switch to the left side, update `rsize`, mark as switched, increment `n\_trials`, and reset `n\_epochs`.

- Otherwise, no switch occurs (`switched = 0.0`).

##### \*\*5. Process Reward Amount and Probe Pokes\*\*

```python

if current\_side == 'left':

assert(~np.isnan(left\_amt))

if (len(reward\_amount) > 0) & (probe\_poke[-1] == 0):

if (switched == 0) & (left\_amt > (reward\_amount[-1] + .5)):

probe\_poke.append(1)

elif (switched == 0) & (left\_amt < (.8\*reward\_amount[-1] - .5)):

probe\_poke.append(-2)

else:

probe\_poke.append(0)

else:

probe\_poke.append(0)

reward\_amount.append(left\_amt)

elif current\_side == 'right':

assert(~np.isnan(right\_amt))

if (len(reward\_amount) > 0) & (probe\_poke[-1] == 0):

if (switched == 0) & (right\_amt > (reward\_amount[-1])+.5):

probe\_poke.append(1)

elif (switched == 0) & (right\_amt < (.8\*reward\_amount[-1] - .5)):

probe\_poke.append(-2)

else:

probe\_poke.append(0)

else:

probe\_poke.append(0)

reward\_amount.append(right\_amt)

```

- Process reward amounts and probe pokes based on the current side:

- For the left side:

- Ensure `left\_amt` is non-NaN.

- If the previous `probe\_poke` was 0 and no switch occurred, check for significant changes in reward amount:

- If `left\_amt` increases by more than 0.5, mark as probe poke (1).

- If `left\_amt` decreases by more than 20% minus 0.5, mark as probe poke (-2).

- Otherwise, mark as no probe poke (0).

- Append `left\_amt` to `reward\_amount`.

- For the right side:

- Similar logic as the left side, using `right\_amt`.

- The `probe\_poke` logic detects significant changes in reward amount, likely to identify exploratory or unexpected behavior.

##### \*\*6. Update Trial Metrics\*\*

```python

left.append(switched)

reward\_number.append(n\_rewards)

starting\_rsize.append(rsize)

trial\_number.append(n\_trials)

epoch\_number.append(n\_epochs)

```

- Append trial metrics to their respective lists:

- `left`: Switch flag (`switched`).

- `reward\_number`: Current reward index.

- `starting\_rsize`: Initial reward size for the trial.

- `trial\_number`: Current trial index.

- `epoch\_number`: Current epoch index within the trial.

##### \*\*7. Update DataFrame\*\*

```python

assert(len(left) == len(behav\_df['LeftAmount']))

behav\_df['reward\_number'] = reward\_number

behav\_df['name'] = subj\_name

behav\_df['condition'] = condition

behav\_df['session\_number'] = session

behav\_df['switch'] = left

behav\_df['switch\_prob'] = behav\_df['switch'].shift(-1)

behav\_df['stay\_prob'] = 1 - behav\_df['switch\_prob']

behav\_df['trial\_number'] = trial\_number

behav\_df['poke\_number'] = epoch\_number

behav\_df['trial\_type'] = starting\_rsize

behav\_df['reward\_amount'] = np.around(np.array(reward\_amount), decimals=4)

return behav\_df

```

- Validate that the length of `left` matches the number of rows in `behav\_df`.

- Add new columns to `behav\_df`:

- `reward\_number`: Reward index.

- `name`: Subject identifier.

- `condition`: Experimental condition.

- `session\_number`: Session number.

- `switch`: Switch flag.

- `switch\_prob`: Probability of switching (shifted by -1, i.e., next trial's switch).

- `stay\_prob`: Probability of staying (1 - `switch\_prob`).

- `trial\_number`: Trial index.

- `poke\_number`: Epoch index.

- `trial\_type`: Initial reward size for the trial.

- `reward\_amount`: Actual reward amount (rounded to 4 decimals).

- Return the modified `behav\_df`.

---

### \*\*Function: `extract\_datetime`\*\*

This function extracts and parses datetime information from file names for sorting purposes.

#### \*\*Inputs and Outputs\*\*

- \*\*Input\*\*:

- `file\_name`: Path to a file with a datetime string in the name (e.g., "01-Jan-2023\_12-00-00\_...").

- \*\*Output\*\*:

- Returns a `datetime` object parsed from the file name.

#### \*\*Step-by-Step Interpretation\*\*

```python

date\_format = "%d-%b-%Y %H:%M:%S"

date\_time\_str = file\_name.split('/')[-1].split('\_')[0]

return datetime.datetime.strptime(date\_time\_str, date\_format)

```

- Define the date format (`%d-%b-%Y %H:%M:%S`, e.g., "01-Jan-2023 12:00:00").

- Extract the datetime string from the file name:

- Split by '/' to get the file name.

- Split by '\_' to get the datetime part (assumed to be the first segment).

- Parse the datetime string into a `datetime` object and return it.

---

### \*\*File Loading and Processing\*\*

This section loads behavioral data files, sorts them by datetime, and processes each session using `proc\_foraging\_df`.

#### \*\*Step-by-Step Interpretation\*\*

##### \*\*1. Define Subject Names and Find Files\*\*

```python

name = ['RbalbP\_W1\_LH', 'RbalbP\_W1\_NH', 'RbalbP\_W1\_RH', ...]

behavfiles = []

for i in name:

ind\_files = glob.glob("/Volumes/Aelita/IL6R-photo\_v2/processed/" + i + "/\*RecBehav.mat")

behavfiles.append(ind\_files)

```

- Define a list of subject names (`name`), likely corresponding to different animals and conditions.

- Use `glob.glob` to find all `\*RecBehav.mat` files for each subject in the specified directory.

- Store the file lists in `behavfiles`.

##### \*\*2. Sort Files by Datetime\*\*

```python

for i in range(len(behavfiles)):

dated\_files = [(extract\_datetime(fn), fn) for fn in behavfiles[i]]

dated\_files.sort()

behavfiles[i] = [fn for dt, fn in dated\_files]

```

- For each subject, sort the files by datetime:

- Use `extract\_datetime` to parse the datetime from each file name.

- Create a list of tuples `(datetime, file\_name)`.

- Sort the list by datetime.

- Update `behavfiles` with the sorted file names.

##### \*\*3. Process Each Session\*\*

```python

df\_tot = []

for j in range(len(behavfiles)):

dfs = []

for i, file in enumerate(behavfiles[j]):

SessionData = loadmat(file, squeeze\_me=True)

n\_trials = len(SessionData['LeftAmount'])

behav\_df = pd.DataFrame.from\_dict({

'LeftAmount': SessionData['LeftAmount'],

'RightAmount': SessionData['RightAmount'],

'name': [name[j]]\*n\_trials,

'session': [i]\*n\_trials

})

df = proc\_foraging\_df(behav\_df.reset\_index(), subj\_name=name[j],

condition="IL6-KD", session=i)

dfs.append(df)

df\_tot.append(dfs)

```

- Initialize `df\_tot` to store processed DataFrames for all subjects.

- For each subject and session:

- Load the `.mat` file using `loadmat` with `squeeze\_me=True` to remove singleton dimensions.

- Create a DataFrame (`behav\_df`) with `LeftAmount`, `RightAmount`, `name`, and `session`.

- Process the DataFrame using `proc\_foraging\_df`.

- Append the processed DataFrame to `dfs`.

- Append `dfs` to `df\_tot`, resulting in a nested list of DataFrames (subjects × sessions).

---

### \*\*Key Concepts and Context\*\*

1. \*\*Purpose\*\*:

- The code processes behavioral data from a foraging task, tracking trial switches, reward amounts, and probe pokes.

- It is designed for a specific experimental setup (e.g., IL6-KD condition) and handles multiple subjects and sessions.

2. \*\*Trial Switches\*\*:

- Trial switches are detected based on NaN values in `LeftAmount` or `RightAmount`, indicating a change in the rewarded side.

- Switches reset the epoch counter and update the trial index.

3. \*\*Probe Pokes\*\*:

- Probe pokes are detected based on significant changes in reward amount (e.g., increase by >0.5 or decrease by >20% minus 0.5).

- This likely identifies exploratory or unexpected behavior.

4. \*\*File Sorting\*\*:

- Files are sorted by datetime to ensure sessions are processed in chronological order, which is critical for longitudinal analyses.

5. \*\*Output\*\*:

- The processed data is organized into a nested list of DataFrames (`df\_tot`), facilitating further analysis (e.g., trial-averaged responses, statistical comparisons).

---

### \*\*Potential Improvements or Considerations\*\*

1. \*\*Error Handling\*\*:

- Add checks for missing or corrupted files in `behavfiles`.

- Validate the structure of `SessionData` (e.g., ensure `LeftAmount` and `RightAmount` exist).

- Handle cases where `proc\_foraging\_df` fails (e.g., due to NaN values or assertion errors).

2. \*\*Probe Poke Logic\*\*:

- The `probe\_poke` logic is complex and may need validation. Consider simplifying or documenting the thresholds (e.g., 0.5, 0.8).

- Handle edge cases (e.g., first trial, missing data).

3. \*\*File Sorting\*\*:

- Ensure the datetime extraction logic is robust to variations in file naming conventions.

- Add error handling for invalid datetime strings.

4. \*\*Output\*\*:

- Consider saving `df\_tot` to a file (e.g., `.csv`, `.pkl`) for persistence and reproducibility.

- Add metadata (e.g., processing date, parameters) to the output.

5. \*\*Performance\*\*:

- For large datasets, optimize memory usage by processing files in batches or using parallel processing.

- Consider using `dask` or `multiprocessing` for scalability.

---

### \*\*Conclusion\*\*

This code processes behavioral data from a foraging task, tracking trial switches, reward amounts, and probe pokes, and organizes the data into structured DataFrames. It handles multiple subjects and sessions, sorting files by datetime for chronological processing. With minor improvements, such as enhanced error handling and output persistence, it can be made more robust and scalable for various experimental designs.