# Promising Methods to Handling Numerical Ties in PCPs

The paper outlines six methods to resolve numerical ties in datasets, enhancing data visualization and analysis. **Attempt 1: Incremental Offset** adds a small value to tied figures based on their placement within a group to maintain the original order. **Attempt 2: Jitter Spacing** randomly modifies tied values to break the ties while preserving the overall structure. **Attempt 3: Random Rank Shuffle** assigns random ranks to tied values, effectively functioning for both numbers and categories. **Attempt 4: Cascade Rank Tie-Breaker** utilizes previous rankings to resolve ties, ensuring the order is clear and comprehensible. **Attempt 5: Tie Breaking the Band** applies small offsets to distinguish tied values while retaining the original sequence. **Attempt 6: Tie Breaking the Band with Auto Fraction** involves automatic detection of data types, robust handling of unusual data values, and adjustments based on the data's distribution. Each method includes code examples that elucidate its advantages and disadvantages for various data types.

## Determine the numerical ties in the dataset

Let's determine the number of numerical ties in the dataset.

| cat_name | bill_length_mm | bill_depth_mm | flipper_length_mm | body_mass_g | year |
|---|---|---|---|---|---|
| Tie Counts | 170.0 | 254.0 | 279.0 | 240.0 | 330.0 |
| Tie Percentages | 51.1 | 76.3 | 83.8 | 72.1 | 99.1 |

## Attempt 1: Incremental Offset (Sequential Tie Handling)

*Equation*:

$$value_i' = value_i + \Delta \times k$$

Where:

- $\Delta$ is a small offset.
- $k$ is the relative position within the group of ties.

*Description*:

Adds a small incremental offset to each tied value based on its relative position in the tie group. Ensures that ties are visually and numerically distinct while maintaining an order.

*Advantages*:

- Maintains the tie order naturally.
- Suitable for ordered or sequential data.

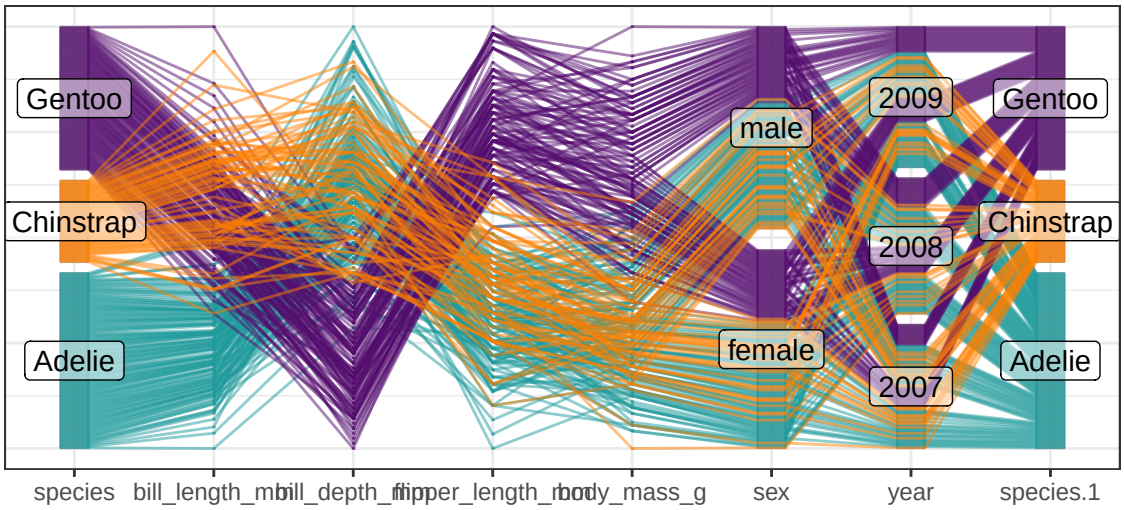
*Implementation*:

Ideal when you want to preserve some form of sequential relationship among tied values without introducing randomness.

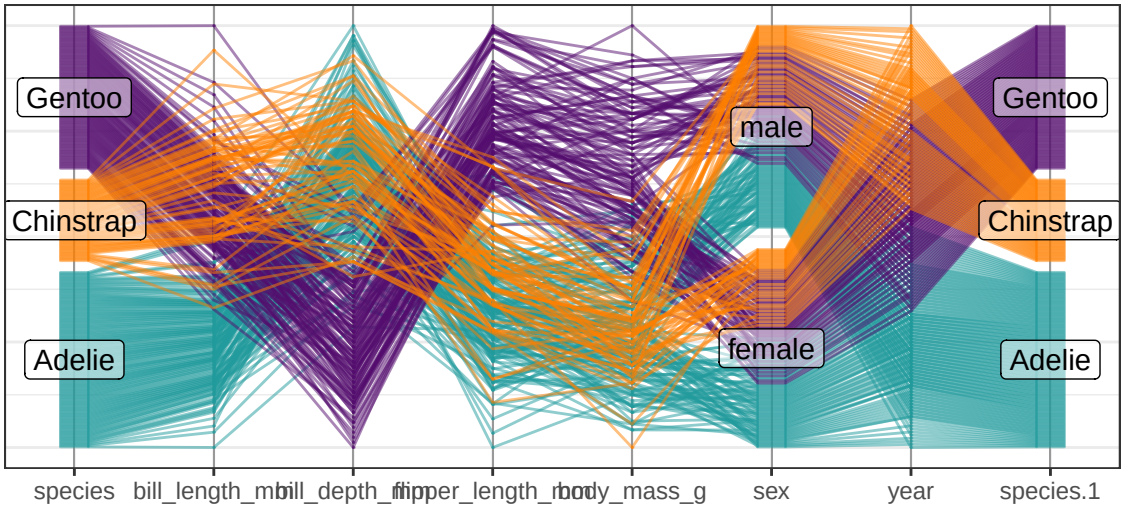
```r
add_space_to_ties <- function(data, tie_spacing = 0.01) {
  # Identify numeric columns
  num_cols <- sapply(data, is.numeric)

  # Adjust numerical ties by adding incremental spacing using
  # Group Averages Over Level Combinations of Factors

  data[num_cols] <- lapply(data[num_cols], function(col) {
    tie_indices <- ave(col, col, FUN = seq_along)
    col + (tie_indices - 1) * tie_spacing
  })

  return(data)
}
```

*PCP Implementation*

Unadjusted PCP of Palmer Penguins




Incremental Offset PCP of Palmer Penguins




### Attempt 2: Jitter Spacing (Numerical Tie Handling)

*Equation*:

$$value_i' = value_i + \epsilon_i$$

where:

- $value_i'$ is the jittered value of $value_i$,
- $\epsilon_i \sim \mathcal{U}(-\delta, \delta)$, a small random uniform jitter.

*Description*:

This method breaks numerical ties by adding a tiny random value ($\epsilon$) within a specified range $(-\delta, \delta)$. It ensures that tied values are slightly separated, preserving the overall data structure while improving visualization clarity.

*Advantages*:

- Simple and intuitive.
- Works well for small datasets with occasional ties.

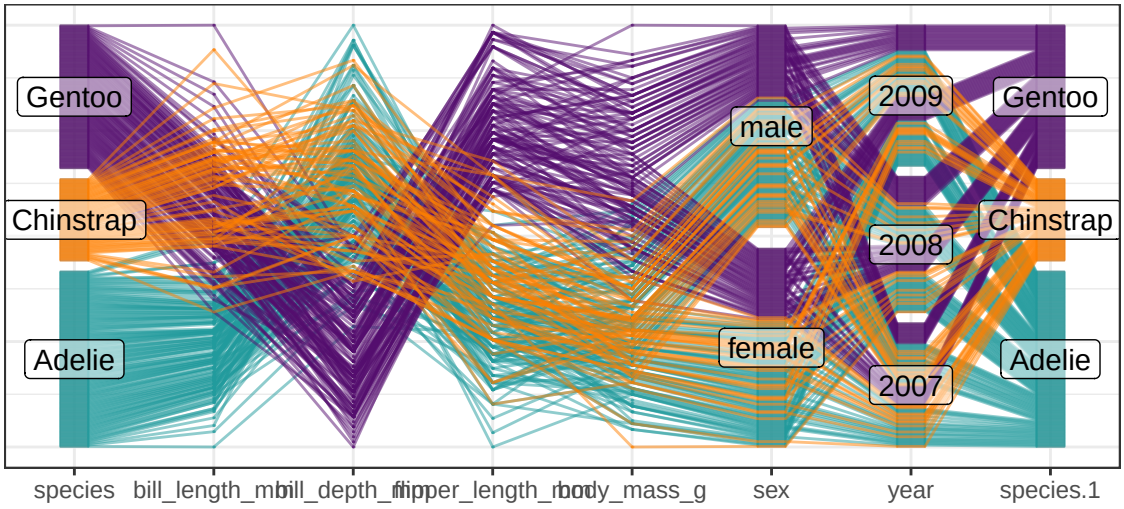
*Implementation*:

Applied to numeric columns. This is typically done before visualizing parallel coordinate plots or when preparing data for algorithms sensitive to ties.

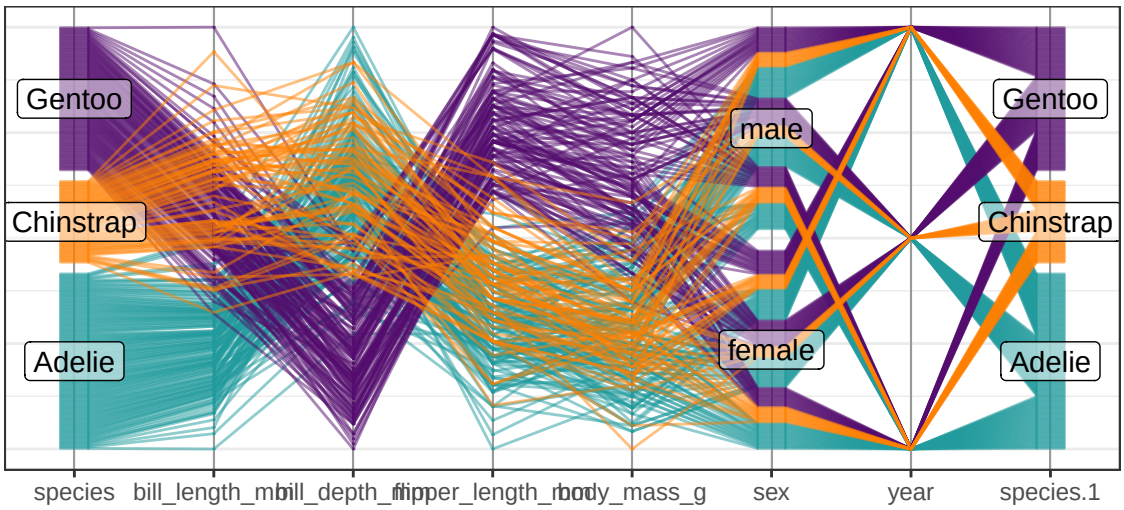
```r
adjust_ties <- function(df, cols_to_adjust, epsilon = 1e-5) {
  df_adjusted <- df

  for (col in cols_to_adjust) {
    if (is.numeric(df[[col]])) {
      ties <- duplicated(df[[col]]) | duplicated(df[[col]], fromLast = TRUE)
      unique_ties <- unique(df[[col]][ties])

      for (tie in unique_ties) {
        tie_indices <- which(df[[col]] == tie)
        adjustment <- seq(-epsilon, epsilon, length.out = length(tie_indices))
        df_adjusted[tie_indices, col] <- df[[col]][tie_indices] + adjustment
      }
    }
  }

  return(df_adjusted)
}
```

*PCP Implementation*

Unadjusted PCP of Palmer Penguins




Jitter Spacing PCP of Palmer Penguins




### Attempt 3: Random Rank Shuffle (Numerical and Categorical Tie Handling)

*Equation*:

$$Rank_i = rank(Value_i, ties.method = "random")$$

*Description*:

Tied values are assigned random ranks, ensuring that the order of ties is resolved randomly while maintaining the distribution. For categorical data, this is applied to ordinal values derived from categories, which are then shuffled if tied.

*Advantages*:

- Guarantees tie resolution without adding noise to the original data.
- Compatible with both numerical and categorical data.

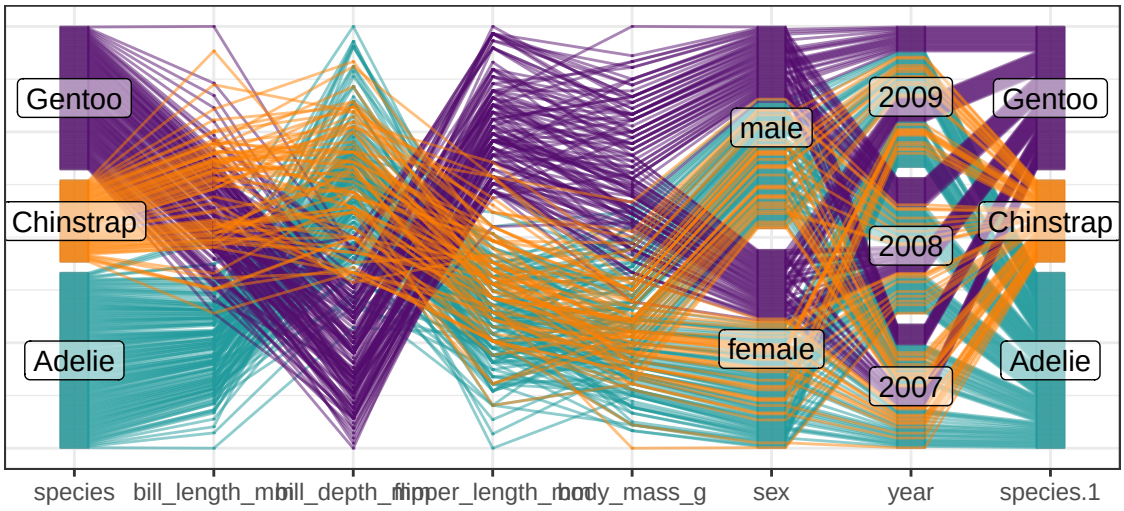
*Implementation*:

Useful for both numeric and categorical data before plotting or modeling to minimize ambiguity in ties. This method aligns well with the ranking logic used in ggpcp.

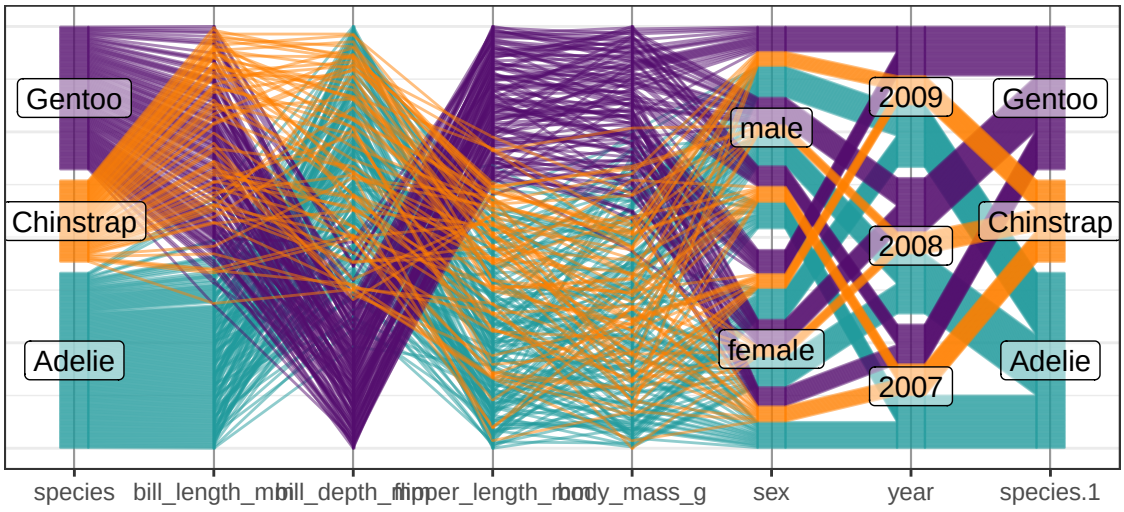
```r
apply_tie_breaking <- function(df, cols_to_adjust) {
  df_adjusted <- df

  for (col in cols_to_adjust) {
    if (is.numeric(df[[col]])) {
      # Rank the values with ties.method = "random"
      ranks <- rank(df[[col]], ties.method = "first")
      # Scale ranks back to the original range
      min_val <- min(df[[col]], na.rm = TRUE)
      max_val <- max(df[[col]], na.rm = TRUE)
      df_adjusted[[col]] <- scales::rescale(ranks, to = c(min_val, max_val))
    }
  }

  return(df_adjusted)
}
```

*PCP Implementation*

Unadjusted PCP of Palmer Penguins




Random Rank Shuffle PCP of Palmer Penguins




### Attempt 4: "Cascade Rank Tie-Breaker" Method

*Key Equations*:

1. Initial Rank Assignment:

Assign initial ranks to each variable, handling ties using a user-specified method.

$$R_{i,j} = \begin{cases} rank(x_{i,j}, \text{ties.method}) & \text{if } x_{i,j} \text{ is numeric} \\ as.numeric(foctor(x_{i,j})) & \text{if } x_{i,j} \text{ is categorical} \end{cases}$$

2. Sequential Rank Adjustment Based on Previous Variables:

Refine ranks by incorporating information from preceding variables to break ties and maintain a hierarchical influence.

$$NormPrev_{i,j-1} = \begin{cases} \frac{R_{i,j-1} - min(R_{i,j-1})}{max(R_{i,j-1}) - min(R_{i,j-1})} & \text{if method} = \text{"range"} \\ \frac{R_{i,j-1} - min(R_{i,j-1})}{\sigma(R_{i,j-1})} & \text{if method} = \text{"sd"} \end{cases}$$

3. Adjusted Current Rank:

Incorporate the normalized previous rank into the current rank:

$$R'_{i,j} = \begin{cases} R_{i,j} & \text{if } j = 1 \\ R'_{i,j} + \epsilon \times NormPrev_{i,j-1} & \text{if } j > 1 \end{cases}$$

- $\epsilon$: A small constant (e.g., 0.01) to control the influence of the previouse rank.

4. Final Data Arrangement for Visualization

Combine the original data with adjusted ranks and arrange the dataset to reflect the unique ordering derived from the "Cascade Rank-Tie Breaker" method.

$$Ranked\ data = original\ data \cup R' \text{ and sorted by } R'_1, R'_2, ..., R'_n$$

*Description*:

The "Cascade Rank-Tie Breaker" method systematically assigns ranks to each variable in a dataset, ensuring that ties are resolved by leveraging information from previous variables. This hierarchical approach maintains the integrity and relative importance of variables, making it particularly useful for data visualization techniques like Parallel Coordinate Plots.

*Advantages*:

- **Hierarchical Influence**: Earlier variables in the dataset have a stronger impact on the ranking, preserving the logical flow and importance of variables.
- **Flexible Tie Handling**: Users can specify different ties.method options (e.g., "min", "average") to control how ties are resolved during the initial ranking.
- **Scalability**: Efficiently handles multiple variables and large datasets by sequentially adjusting ranks. Enhanced Visualization: Facilitates clearer and more informative multivariate visualizations by ensuring a unique and meaningful ordering of data points.
- **Customization**: Allows users to choose between different normalization methods ("range", "sd") and to define custom normalization functions, tailoring the method to specific analytical needs.

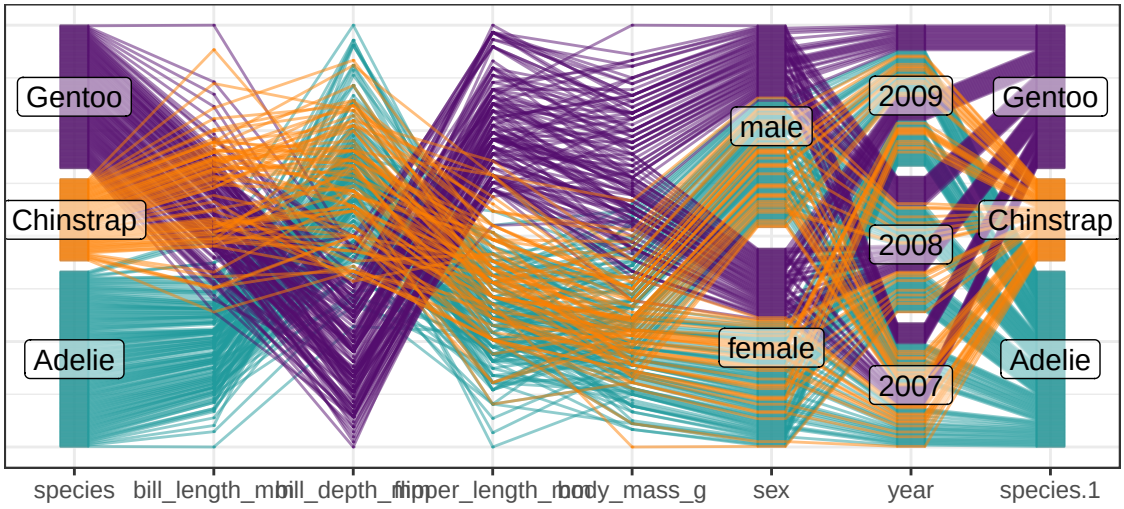
*Implementation*:

```r
cascade_tie_break <- function(data, method = c("range", "sd"), ties_method = "min") {
  # Match the 'method' argument to ensure it's one of the allowed options
  method <- match.arg(method)

  # Validate the 'ties_method' argument to ensure it's a valid option for the rank function
  valid_ties_methods <- c("average", "first", "random", "max", "min")
  if (!ties_method %in% valid_ties_methods) {
    stop(paste("Invalid ties_method. Choose one of:", paste(valid_ties_methods, collapse = ", ")))
  }

  # Initialize a dataframe to store ranks, excluding any existing 'rank' columns
  rank_df <- data.frame(id = 1:nrow(data))
```

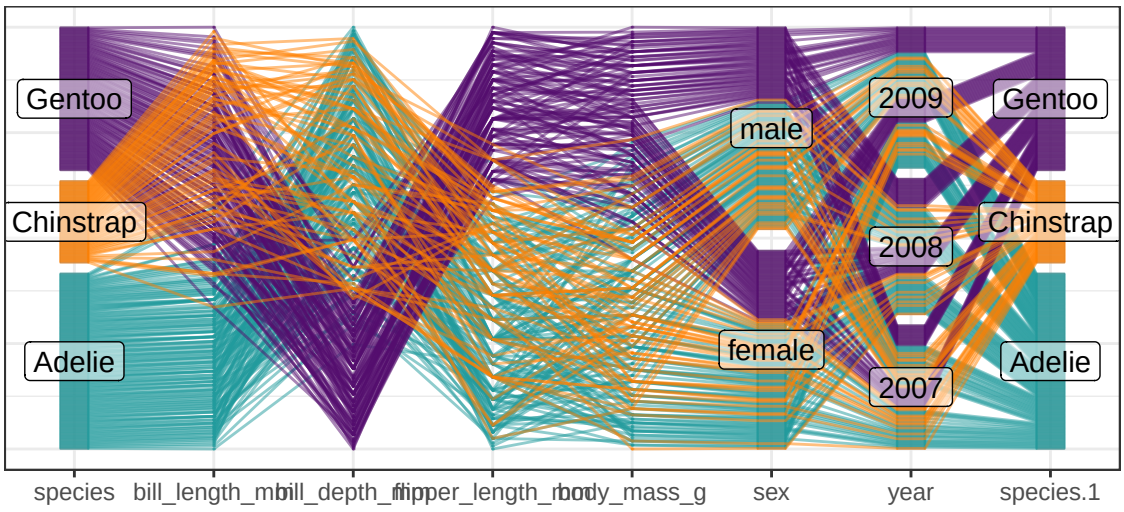
```r
  # Iterate over each column to assign ranks
  for (i in seq_along(data)) {
    var <- data[[i]]

    # Determine if the variable is numeric or categorical
    if (is.numeric(var)) {
      # Apply the specified ties.method in the rank function
      current_rank <- rank(var, ties.method = ties_method)
    } else {
      # For categorical variables, convert factors to numeric ranks based on their levels
      current_rank <- as.numeric(as.factor(var))
    }

    # If not the first variable, adjust ranks based on the previous variable
    if (i > 1) {
      prev_rank <- rank_df[[i - 1]]

      # Normalize the previous rank based on the selected method
      if (method == "range") {
        norm_prev_rank <- (prev_rank - min(prev_rank)) / (max(prev_rank) - min(prev_rank))
      } else if (method == "sd") {
        norm_prev_rank <- (prev_rank - min(prev_rank)) / sd(prev_rank)
      }

      # Adjust the current rank by adding a small fraction of the normalized previous rank
      # This helps in breaking ties based on the previous variable
      current_rank <- current_rank + norm_prev_rank * 0.01
    }

    # Assign the adjusted rank to the rank dataframe
    rank_df[[i]] <- current_rank
  }

  # Combine the original data with the rank dataframe, excluding the 'id' column
  ranked_data <- bind_cols(data, rank_df[ , -1])

  # Arrange the data based on the cumulative adjusted ranks to establish a unique ordering
  ranked_data <- ranked_data %>%
    arrange(across(starts_with("rank")))

  return(ranked_data)
}
```

*PCP Implementation*

Unadjusted PCP of Palmer Penguins




Cascade Ranked PCP of Palmer Penguins




### Attempt 5: Tie Breaking the Band (Resolving Numerical Ties with Incremental Offset)

*Equation*:

The incremental offset technique for numerical tie-breaking is defined as:

$$v_i' = v_i + \Delta * k$$

Where:

$v_i'$: Adjusted value for the $i$-th data point. $v_i$: Original value for the $i$-th data point. $\Delta$: A small incremental offset (e.g., 0.01). $k$: The relative position of the value within the tie group.

*Advantages*:

- Maintains Sequential Order: Ensures that the original order of tied values is preserved.
- Simple and Deterministic: The adjustment is straightforward and avoids randomness.
- Improves Visualization Clarity: Slight differences between tied values enhance the interpretability of parallel coordinate plots (PCPs).
- Scalable: Handles large datasets efficiently, provided ties are reasonably distributed.

*Disadvantages*:

- Fixed Offset May Bias Results: Applying a uniform incremental offset could introduce a small, consistent bias in downstream analyses.

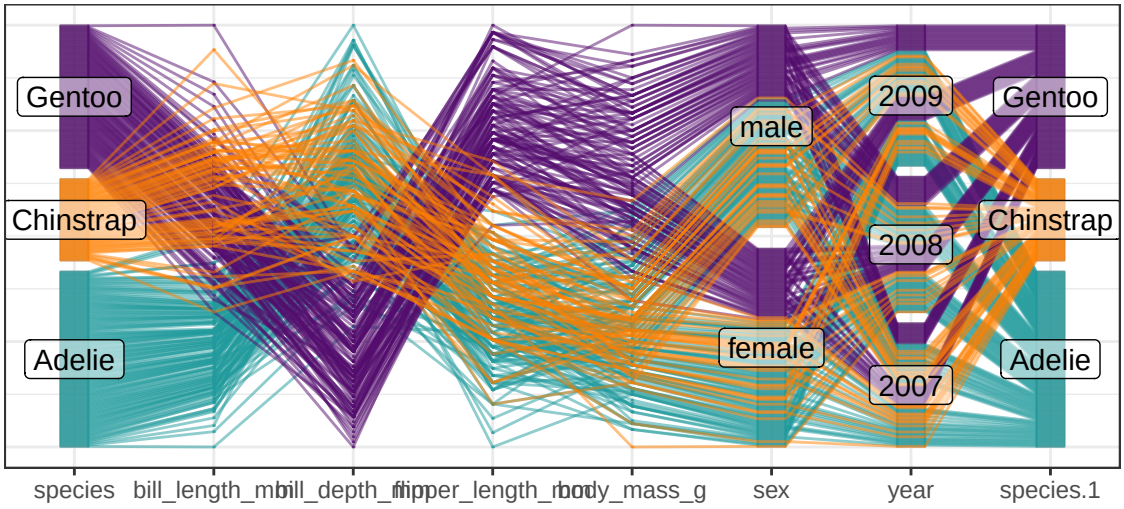
- Less Effective for Dense Ties: When ties are highly frequent, offsets might still result in visual overlap or distortions in high-resolution plots.
- Sequential Dependency: Adjustments depend on the sorted order of the data, which may not be ideal for certain applications.

*Implementation*:

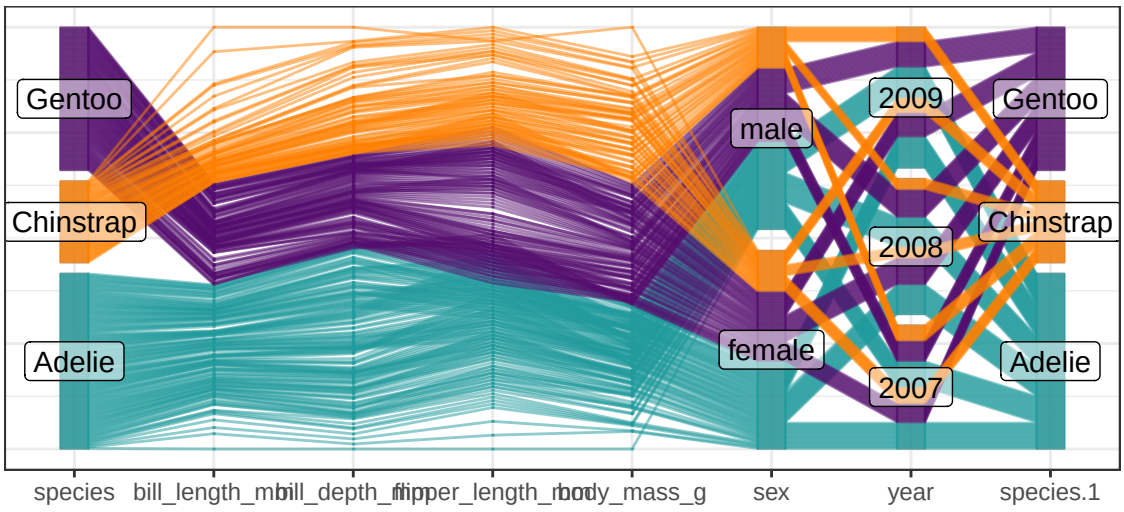
```r
# This function takes a numerical vector and adds a small band around ties
numerical_tie_breaker <- function(values, tie_band = 0.1) {
  # Sort the values
  sorted_values <- sort(values)

  # Initialize the adjusted values
  adjusted_values <- sorted_values

  # Loop through the values to add tie band for ties
  for (i in seq_along(sorted_values)) {
    if (i > 1 && sorted_values[i] == sorted_values[i - 1]) {
      adjusted_values[i] <- adjusted_values[i - 1] + tie_band
    }
  }

  # Return the adjusted values
  return(adjusted_values)
}
```

*PCP Implementation*

Unadjusted PCP of Palmer Penguins




Incremental Tie−Breaking PCP of Palmer Penguins




### Attempt 6: Tie Breaking the Band with auto fraction (Resolving Numerical Ties with Incremental Offset)

*Equation*:

**Step 1 - Detect data type**:

if $\forall x \in \mathbb{Z} \implies$ use a small fixed increment (e.g., $\alpha$).

else $\implies$ continue to outlier check.

**Step 2 - Outlier check ($1.5 \times IQR$ rule):**

$IQR = Q3 - Q1$; $LowerBound = Q1 - 1.5 \times IQR$; $UpperBound = Q3 + 1.5 \times IQR$

$has_outliers = (min(values) < LowerBound) \vee (max(values) > UpperBound)$

**Step 3 - Determine tie band (`tie_band`):**

$$tieband = \begin{cases} \text{if no outliers:} & \alpha \text{ x SD (values)} \\ \text{if outliers:} & \alpha \text{ x IQR (values)} \\ \text{if whole-number:} & \alpha \end{cases}$$

**Step 4 - Increment duplicate sorted values by `tie_band` whenever ties are detected:**

The incremental offset technique for numerical tie-breaking is defined as:

$$v_i' = v_i + \Delta * k$$

Where:

$v_i'$: Adjusted value for the $i$-th data point. $v_i$: Original value for the $i$-th data point. $\Delta$: A small incremental offset (e.g., 0.01). $k$: The relative position of the value within the tie group.

*Advantages*:

- Automated data-type detection: Distinguishes whole-number from continuous data.
- Normality check: Uses Shapiro-Wilk to detect Gaussian vs. non-Gaussian data (if the sample size permits).
- Robust outlier consideration: Employs $1.5 \times IQR$ to switch to a robust measure of spread (IQR) when outliers exist or when data are non-Gaussian.

*Disadvantages*:

- Reliance on Shapiro-Wilk: The test can be sensitive for large samples and is invalid for very small samples (n $< 3$).
- Cutoff choices are somewhat arbitrary: The 0.05 alpha in Shapiro-Wilk and 1.5×IQR threshold are common but not universal.
- Potential instability for very small datasets: Both outlier detection and normality testing can be less reliable with limited data.

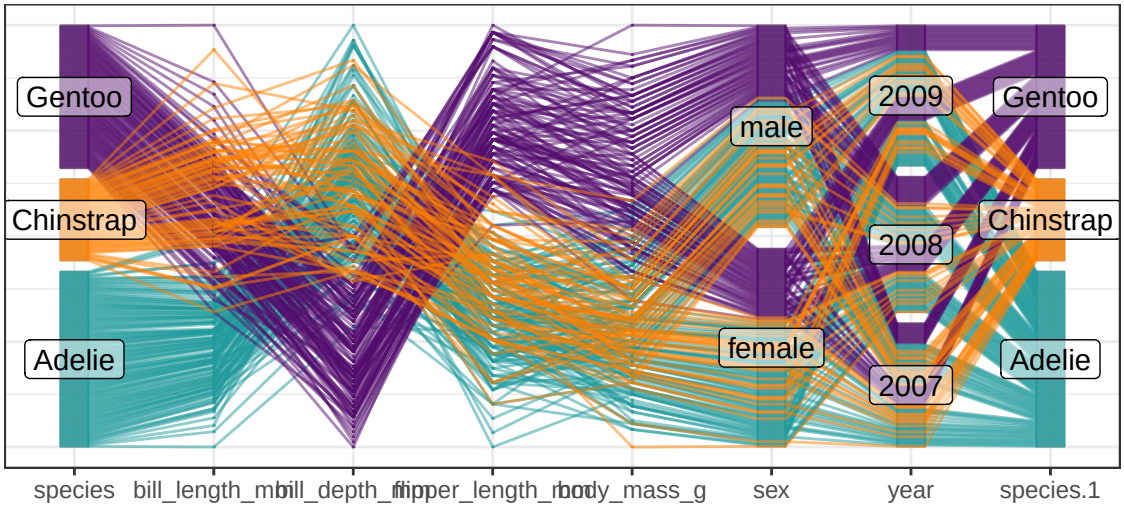
*Implementation*:

```r
auto_fraction <- function(values, scale_factor = 0.1) {
  unique_vals <- unique(sort(values))
  diffs <- diff(unique_vals)

  # Keep only positive gaps
  positive_diffs <- diffs[diffs > 0]

  # If everything is identical or only one unique value, return a small default
  if (length(positive_diffs) == 0) {
    return(scale_factor)
  }

  # A robust choice: median of these positive gaps
  typical_gap <- median(positive_diffs)

  # Return a fraction that is a multiple of the median gap
  scale_factor * typical_gap
}

numerical_tie_breaker <- function(values, base_fraction = 0.1) {
  # 1) Determine if all values are whole numbers
  is_whole_number <- all(abs(values - round(values)) < .Machine$double.eps^0.5)

  # 2) If data are all whole numbers, use a small fraction increment
  if (is_whole_number) {
    tie_band <- base_fraction

  } else {
    # For continuous data, attempt a normality check using Shapiro-Wilk if n >= 3
    # (Shapiro test not valid for n < 3, so skip if too small)
    is_normal <- FALSE
    if (length(values) >= 3) {
      shapiro_p <- shapiro.test(values)$p.value
      is_normal <- (shapiro_p > 0.05)
    }

    # Check for outliers using 1.5 * IQR rule
    q <- stats::quantile(values, probs = c(0.25, 0.75))
    iqr_value <- q[2] - q[1]
    lower_bound <- q[1] - 1.5 * iqr_value
```

```
    upper_bound <- q[2] + 1.5 * iqr_value
    has_outliers <- any(values < lower_bound) || any(values > upper_bound)

    # Decide tie_band logic
    # If data appear normal and have no outliers, use fraction * SD; otherwise use fraction * IQR
    if (is_normal && !has_outliers) {
      tie_band <- base_fraction * stats::sd(values)
    } else {
      tie_band <- base_fraction * iqr_value
    }
  }

  # Optionally, derive the fraction from actual gaps in data (robust approach)
  # fraction <- auto_fraction(values, base_fraction)
  # tie_band <- fraction * <sd or iqr or fixed increment>

  # 3) Sort the values
  sorted_values <- sort(values)

  # 4) Adjust for ties by adding tie_band to each duplicate
  adjusted_values <- sorted_values
  for (i in seq_along(sorted_values)) {
    if (i > 1 && sorted_values[i] == sorted_values[i - 1]) {
      adjusted_values[i] <- adjusted_values[i - 1] + tie_band
    }
  }

  return(adjusted_values)
}
```

*PCP Implementation*

Unadjusted PCP of Palmer Penguins




Auto Fraction with Incremental Tie−Breaking PCP of Palmer Penguins