FORMULA FOR CALCULATING THE FOUR STRETCH RATIOS

I. BASIC PROBLEM AND FORMULA

1.1. The Problem

Given **n colours** with corresponding percentage ratios (total = 100%), find **the 4 stretch ratios** E_1, E_2, E_3, E_4 such that when mapped to **8 bending positions** (A, B, C, D, E, F, G, H), the error between the calculated colour ratio and the target colour ratio is minimised.

1.2. Formula for calculating the ratio from the 4 stretching indices

With the 4 stretch factors E_1, E_2, E_3, E_4, the percentage ratios for the 8 bending positions are calculated using the formula:

Common denominator:

$D = (1/E_1) + (1/E_4) + 2 + (2/E_2) + (1/E_4) + (1/E_3)$

Simplified:

$D = (1/E_1) + (2/E_4) + 2 + (2/E_2) + (1/E_3)$

Position ratios:

Position	Formula	Percentage
A	$(1/E_1) / D$	× 100%
B	$(1/E_4) / D$	× 100%
C	$1 / D$	× 100%
D	$1 / D$	× 100%
E	$(1/E_2) / D$	× 100%
F	$(1/E_2) / D$	× 100%
G	$(1/E_4) / D$	× 100%

H	$(1/E_3) / D$	$\times 100\%$

Properties:

- Positions C and D have the same value
- Positions E and F have the same value
- Positions B and G have the same value
- The sum of the ratios equals 100%

1.3. Constraint conditions

Stretch limit:

$1.1 \leq E_1, E_2, E_3 \leq 4.0$
$1.1 \leq E_4 \leq 6.0$

Relationship between indices:

$E_4 > E_1$ and $E_4 > E_3$
$E_4/E_1 < 4.0$ and $E_4/E_3 < 4.0$

Valid denominator:

$0.5 < D < 10$

Example:

Given $E_1 = 2.0$, $E_2 = 1.5$, $E_3 = 2.5$, $E_4 = 3.0$

Step 1: Calculate the denominator

$D = 1/2.0 + 2/3.0 + 2 + 2/1.5 + 1/2.5 = 0.5 + 0.667 + 2 + 1.333 + 0.4 = 4.9$

Step 2: Calculate the proportion of positions

$A = (1/2.0) / 4.9 \times 100 = 10.20\%$
$B = (1/3.0) / 4.9 \times 100 = 6.80\%$
$C = 1 / 4.9 \times 100 = 20.41\%$
$D = 1 / 4.9 \times 100 = 20.41\%$
$E = (1/1.5) / 4.9 \times 100 = 13.61\%$
$F = (1/1.5) / 4.9 \times 100 = 13.61\%$
$G = (1/3.0) / 4.9 \times 100 = 6.80\%$
$H = (1/2.5) / 4.9 \times 100 = 8.16\%$

Total = 100.00%

II. TARGET FUNCTION AND ERROR

2.1. Error calculation procedure

Step 1: Calculate the ratio for 8 positions

From (E_1, E_2, E_3, E_4), apply the formula to calculate the percentage ratio for A, B, C, D, E, F, G, H.

Step 2: Map colours to positions

For n colours to be matched {$Colour_1$: R_1%, $Colour_2$: R_2%, ..., Colour□: R□%}, map each colour to one or more positions in A-H such that:

|Σ(Percentage of positions for each colour) - R(colour)| ≤ tolerance

With tolerance = 1.5%

Step 3: Calculate the error for each colour

Error(colour i) = |R(actual, i) - R(target, i)|

Where:

- R(actual, i): Total proportion of positions assigned to colour i
- R(target, i): Proportion of colour i in the input

Step 4: Calculate the total error

Total error = Σ Error(colour i) with i = 1..n

2.2. Priority error

For priority colours:

Priority error = Σ Error(colour i) with i ∈ priority colours

The final results are sorted in the following order:

1. Priority error (ascending order)
2. Total error (increasing)

2.3. Acceptance threshold

Only accept results that satisfy:

Total error < 1.5%

2.4. Example of error calculation

For target colour ratio:

- Colour G004: 18%
- Colour G024: 40%
- SW colour: 42%

After calculating with E_1=2.0, E_2=1.5, E_3=2.5, E_4=3.0, assuming the mapping:

- G004 → A + B = 10.20% + 6.80% = 17.00%
- G024 → C + D = 20.41% + 20.41% = 40.82%
- SW → E + F + G + H = 13.61% + 13.61% + 6.80% + 8.16% = 42.18%

Calculate the error:

Error (G004) = |17.00 - 18.00| = 1.00%
Error (G024) = |40.82 - 40.00| = 0.82%
Error (SW) = |42.18 - 42.00| = 0.18%
Total error = 1.00 + 0.82 + 0.18 = 2.00%

Since 2.00% > 1.5%, this result is not acceptable.

III. OVERVIEW OF SEARCH METHODS

3.1. Comparison of methods

Method	Main idea	Time	Number of Results	Comprehensiveness	When to use
Optimisation	Mathematical optimisation with Nelder-Mead	5-15 seconds	50-200	Part	Need to be fast
Grid Search	Full grid scan	20-60 seconds	100-500	High	Full coverage required
All Methods	Optimisation + Grid	30-90 seconds	200-1000 +	Very high	Maximum results

3.2. Search space

Total theoretical combinations:

Number of combinations = $290^3 \times 490 \approx 12$ billion combinations

Where:

- 290 values for E_1, E_2, E_3 (from 1.10 to 4.00, step 0.01)
- 490 values for E_4 (from 1.10 to 6.00, step 0.01)

After filtering constraints $E_4 > E_1$ and $E_4 > E_3$:

Number of remaining combinations ≈ 3 billion combinations

After quick filtering:

Number of combinations to be calculated ≈ 30 million combinations (~1%)

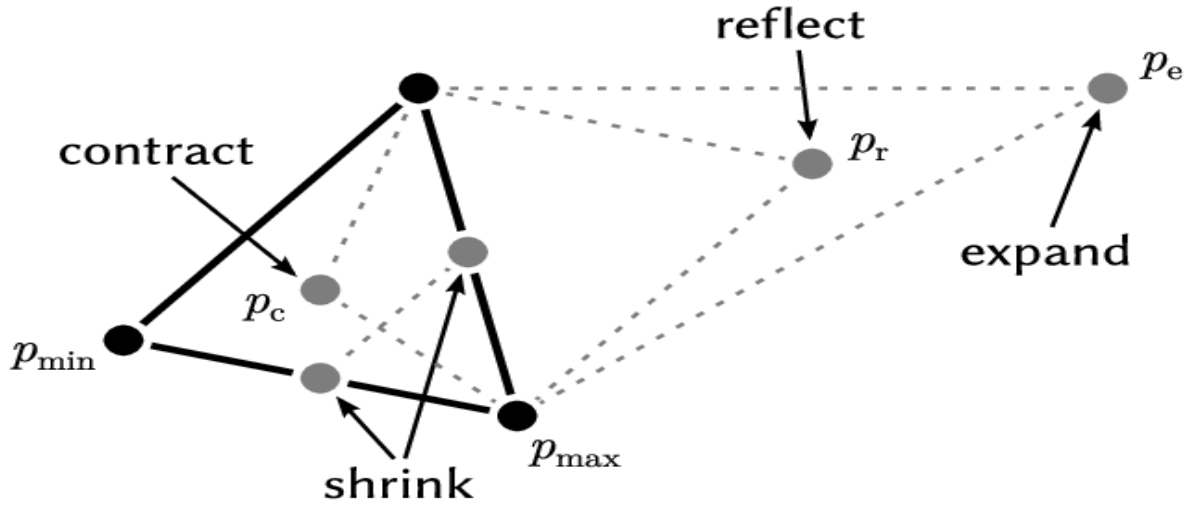
IV. METHOD 1: OPTIMISATION (SCIPY)

4.1. Principle

Use **the Nelder-Mead algorithm** to find the minimum point of the error function in 4-dimensional space (E_1, E_2, E_3, E_4).

Characteristics of the Nelder-Mead algorithm:

- No need to calculate derivatives
- Creates a simplex (polytope) in the search space
- Iteratively performs transformations: reflection, expansion, contraction, and reduction
- Converges quickly in 4-dimensional space



Nelder-Mead algorithm

4.2. Objective function

$f(E_1, E_2, E_3, E_4)$ =
  10000 if a hard constraint is violated
  10000 if colour cannot be mapped
  Total error if valid

Hard constraint:

- $E_1, E_2, E_3 \leq 4.0$
- $E_4 \leq 6.0$
- $E_4 > E_1$ and $E_4 > E_3$
- $E_4/E_1 < 4.0$ and $E_4/E_3 < 4.0$

4.3. Overall Algorithm

ALGORITHM: Optimisation_Search
INPUT:
  - target_ratios: Target colour ratios
  - stretch_bounds: Boundary range (1.1, 6.0)
  - runs: Number of runs = 5000 # number of runs (can be modified)

OUTPUT:
  - List of sets (E_1, E_2, E_3, E_4, error)

STEP 1: Initialise
  cache ← empty dictionary
  results ← empty list
  found_count ← 0

STEP 2: Search loop
  FOR seed = 0 TO 4999 DO:
    // Generate a random starting point
    E_1^0 ← random(1.1, 4.0)
    E_2^0 ← random(1.1, 4.0)
    E_3^0 ← random(1.1, 4.0)
    E_4^0 ← random(1.1, 6.0)
    // Run optimisation
    (E_1*, E_2*, E_3*, E_4*, error*) ← Nelder_Mead(
      x_0 = [E_1^0, E_2^0, E_3^0, E_4^0],
      f = objective_function,
      maxiter = 500,
      bounds = [(1.1, 4.0), (1.1, 4.0), (1.1, 4.0), (1.1, 6.0)]
)
    // Check and save results
    IF error* < 1.5 AND satisfy_constraints(E_1*, E_2*, E_3*, E_4*) THEN:
      results.append((E_1*, E_2*, E_3*, E_4*, error*))
      found_count ← found_count + 1
    // Print progress each time

```
    IF seed > 0 THEN:
      PRINT("Runs completed: ", seed, "/5000 | Found: ", found_count)
STEP 3: Remove duplicates
  unique_results ← []
  seen ← empty set
  FOR EACH result IN sort(results, by=error) DO:
    key ← (result.E₁, result.E₂, result.E₃, result.E₄)
    IF key is NOT IN seen THEN:
      seen.add(key)
      unique_results.append(result)
STEP 4: Return
  RETURN unique_results
```

4.4. Objective function with cache

```
FUNCTION: objective_function(E₁, E₂, E₃, E₄)
STEP 1: Check hard constraints
  IF E₁ > 4.0 OR E₂ > 4.0 OR E₃ > 4.0 THEN:
    RETURN 10000
  IF E₄ > 6.0 THEN:
    RETURN 10000
  IF E₄ ≤ E₁ OR E₄ ≤ E₃ THEN:
    RETURN 10000
  IF E₄/E₁ ≥ 4.0 OR E₄/E₃ ≥ 4.0 THEN:
    RETURN 10000
STEP 2: Rounding and cache check
  E₁ ← round(E₁, 3)
  E₂ ← round(E₂, 3)
  E₃ ← round(E₃, 3)
  E₄ ← round(E₄, 3)
  key ← (E₁, E₂, E₃, E₄)
  IF key is in the cache THEN:
    RETURN cache[key]
STEP 3: Quick-filter
  IF NOT quick_filter(E₁, E₂, E₃, E₄) THEN:
    cache[key] ← 1000
    RETURN 1000
STEP 4: Calculate A-H ratio
  calc_ratios ← calculate_ratios(E₁, E₂, E₃, E₄)
  IF calc_ratios == NULL THEN:
    cache[key] ← 1000
    RETURN 1000
STEP 5: Colour mapping
  match_result ← match_colours(calc_ratios, target_ratios)
  IF match_result is NULL THEN:
    cache[key] ← 1000
    RETURN 1000
```

```
STEP 6: Calculate error
  error ← match_result.total_error
  cache[key] ← error
  RETURN error
```

4.5. Optimisation Techniques

4.5.1. Cache to Avoid Repetitive Calculations

Since the optimisation runs 5000 times with multiple starting points, many combinations (E_1, E_2, E_3, E_4) may be re-evaluated. Caching helps:

- Store previously calculated results
- Reduce computation by 70-80%
- Increase processing speed

4.5.2. Rounding to 3 decimal places

Rounding E_1, E_2, E_3, E_4 to 3 decimal places to:

- Increase the cache hit rate
- Reduce the number of combinations to be checked
- Maintain high accuracy

4.5.3. Randomise multiple starting points

Run 5000 times with random starting points to:

- Avoid getting stuck in local minima
- Explore more regions in the search space
- Increase the likelihood of finding a good solution

4.6. Characteristics

Advantages:

- Fast: Completed in 5–15 seconds
- Accurate: Finds the local optimum region
- Efficiency: Use cache to avoid repetition
- Diversity: Run multiple times with different starting points

Disadvantages:

- Incompleteness: May omit some solutions

When to use:

- When quick results are needed
- Accept 50-200 good solutions
- Prioritise speed over completeness

V. METHOD 2: GRID SEARCH

5.1. Principle

Scan the entire space with a fixed step size (0.01) to find all solutions that satisfy the criteria.

5.2. Search space

Create a grid of values:

$range_{123}$ = {1.10, 1.11, 1.12, ..., 4.00} (290 values)

$range_4$ = {1.10, 1.11, 1.12, ..., 6.00} (490 values)

Total number of initial combinations:

N(total) = $290^3 \times 490 \approx 11{,}947{,}690{,}000$ combinations

After filtering conditions $E_4 > E_1$ and $E_4 > E_3$:

N(filtered) ≈ 3,000,000,000 combinations

5.3. Quick Filter Technique

To reduce the number of calculations, apply filtering conditions before performing full calculations:

FUNCTION: quick_filter(E_1, E_2, E_3, E_4)

// Condition 1: Upper limit

IF $E_1 > 4.0$ OR $E_2 > 4.0$ OR $E_3 > 4.0$ OR $E_4 > 6.0$ THEN:

  RETURN False

// Condition 2: Relationship with E_4

IF $E_4 \leq E_1$ OR $E_4 \leq E_3$ THEN:

  RETURN False

// Condition 3: Ratio E_4

IF $E_4/E_1 \geq 4.0$ OR $E_4/E_3 \geq 4.0$ THEN:

  RETURN False

// Condition 4: Lower limit

IF $E_1 < 1.1$ OR $E_2 < 1.1$ OR $E_3 < 1.1$ OR $E_4 < 1.1$ THEN:

  RETURN False

// Condition 5: Valid denominator

$D \leftarrow 1/E_1 + 2/E_4 + 2 + 2/E_2 + 1/E_3$

IF $D < 0.5$ OR $D > 10$ THEN:

  RETURN False

RETURN True

Effectiveness of Quick Filter:

Stage	Number of combinations	Ratio
Initial	**~12 billion**	**100**
After filtering $E_4 > E_1$, $E_4 > E_3$	**~3 billion**	**25**
After Quick Filter	**~30 million**	**1**

5.4. Sampling Techniques

When the number of combinations after filtering is still too large (> 100,000), use random sampling:

FUNCTION: create_samples(range_123, range_4, max_combinations)

IF total_combinations > max_combinations THEN:

  samples ← []

  FOR i = 1 TO max_combinations DO:

    $E_1 \leftarrow$ random_choice(range_123)

    $E_2 \leftarrow$ random_choice(range_123)

    $E_3 \leftarrow$ random_choice(range_123)

    $E_4 \leftarrow$ random_choice(range_4)

```
    IF E₄ > E₁ AND E₄ > E₃ THEN:
      samples.append((E₁, E₂, E₃, E₄))
    RETURN samples
ELSE:
  // Generate all combinations
  candidates ← []
  FOR E₁ IN range_123 DO:
    FOR E₂ IN range_123 DO:
      FOR E₃ IN range_123 DO:
        FOR E₄ IN range_4 DO:
          IF E₄ > E₁ AND E₄ > E₃ THEN:
            candidates.append((E₁, E₂, E₃, E₄))
  RETURN candidates
```

Notes on Sampling:

- Helps reduce processing time
- May omit some solutions
- Suitable when the search space is too large

5.5. Overall Algorithm

ALGORITHM: Grid_Search

INPUT:

- target_ratios: Target colour ratios

- stretch_range: Stretch range (default: 1.5–6.0)

- max_combinations: Sampling limit = 50,000

OUTPUT:

- List of sets (E₁, E₂, E₃, E₄, error)

STEP 1: Create the value grid

```
  IF stretch_range == NULL THEN:
```

range_123 ← [1.10, 1.11, ..., 4.00] (step 0.01)

range_4 ← [1.10, 1.11, ..., 6.00] (step 0.01)

ELSE:

range_123 ← [value ≤ 4.0 in stretch_range]

range_4 ← [values ≤ 6.0 in stretch_range]

STEP 2: Create combinations

samples ← create_samples(range_123, range_4, max_combinations)

PRINT("Total number of test combinations: ", len(samples))

STEP 3: Scan and calculate

results ← []

checked ← 0

skipped ← 0

calc_cache ← {}

FOR i = 0 TO len(samples)-1 DO:

(E_1, E_2, E_3, E_4) ← samples[i]

// Report progress every 10,000 combinations

IF i % 10000 == 0 AND i > 0 THEN:

PRINT("Processed: ", i, "/", len(samples))

PRINT("Found: ", len(results), " | Skipped: ", skipped)

// Quick filter

IF NOT quick_filter(E_1, E_2, E_3, E_4) THEN:

skipped ← skipped + 1

CONTINUE

checked ← checked + 1

// Check cache

key ← (E_1, E_2, E_3, E_4)

```
    IF key NOT IN calc_cache THEN:
      calc_ratios ← calculate_ratios(E1, E2, E3, E4)
      IF calc_ratios == NULL THEN:
        calc_cache[key] ← NULL
        CONTINUE
      calc_cache[key] ← calc_ratios
    ELSE:
      calc_ratios ← calc_cache[key]
      IF calc_ratios == NULL THEN:
        CONTINUE
    // Colour mapping
    match_result ← match_colours(calc_ratios, target_ratios)
    // Save good result
    IF match_result ≠ NULL AND match_result.total_error < 1.5 THEN:
      results.append((E1, E2, E3, E4, match_result.total_error))
STEP 4: Remove duplicates
  unique_results ← []
  seen ← empty set
  FOR EACH result IN sort(results, by=error) DO:
    key ← (round(result.E1,2), round(result.E2,2),
          round(result.E3,2), round(result.E4,2))
    IF key NOT IN seen THEN:
      seen.add(key)
      unique_results.append(result)
STEP 5: Limit the results
  RETURN unique_results[0:500]  // Take the top 500 best results
```

5.6. Optimisation techniques

5.6.1. Computation Cache

Store the A-H ratio calculation results for each combination (E_1, E_2, E_3, E_4) to:

- Avoid redundant calculations
- Increase processing speed by 3-5 times
- Save CPU resources

5.6.2. Progress reporting

Display information every 10,000 combinations:

- Number of combinations processed
- Number of results found
- Number of combinations skipped
- Hit rate

5.6.3. Smart duplicate removal

Round to two decimal places when removing duplicates to:

- Consider results that are close (difference < 0.01) as identical
- Reduce the number of redundant results
- Retain the top 500 results

5.7. Features

Advantages:

- Comprehensive: Scans the entire space (if not sampling)
- Accuracy: No omissions within the scanned area
- Stable: Consistent results each time it runs
- Optimised: Efficient use of cache

Disadvantages:

- Slow: Takes 20–60 seconds
- Resource-intensive: Requires large memory for cache
- May miss data: If using sampling

When to use:

- When the most comprehensive results possible are required
- Accept longer processing times
- Want stable, repeatable results

VI. METHOD 3: COMBINED APPROACH (ALL METHODS)

6.1. Strategy

Combine two data sources to maximise the quantity and quality of results:

1. Optimisation: Find the optimal region (accurate)
2. Grid Search: Perform a comprehensive (exhaustive) scan

6.2. Eliminating Duplicates

Strategy:

Use the Mapping sequence (sorted) as the key to eliminate duplicates instead of using (E_1, E_2, E_3, E_4).

Reason:

- Multiple sets (E_1, E_2, E_3, E_4) can produce the same sort order.
- Users are interested in the sort order, not the stretch index.
- Reduce redundant results.

Example:

Two different sets of stretch ratios:

- (2.00, 1.50, 2.50, 3.00) → Mapping: "1G004/1G004/1G024+1SW/2SW/1G024/1SW"
- (2.01, 1.49, 2.51, 3.02) → Mapping: "1G004/1G004/1G024+1SW/2SW/1G024/1SW"

Only retain one result as they have the same mapping.

6.4. Handling extended filter conditions

Case 1: No conditions

stretch_bounds ← (1.1, 6.0)

stretch_range ← [1.5, 1.51, ..., 6.0]

Case 2: Upper bound (e.g., ≤ 2.5)

stretch_bounds ← (1.1, 2.5)

stretch_range ← [1.1, 1.11, ..., 2.5]

Case 3: Range [min, max] (e.g., 1.5 to 3.0)

stretch_bounds ← (1.5, 3.0)

stretch_range ← [1.5, 1.51, ..., 3.0]

Case 4: Exact (e.g., exact:1.5,1.3,2.5)

Case 5: Filter by fixed stretch indices E1, E2, E3

Example: E1 = 1.3 or E1 = 1.5, E3 = 3 or E1 = 3, E2 = 2, E3 = 1.3 …

Optimisation:

Grid Search:

stretch_range ← [1.20, 1.21, ..., 1.80] // 1.5 ± 0.3

∪ [1.00, 1.01, ..., 1.60] // 1.3 ± 0.3

∪ [2.20, 2.21, ..., 2.80] // 2.5 ± 0.3

6.5. Features

Advantages:

- Maximum quantity: Combines 3 sources
- High quality: Combines the best from each method
- Versatility: Results from multiple approaches

Disadvantages:

- Slowest: Requires running both methods (30-90 seconds)
- Resource-intensive: High CPU and memory usage
- Potential for duplication: Requires handling of duplicates

When to use:

- When maximising the number of results is required
- Processing time is not critical
- Want comprehensive results from all sources

VII. COLOUR MAPPING ALGORITHM TO POSITION

7.1. The mapping problem

Input:

- Target colour ratios: {Colour$_1$: R_1%, Colour$_2$: R_2%, ..., Colour□: R□%}
- Calculated 8-position ratios: {A: a%, B: b%, C: c%, D: d%, E: e%, F: f%, G: g%, H: h%}
- Tolerance: 1.5%

Output:

- Mapping: Each position A-H is assigned to 1 colour
- Error: Deviation between actual and target ratios for each colour

7.2. Mapping strategy

Principle:

1. Prioritise colours with smaller ratios first, with white placed at positions A and H
2. Allow one colour to occupy multiple positions
3. Allow merging of 2-3 positions if necessary
4. Remaining colours (removed or scattered) will fill empty positions

7.3. Illustrative example

Input:

Target colour ratio (adjusted):

- G004: 24%
- G024: 53%
- SW: 23%

Calculated ratio for 8 positions (E_1=1.8, E_2=1.6, E_3=2.2, E_4=2.8):

A: 11.32%, B: 7.08%, C: 19.81%, D: 19.81%, E: 12.38%, F: 12.38%, G: 7.08%, H: 9.01%

Mapping process:

Step 1: Sort colours

sorted_colours = [(G024, 53%), (G004, 24%), (SW, 23%)]

Step 2: Map SW (23%)

Remaining positions: B, G, H

Total: 7.08 + 7.08 + 9.01 = 23.17%

→ |23.17 - 23.00| = 0.17 ≤ 2.0 ✅

→ B, G, H = SW

Step 3: Map G004 (24%)

Single match test: No positions ≈ 24%

Double match test: A + F = 11.32 + 12.38 = 23.70%

→ |23.70 - 24.00| = 0.30 ≤ 2.0 ✅

→ A, F = G004

Step 2: Mapping G024 (53%)

Single match test: No positions ≈ 53%

Test double match: No pairs ≈ 53%

Triple match test: C + D + E = 19.81 + 19.81 + 12.38 = 52.00%

→ |52.00 - 53.00| = 1.00 ≤ 2.0 ✅

→ C, D, E = G024

Result:

Mapping: 1G004/1SW/2G024/1G024+1G004/1SW/1SW

(A) / (B) / (C+D) / (E+F) / (G) / (H)

Error:

- G004: |23.70 - 24.00| = 0.30%

- G024: |52.00 - 53.00| = 1.00%

- SW: |23.17 - 23.00| = 0.17%

- Total: 1.47% < 1.5% ✅

Reference implementation (source code)

https://drive.google.com/file/d/1OEc6Y76A16YYY6khkaZqStD8r5HGPblT/view?usp=sharing