# INDIVIDUAL MATRIX MODEL

In this fast-paced world, we frequently interact with people from different backgrounds who hold diverse beliefs and morals.

Moreover, personal appearance varies for each individual.

We believe that each person has a unique marker that determines how they present themselves to the outside world.

Furthermore, we believe that in each interaction, we can **quantify** the **individual identity** of each person and the **shared stimulus** between all people.

We suggest that each individual will have a **weight matrix**, which is multiplied by the **universal stimulus matrix** (which is the same for all subjects).

The multiplication of the **weight matrix** and the **stimulus matrix** will result in the **response matrix** of the individual.

Let's model it to make it clearer:

We have our underlying assumptions:

1. i = 1,2...,m for m **subjects**
2. We have our index v for different **voxels** and our index d for different **timeframes**
3. We'll define our matrix $X_i$ with dimensions $v_{rows}$ X $d_{columns}$

Let's write our suggested model:

For the matrices:

$X_i \in \mathbb{R}^{vxd}$, $W_i \in \mathbb{R}^{vxk}$, $S \in \mathbb{R}^{kxd}$ :

Model_1 is:

$$X_i = W_i \cdot S + E_i \quad , \quad \text{for i subject in our sample}$$

Note: to ensure uniqueness of coordinates it is necessary that $W_i$ has linearly independent columns.

Thus, in our model we make a bigger assumption that the weights matrix is **orthogonal**, meaning: $W_i^T W_i = I_k$

Now we're going to optimize our $W_i$ in the following method:

1. Our goal is to **minimize**:

$$Min_{W_i,S} \quad \sum_{i=1}^{m} \|X_i - W_i S\|_F^2$$

1. Select initial $W_i$
2. Set $S = \frac{1}{m} \sum_{i=1}^{m} W_i^T X_i$
3. We have m separate subproblems of the form $\sum_{i=1}^{m} \|X_i - W_i S\|_F^2$

1. Our solution for each subject in each iteration is:

$$W_i = \tilde{U}_i \tilde{V}_i^T \quad \text{where} \quad \tilde{U}_i \tilde{D}_i \tilde{V}_i^T = SVD(X_i S^T)$$

One question immediately comes to mind:

***WHAT'S SVD?***

## SVD

SVD says we can write **ANY** matrix A as $A_{mxn} = U_{mxm}D_{mxn}V^T_{nxn}$ where:

**U** is an **orthogonal matrix** (each of its rows and columns are orthonormal vectors). in general, if A is an m x n matrix, then V is an m x m matrix:

$$U = \begin{bmatrix} | & | & | & | \\ u_1 & u_2 & \dots & u_n \\ | & | & | & | \end{bmatrix}$$

**D** is matrix with the **singular values** on the diagonal and zeros elsewhere. In general if A is an m x n matrix, then D is an m x n matrix. We have in our main diagonal the singular values of A which equal to the square root of the eigenvalues of $A^TA$ or $AA^T$. The rest of the elements are zeros:

$$D = \begin{bmatrix} d_1 & 0 & 0 & 0 & 0 & \dots & 0 \\ 0 & d_2 & 0 & 0 & 0 & \dots & 0 \\ 0 & 0 & \dots & 0 & 0 & \dots & 0 \\ 0 & 0 & 0 & d_n & 0 & \dots & 0 \end{bmatrix}$$

**V** is an **orthogonal** matrix too. In general, if A is an m x n matrix, then V is an n x n matrix.

$$V = \begin{bmatrix} | & | & | & | \\ v_1 & v_2 & \dots & v_n \\ | & | & | & | \end{bmatrix}$$

## Let's visualize it shall we?

In [99]:
```
import numpy as np
import matplotlib.pyplot as plt

def plot_circle(ax, center, radius, **kwargs):
    """Plot a circle with a given center and radius."""
    circle = plt.Circle(center, radius, **kwargs)
    ax.add_artist(circle)

# Generate a circle of points
theta = np.linspace(0, 2 * np.pi, 100)
circle = np.vstack((np.cos(theta), np.sin(theta)))

# Define a matrix to be decomposed
A = np.array([[3, 1], [1, 2]])

# Perform SVD
U, Sigma, Vt = np.linalg.svd(A)
Sigma_matrix = np.diag(Sigma)

# Apply transformations
circle_U = U @ circle
circle_SigmaU = Sigma_matrix @ circle_U
circle_SigmaUVt = Vt @ circle_SigmaU
circle_A = A @ circle

# Plotting
fig, ax = plt.subplots(1, 5, figsize=(25, 5))

# Original circle
ax[0].plot(circle[0, :], circle[1, :], 'b')
plot_circle(ax[0], (0, 0), 1, color='b', fill=False)
ax[0].set_xlim(-4, 4)
ax[0].set_ylim(-4, 4)
ax[0].set_aspect('equal', 'box')
ax[0].set_title("Original Circle")
```

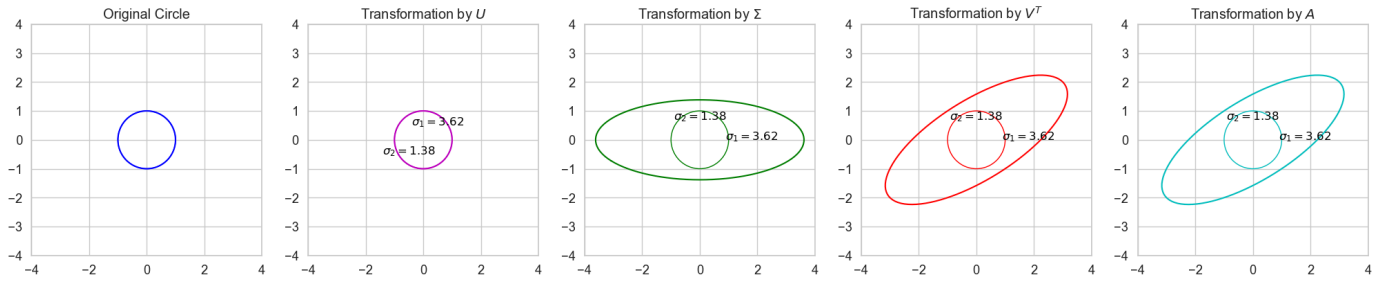
```
# Transformed by U
ax[1].plot(circle_U[0, :], circle_U[1, :], 'm')
plot_circle(ax[1], (0, 0), 1, color='m', fill=False)
ax[1].set_xlim(-4, 4)
ax[1].set_ylim(-4, 4)
ax[1].set_aspect('equal', 'box')
ax[1].set_title("Transformation by $U$")
ax[1].annotate(f"$\\sigma_1 = {Sigma[0]:.2f}$", (0.5, 0.5), color='black', fontsize=12, ha='center')
ax[1].annotate(f"$\\sigma_2 = {Sigma[1]:.2f}$", (-0.5, -0.5), color='black', fontsize=12, ha='center')

# Transformed by Sigma
ax[2].plot(circle_SigmaU[0, :], circle_SigmaU[1, :], 'g')
plot_circle(ax[2], (0, 0), 1, color='g', fill=False)
ax[2].set_xlim(-4, 4)
ax[2].set_ylim(-4, 4)
ax[2].set_aspect('equal', 'box')
ax[2].set_title("Transformation by $\Sigma$")
ax[2].annotate(f"$\\sigma_1 = {Sigma[0]:.2f}$", (Sigma[0]/2, 0), color='black', fontsize=12, ha='center')
ax[2].annotate(f"$\\sigma_2 = {Sigma[1]:.2f}$", (0, Sigma[1]/2), color='black', fontsize=12, ha='center')

# Transformed by Vt
ax[3].plot(circle_SigmaUVt[0, :], circle_SigmaUVt[1, :], 'r')
plot_circle(ax[3], (0, 0), 1, color='r', fill=False)
ax[3].set_xlim(-4, 4)
ax[3].set_ylim(-4, 4)
ax[3].set_aspect('equal', 'box')
ax[3].set_title("Transformation by $V^T$")
ax[3].annotate(f"$\\sigma_1 = {Sigma[0]:.2f}$", (Sigma[0]/2, 0), color='black', fontsize=12, ha='center')
ax[3].annotate(f"$\\sigma_2 = {Sigma[1]:.2f}$", (0, Sigma[1]/2), color='black', fontsize=12, ha='center')

# Transformation by A directly
ax[4].plot(circle_A[0, :], circle_A[1, :], 'c')
plot_circle(ax[4], (0, 0), 1, color='c', fill=False)
ax[4].set_xlim(-4, 4)
ax[4].set_ylim(-4, 4)
ax[4].set_aspect('equal', 'box')
ax[4].set_title("Transformation by $A$")
ax[4].annotate(f"$\\sigma_1 = {Sigma[0]:.2f}$", (Sigma[0]/2, 0), color='black', fontsize=12, ha='center')
ax[4].annotate(f"$\\sigma_2 = {Sigma[1]:.2f}$", (0, Sigma[1]/2), color='black', fontsize=12, ha='center')

plt.show()
```




Now, that's beautiful!

As we can (carefully) see U and $V^T$ rotate the circle and $\Sigma$ stretches it. We get the exact same operation as if we simply multiplied by A!!

# Imports

In [61]:
```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib as plt
import os
```

# Reading all the files from my folders:

In [62]:
```
directory = r"C:\Users\maorb\CSVs"

# Our columns of interest
columns = [
    'AU02_r', 'AU04_r', 'AU05_r', 'AU06_r', 'AU07_r',
    'AU09_r', 'AU10_r', 'AU12_r', 'AU14_r', 'AU15_r', 'AU17_r',
    'AU20_r', 'AU23_r', 'AU25_r', 'AU26_r', 'AU45_r'
```

```
]

# Function to read files and process them
def read_and_process_file(file_path):
    if file_path.endswith('.csv'):
        df = pd.read_csv(file_path)
    elif file_path.endswith('.xlsx'):
        df = pd.read_excel(file_path)
    else:
        return None

    df.columns = df.columns.str.strip() # Strip whitespace from column names
    return df[columns]

# Get all files that start with "Argaman"
files = [f for f in os.listdir(directory) if f.startswith("Argaman") and (f.endswith(".csv") or f.endswith(".xlsx"))]

# Read and process all files
dataframes = [read_and_process_file(os.path.join(directory, file)) for file in files]

# Drop any None values in case some files were not processed
dataframes = [df for df in dataframes if df is not None]

# Find the minimum length of all dataframes
min_length = min(len(df) for df in dataframes)

# Trim all dataframes to the minimum length
dataframes = [df.iloc[:min_length, :] for df in dataframes]

# Combine the dataframes
combined_data = dataframes[0]
for df in dataframes[1:]:
    df = df.add_suffix(f'_{df}')  # Add suffix to each dataframe to avoid column name clashes
    combined_data = combined_data.join(df)
```

In [102…

```
dataframes[2]
```

Out[102]:

| | AU02_r | AU04_r | AU05_r | AU06_r | AU07_r | AU09_r | AU10_r | AU12_r | AU14_r | AU15_r | AU17_r | AU20_r | AU23_r | AU25_r | AU26_r | AU45 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 0.00 | 0.75 | 0.0 | 0.00 | 0.21 | 0.00 | 0.00 | 0.32 | 0.00 | 0.46 | 0.56 | 0.06 | 0.0 | 0.36 | 0.13 | 0. |
| **1** | 0.01 | 0.77 | 0.0 | 0.00 | 0.30 | 0.00 | 0.00 | 0.31 | 0.00 | 0.43 | 0.51 | 0.12 | 0.0 | 0.40 | 0.24 | 0. |
| **2** | 0.03 | 0.76 | 0.0 | 0.00 | 0.30 | 0.00 | 0.00 | 0.31 | 0.00 | 0.41 | 0.49 | 0.13 | 0.0 | 0.43 | 0.33 | 0. |
| **3** | 0.04 | 0.73 | 0.0 | 0.01 | 0.26 | 0.00 | 0.00 | 0.31 | 0.00 | 0.38 | 0.49 | 0.13 | 0.0 | 0.47 | 0.35 | 0. |
| **4** | 0.06 | 0.71 | 0.0 | 0.01 | 0.25 | 0.00 | 0.00 | 0.31 | 0.00 | 0.35 | 0.47 | 0.12 | 0.0 | 0.48 | 0.38 | 0. |
| **…** | … | … | … | … | … | … | … | … | … | … | … | … | … | … | … | |
| **1333** | 0.00 | 0.25 | 0.0 | 1.17 | 1.77 | 0.09 | 0.57 | 1.87 | 0.76 | 0.00 | 0.00 | 0.00 | 0.0 | 0.23 | 0.00 | 0. |
| **1334** | 0.00 | 0.28 | 0.0 | 1.18 | 1.82 | 0.10 | 0.56 | 1.87 | 0.79 | 0.00 | 0.00 | 0.00 | 0.0 | 0.24 | 0.00 | 0. |
| **1335** | 0.00 | 0.27 | 0.0 | 1.19 | 1.83 | 0.09 | 0.53 | 1.89 | 0.83 | 0.00 | 0.00 | 0.00 | 0.0 | 0.24 | 0.00 | 0. |
| **1336** | 0.00 | 0.28 | 0.0 | 1.19 | 1.82 | 0.08 | 0.50 | 1.90 | 0.85 | 0.00 | 0.00 | 0.00 | 0.0 | 0.27 | 0.00 | 0. |
| **1337** | 0.00 | 0.27 | 0.0 | 1.20 | 1.82 | 0.08 | 0.52 | 1.90 | 0.81 | 0.00 | 0.00 | 0.00 | 0.0 | 0.27 | 0.00 | 0. |

1338 rows × 16 columns

Note: In our datasets we have

$$Rows_{timeframes} \quad x \quad Columns_{muscles}, \text{ or } dxv$$

Thus, we're going to transpose the matrix!

In [65]:

```
dataframes_Trans = [df.T for df in dataframes]
```

# Our Identity Matrix model

In [86]:

```
def W_i_calc3(X_i, S):
    X_S_T = np.dot(X_i, S.T)
```

```
    U_i, Sigma, V_i_T = np.linalg.svd(X_S_T, full_matrices=False)
    W_i = np.dot(U_i, V_i_T)
    return W_i

def SRM(X, tol=1e-10, max_iter=100000):
    dist_vec = []
    indices = []
    W_i_vec = []
    W_i_new_vec = []
    n = 16
    m = 8
    W_i_new_group = np.ones((m, n, n))
    W_i = np.ones((n,n)) # Initialize W_i as a matrix of ones
    iter_count = 0
    converged = False
    k = 1
    while not converged and iter_count < max_iter:
            for j, X_i in enumerate(X):
                # Compute S for the current j
                S = (1 / len(X)) * sum(np.dot(W_i.T, X_i) for W_i in W_i_new_group)
                W_i_new_group[j] = W_i_calc3(X_i, S)

                # Calculate distance for convergence check
                dist = np.linalg.norm(X_i - np.dot(W_i_new_group[j], S), 'fro')
                dist_vec.append(dist)
                indices.append(k)
                k += 1

                if dist < tol:
                    converged = True
                    break

            iter_count += 1
            if iter_count >= max_iter:
                converged = True

        # We can find the argmin W_i of the function ||X - W_i @ S||^2 by finding the argmin of ||X - U_i @ D_i @ V_i^T
    A = (np.linalg.norm(X_i - np.dot(W_i[j], S), 'fro'))**2  # We'll find the argmin W_i of this function

    return iter_count, W_i_new_group, S, A, dist_vec, indices
```

In [3]:
```
len(dataframes_Trans[0])
```

Out[3]:
```
16
```

In [87]:
```
iter_count, W_i_new_group, S, A, dist_vec, indices = SRM(dataframes_Trans)
```

Note: We did it with small number of iterations:

In [94]:
```
pd.DataFrame({'Loops entered': iter_count, 'Adjustments': len(indices)}, index=['SRM'])
```

Out[94]:

| | Loops entered | Adjustments |
|---|---|---|
| **SRM** | 8 | 59 |

## W_i for example:

In [95]:
```
W_i_exam = pd.DataFrame(W_i_new_group[0])
W_i_exam
```

Out[95]:

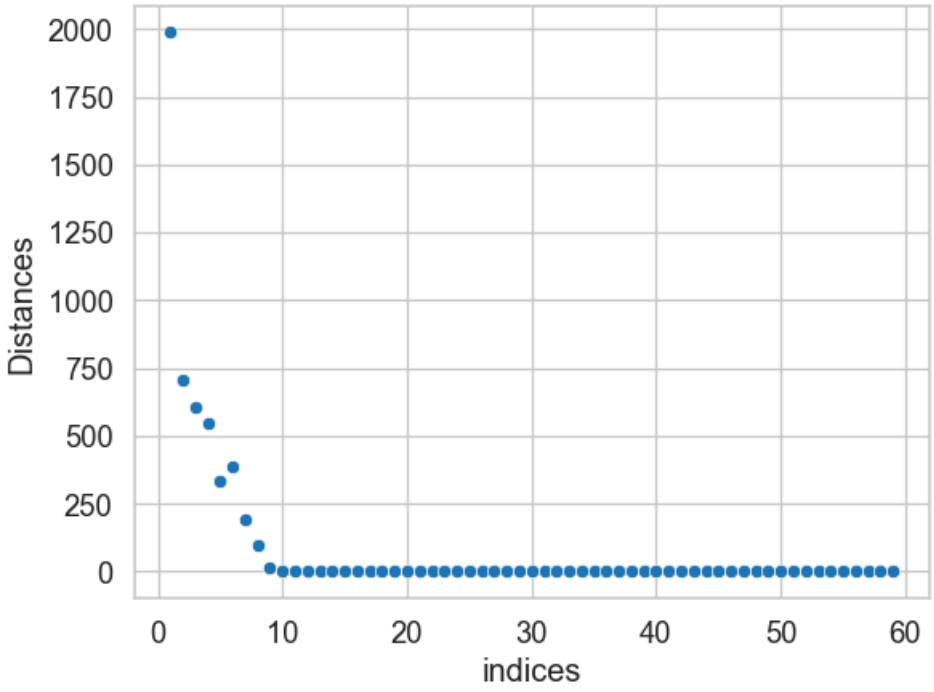
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | -0.909520 | 0.161395 | 0.059021 | -0.037956 | 0.161109 | 0.023087 | 0.105217 | 0.205832 | 0.134212 | 0.124786 | -0.010435 | 0.063397 | 0.019950 |
| **1** | 0.122729 | 0.005368 | -0.027931 | -0.355321 | 0.361675 | -0.043709 | 0.001358 | -0.132437 | -0.248358 | 0.091782 | -0.159589 | 0.648465 | 0.248374 |
| **2** | 0.022640 | 0.004228 | -0.001466 | -0.564851 | 0.201137 | 0.166203 | 0.033360 | -0.173280 | 0.059096 | -0.095786 | 0.623725 | -0.350497 | -0.022896 |
| **3** | 0.040449 | -0.210042 | 0.876351 | 0.188791 | 0.110990 | -0.030440 | 0.082117 | 0.032718 | -0.034516 | -0.024645 | 0.152594 | 0.123795 | -0.080125 |
| **4** | 0.081398 | 0.279939 | 0.034454 | 0.117174 | 0.053497 | 0.924505 | 0.009055 | 0.004904 | -0.029156 | 0.092194 | -0.087741 | 0.091189 | -0.061365 |
| **5** | 0.014285 | 0.007352 | -0.000078 | -0.002628 | 0.017918 | -0.002406 | -0.897419 | 0.357525 | 0.105134 | 0.065379 | 0.123450 | 0.058404 | 0.017587 |
| **6** | 0.251092 | -0.199903 | -0.092763 | -0.050951 | 0.377247 | 0.041929 | 0.280238 | 0.767642 | 0.075435 | -0.016545 | 0.012313 | -0.027563 | -0.115666 |
| **7** | 0.007584 | -0.283345 | -0.291696 | 0.575193 | -0.042950 | 0.082273 | 0.117059 | -0.066171 | 0.118678 | 0.130503 | 0.455151 | 0.159008 | 0.172860 |
| **8** | 0.228575 | 0.836298 | 0.135397 | 0.190983 | 0.045635 | -0.257676 | 0.074342 | 0.080732 | 0.085305 | 0.027741 | 0.214100 | 0.025644 | 0.019526 |
| **9** | 0.042533 | -0.019012 | 0.149126 | -0.329455 | -0.779326 | 0.101276 | 0.154002 | 0.312433 | 0.071640 | 0.056593 | 0.043073 | 0.157563 | 0.193680 |
| **10** | -0.002798 | 0.026095 | 0.066309 | -0.003341 | 0.116161 | -0.007537 | 0.089939 | 0.091914 | 0.123697 | 0.084364 | 0.107222 | -0.121956 | 0.568707 |
| **11** | 0.033167 | 0.012313 | -0.006641 | 0.024243 | 0.058763 | 0.016108 | -0.020577 | 0.008206 | 0.036016 | 0.035957 | -0.008956 | -0.035464 | 0.656028 |
| **12** | 0.044470 | 0.007512 | 0.018894 | 0.020535 | 0.075188 | 0.059756 | 0.009120 | -0.055641 | 0.575921 | -0.648821 | -0.322128 | -0.034021 | 0.128782 |
| **13** | 0.147714 | -0.098510 | 0.103517 | -0.128461 | 0.074946 | -0.020218 | -0.012749 | -0.218188 | 0.657001 | 0.627197 | -0.143027 | 0.003798 | -0.110937 |
| **14** | 0.032532 | 0.138386 | -0.277042 | -0.077220 | -0.063516 | -0.151436 | 0.204394 | 0.120719 | 0.179438 | 0.078969 | 0.070114 | 0.252621 | -0.221277 |
| **15** | 0.074892 | 0.040283 | 0.005220 | 0.043916 | 0.037868 | -0.043382 | 0.060113 | 0.097850 | -0.243726 | 0.321217 | -0.378738 | -0.541228 | 0.118679 |

Indeed, we have 16X16 matrix, row and matrix for each different muscle.

## Let's plot the distances

In [96]:
```
import matplotlib.pyplot as plt
# Create a scatter plot using seaborn
dists = pd.DataFrame({'indices': indices, 'Distances': dist_vec})
sns.scatterplot(dists, x= 'indices', y='Distances', )

# Set the labels and title using seaborn's functionality
sns.set_context("notebook", font_scale=1.2)
sns.set_style("whitegrid")
#sns.label(x = "Indices")
```




with tol=1e-10, max_iter=100000 and 59 adjustments