



Uniwersytet Mikołaja Kopernika

Wydział Fizyki, Astronomii i Informatyki Stosowanej

---

# Wizualizacja danych wielowymiarowych

Praca inżynierska napisana  
pod kierunkiem:

**dr Antoine'a Nauda**

Wykonał:

**Sławomir Ostrowski**

---

Toruń 2008

---

*„Scientia nihil est quam veritatis imago”*

# Spis treści

<b>Streszczenie</b>	<b>5</b>
<b>1 Skalowanie wielowymiarowe</b>	<b>6</b>
1.1 Podział metod . . . . .	6
1.2 Klasyczne skalowanie . . . . .	7
<b>2 Analiza prokrustowa</b>	<b>10</b>
2.1 Ujęcie teoretyczne . . . . .	11
<b>3 Cele pracy i ujęcie praktyczne</b>	<b>13</b>
3.1 Cele pracy . . . . .	13
3.2 Ujęcie praktyczne . . . . .	13
<b>4 Implementacja</b>	<b>21</b>
4.1 Struktura aplikacji . . . . .	21
4.2 Przepływ danych i działanie aplikacji . . . . .	24
4.3 Wykorzystanie zasobów systemowych . . . . .	26
4.4 Interface i obsługa aplikacji . . . . .	28
<b>5 Podsumowanie</b>	<b>33</b>
5.1 Możliwości rozwoju aplikacji . . . . .	33

---

<b>A</b>	<b>Pliki nagłówkowe</b>	<b>35</b>
A.1	Klasa <i>Matrix</i> . . . . .	35
A.2	Klasa <i>DataFile</i> . . . . .	37
A.3	Klasa <i>Graph</i> . . . . .	40
A.4	Klasa <i>CMDS</i> . . . . .	43
A.5	Klasa <i>Procrust</i> . . . . .	44
	<b>Bibliografia</b>	<b>45</b>



# Spis tabel

4.1	Obiekty w programie głównym . . . . .	25
4.2	Przybliżone użycie pamięci . . . . .	27
4.3	Przykładowe czasy obliczeń . . . . .	28
4.4	Funkcje myszy dla obszaru wykresu . . . . .	32

# Spis rysunków

4.1	Zależności między klasami. . . . .	22
4.2	Zależności między obiektami w programie głównym. . . . .	26
4.3	Panel klasycznego skalowania. . . . .	29
4.4	Panel analizy prokrustowej. . . . .	30
4.5	Panel wizualizacji. . . . .	31

# Streszczenie

Praca zawiera opis klasycznej metody skalowania wielowymiarowego oraz analizy prokrustowej. Zagadnienia te zostały przedstawione zarówno w ujęciu teoretycznym jak i w podejściu praktycznym, gdzie następuje przełożenie zgrabnego algorytmu na język komputerowy. Ważnym elementem pracy jest również optymalizacja dokonana na różnych etapach zarówno projektowania jak i samej implementacji.

Uzupełnieniem niniejszej pracy jest zestaw trzech aplikacji spełniających narzucone wymagania analizy wielowymiarowej. Dwa programy konsolowe, wykonujące algorytmy klasycznego skalowania oraz analizy prokrustowej, zostały stworzone aby umożliwić skryptowe przetwarzanie licznych plików z danymi. Aplikacja z interfejsem graficznym umożliwia za to kompleksowe przetwarzanie danych wielowymiarowych, włącznie z końcową wizualizacją danych w postaci dwuwymiarowej reprezentacji graficznej.

# 1. Skalowanie wielowymiarowe

Skalowanie wielowymiarowe (MDS), polega na redukcji wymiarowości zestawów danych. Redukcja nie polega jednak na prostym usunięciu nadmiarowych wymiarów. Wynikiem takiego skalowania powinien być zestaw danych w możliwie najlepszy sposób zachowujący pewne własności danych początkowych, np. wzajemne odległości pomiędzy poszczególnymi obiektami.

Dane wejściowe to zbiór obiektów opisanych zestawem parametrów. Charakter parametrów nie ma właściwie żadnych ograniczeń, mogą to być, np. wielkości charakteryzujące obiekt lub stopień podobieństwa.

Cele takiego skalowania są różnorakie, ale w większości przypadków służy ono analizie statystycznej. Poprzez redukcję wymiarów poszukuje się, np. ukrytych korelacji lub tendencji, trudnych do zauważenia w danych oryginalnych. Traktując niepodobieństwa obiektów jako odległości można poszukiwać niskowymiarowego rozwiązania w celu wizualnej interpretacji danych. W podejściu psychologicznym, można zastosować skalowanie wielowymiarowe jako model wyjaśniający postrzeganie pewnych różnic i podobieństw w odniesieniu do rzeczywistych parametrów.

## 1.1 Podział metod

Pośród mnóstwa metod skalowania wielowymiarowego, właściwie wyróżnia się dwie zasadnicze grupy: skalowanie metryczne oraz skalowanie niemetryczne.

Metody metryczne charakteryzują się tym, że obiekty zbioru są opisane parametrami, na których można przeprowadzać działania matematyczne, np. można

wyznaczyć różnicę parametrów pomiędzy punktami. Do wyznaczenia odległości między obiektami można się wówczas posłużyć różnymi funkcjami transformującymi niepodobieństwa. Szczególnym przypadkiem jest przedstawiona w dalszej części tego rozdziału metoda klasycznego skalowania.

Metody niemetryczne w przeciwieństwie do wspomnianych powyżej działają na zestawach danych opisanych parametrami porządkowymi. Na takich parametrach nie wykonuje się działań matematycznych. Można jedynie określać relacje między nimi, np. który jest większy, ale już bez podania różnicy między nimi. Odległości między obiektami muszą być definiowane na takiej samej zasadzie, tzn. im większe podobieństwo dwóch punktów tym mniejsza odległość między nimi lub im większe niepodobieństwo tym większa odległość. Same odległości nie są jednak interesujące a jedynie relacje między nimi.

## 1.2 Klasyczne skalowanie

Klasyczne skalowanie jest również czasem nazywane analizą osi głównych. Metoda ta została stworzona w 1952 roku przez Warrena S. Torgersona ([1]), i jest to szczególny przypadek metrycznego skalowania wielowymiarowego. W przypadku klasycznego skalowania niepodobieństwa między poszczególnymi obiektami konfiguracji są traktowane jako odległości w przestrzeni Euklidesa, a mogące je jak najlepiej wyjaśnić rozwiązanie poszukiwane jest w niskowymiarowej przestrzeni. Zaletą tej metody jest prosty i przejrzysty algorytm nie wymagający iteracji.

### 1.2.1 Ujęcie teoretyczne<sup>1</sup>

Mając daną macierz źródłową  $Y$  o rozmiarze  $n \times m$ , poszukiwana jest taka macierz  $X$  o rozmiarze  $n \times r$ , gdzie  $r < m$ , która jak najlepiej zachowuje wzajemne odległości między poszczególnymi obiektami macierzy źródłowej.

---

<sup>1</sup>Zarówno notacja jak i cały algorytm zostały zapożyczone z rozdziału 12 w pozycji [2].

Definiując macierz kwadratów odległości dla danych wejściowych

$$\Delta_{ij}^{(2)} = \sum_{l=1}^m (y_{il} - y_{jl})^2, \quad i, j = 1, 2, \dots, n \quad (1.1)$$

oraz dla danych wyjściowych

$$D_{ij}^{(2)} = \sum_{l=1}^r (x_{il} - x_{jl})^2, \quad i, j = 1, 2, \dots, n, \quad (1.2)$$

oczekuje się zatem, że:

$$\Delta^{(2)} = D^{(2)}. \quad (1.3)$$

Wyrażenie 1.2 można zapisać w postaci macierzowej

$$D^{(2)} = \mathbf{c}\mathbf{1}^T + \mathbf{1}\mathbf{c}^T - 2\mathbf{X}\mathbf{X}^T, \quad (1.4)$$

gdzie  $c_i = \sum_{j=1}^r x_{ij}^2$ ,  $i = 1, 2, \dots, n$ . Mnożąc 1.4 obustronnie przez macierz centrującą  $\mathbf{J} = \mathbf{I} - n^{-1}\mathbf{1}\mathbf{1}^T$  i czynnik  $-\frac{1}{2}$  oraz korzystając z 1.3 można zapisać:

$$\begin{aligned} -\frac{1}{2}\mathbf{J}\Delta^{(2)}\mathbf{J} &= -\frac{1}{2}\mathbf{J}(\mathbf{c}\mathbf{1}^T + \mathbf{1}\mathbf{c}^T - 2\mathbf{X}\mathbf{X}^T)\mathbf{J} \\ &= -\frac{1}{2}\mathbf{J}\mathbf{c}\mathbf{1}^T\mathbf{J} - \frac{1}{2}\mathbf{J}\mathbf{1}\mathbf{c}^T\mathbf{J} + \mathbf{J}\mathbf{X}\mathbf{X}^T\mathbf{J}. \end{aligned} \quad (1.5)$$

Mnożenie macierzy przez  $\mathbf{J}$  powoduje scentrowanie jej kolumn wokół zera, zatem  $\mathbf{J}\mathbf{1} = \mathbf{0}$ . Ponadto, ponieważ nie ma to wpływu na odległości między poszczególnymi obiektami, na macierz wynikową można narzucić dodatkowy warunek aby jej kolumny były scentrowane wokół zera. Definiując  $\mathbf{B}_\Delta = \mathbf{X}\mathbf{X}^T$ , równanie 1.5 można zatem uprościć do postaci:

$$-\frac{1}{2}\mathbf{J}\Delta^{(2)}\mathbf{J} = \mathbf{B}_\Delta. \quad (1.6)$$

Aby otrzymać macierz wynikową pozostał już tylko jeden problem do rozwiązania, a mianowicie jak mając  $\mathbf{B}_\Delta$  wyznaczyć  $\mathbf{X}$ . Należy w tym celu skorzystać z teorii zagadnienia własnego. Ponieważ  $\mathbf{B}_\Delta$  jest macierzą symetryczną, można ją przedstawić jako:

$$\mathbf{B}_\Delta = \mathbf{Q}\mathbf{\Lambda}\mathbf{Q}^T, \quad (1.7)$$

gdzie  $\mathbf{\Lambda}$  jest diagonalną macierzą wartości własnych, a  $\mathbf{Q}$  jest ortogonalną macierzą wektorów własnych. Jeśli wszystkie wartości własne są dodatnie wówczas prawdą jest:

$$\mathbf{Q}\mathbf{\Lambda}\mathbf{Q}^T = \left(\mathbf{Q}\mathbf{\Lambda}^{\frac{1}{2}}\right)\left(\mathbf{Q}\mathbf{\Lambda}^{\frac{1}{2}}\right)^T = \mathbf{X}\mathbf{X}^T. \quad (1.8)$$

Przyjmując rozwiązanie jako  $\mathbf{X} = \mathbf{Q}\mathbf{\Lambda}^{\frac{1}{2}}$ , oznaczałoby to przyjęcie  $n$  wymiarowego rozwiązania, co nie koniecznie musi być zredukowaną liczbą wymiarów. Oznaczając przez  $r$  ( $0 < r \leq \text{rz}\mathbf{B}_{\Delta}$ ) wymiar poszukiwanego rozwiązania, można z  $r$  największych wartości własnych skonstruować macierz  $\mathbf{\Lambda}_+$ , a z odpowiadających im wektorów własnych macierz  $\mathbf{Q}_+$ . Wówczas ostateczne rozwiązanie przyjmuje postać:

$$\mathbf{X} = \mathbf{Q}_+ \mathbf{\Lambda}_+^{\frac{1}{2}}. \quad (1.9)$$

Aby błąd był jak najmniejszy wymiar rozwiązania powinien być tak dobrany aby suma wartości własnych w  $\mathbf{\Lambda}_+$  w miarę dokładnie przybliżała sumę wartości własnych w  $\mathbf{\Lambda}$ :

$$\text{Tr } \mathbf{\Lambda}_+ \approx \text{Tr } \mathbf{\Lambda}. \quad (1.10)$$

Funkcja strat, czasem nazywana *Strain*, definiowana jako:

$$\begin{aligned} L(\mathbf{X}) &= \left\| -\frac{1}{2} \mathbf{J} [\mathbf{D}^2 - \mathbf{\Delta}^2] \mathbf{J} \right\|^2 \\ &= \left\| \mathbf{X}\mathbf{X}^T + \frac{1}{2} \mathbf{J}\mathbf{\Delta}^2 \mathbf{J} \right\|^2 \\ &= \left\| \mathbf{X}\mathbf{X}^T - \mathbf{B}_{\Delta} \right\|^2, \end{aligned} \quad (1.11)$$

określa błąd klasycznego skalowania. Można również pokazać, że klasyczne skalowanie minimalizuje funkcję strat.

## 2. Analiza prokrustowa

Prokrust (*Prokrustes*) to pochodzący z mitologii greckiej zbój attycki, syn Posej-dona. Na swoje ofiary czekał przy drodze z Megary do Aten, a schwytanych podróżnych dopasowywał do łoża poprzez obcinanie wystających kończyn lub rozciąganie. Schwytany przez Tezeusza padł ofiarą swojej metody.

Analiza prokrustowa polega na jak najlepszym dopasowaniu zestawu danych wejściowych (*testee*) do zestawu danych docelowych (*target*). Jednakże w przeciwieństwie do swojego imiennika, dopasowanie takie polega na dopasowaniu jeden do jednego, tzn. każdemu elementowi zbioru wejściowego odpowiada jeden element zbioru docelowego, a rozmiar danych wejściowych nie ulega zmianie. Dopasowanie polega na wyznaczeniu zestawu odpowiednich transformacji, a jego celem jest porównanie dwóch konfiguracji. Należy tutaj zaznaczyć, że porównywanie odbywa się w tym przypadku na zasadzie podobieństwa znanego z geometrii. Narzuca to zasadniczy warunek na charakter wyznaczanych transformacji. Możliwe są jedynie takie transformacje, które zachowują kształt konfiguracji, co sprowadza się tutaj do zachowania odległości pomiędzy poszczególnymi punktami. Dozwolone jest zatem skalowanie, rotacja, translacja i odbicie symetryczne. Niedopuszczalne są natomiast, np. skręcenia lub rozciąganie. Analiza prokrustowa pozwala zatem usunąć tylko czysto wizualne różnice wynikające nie z konfiguracji a jedynie położenia danych.

Metoda ta może służyć na przykład do porównywania zestawów danych otrzymanych przez wielowymiarowe skalowanie jednego zestawu danych różnymi metodami.



## 2.1 Ujęcie teoretyczne<sup>1</sup>

Dla dwóch macierzy  $n \times m$ : źródłowej  $\mathbf{Y}$  i celu  $\mathbf{X}$ , poszukiwane są takie transformacje aby:

$$\mathbf{X} \approx s\mathbf{Y}\mathbf{T} + \mathbf{1}\mathbf{t}^T, \quad (2.1)$$

gdzie  $s$  jest czynnikiem skalującym,  $\mathbf{T}$  macierzą transformacji a  $\mathbf{t}$  wektorem przesunięcia. Ze względu na wspomniane wcześniej ograniczenie do transformacji zachowujących kształt konfiguracji, konieczne jest narzucenie warunku na macierz transformacji:

$$\mathbf{T}\mathbf{T}^T = \mathbf{T}^T\mathbf{T} = \mathbf{I}. \quad (2.2)$$

Warunek ten ogranicza  $\mathbf{T}$  do macierzy rotacji i odbicia symetrycznego, a bez niego macierz transformacji mogłaby przyjmować postać dowolnego przekształcenia liniowego, które w ogólności nie musi zachowywać kształtu.

Funkcję strat, która jest definiowana jako suma kwadratów odległości pomiędzy punktami docelowymi a wynikowymi, można zapisać w postaci macierzowej jako:

$$L(s, \mathbf{t}, \mathbf{T}) = \text{Tr} \left[ \mathbf{X} - (s\mathbf{Y}\mathbf{T} + \mathbf{1}\mathbf{t}^T) \right]^T \left[ \mathbf{X} - (s\mathbf{Y}\mathbf{T} + \mathbf{1}\mathbf{t}^T) \right]. \quad (2.3)$$

Przekształcenia powinny być tak wyznaczone aby minimalizować funkcję strat. Zatem optymalny wektor translacji można wyznaczyć przyrównując pochodną cząstkową równania 2.3 ze względu na  $\mathbf{t}$  do zera<sup>2</sup>:

$$\frac{\partial L(s, \mathbf{t}, \mathbf{T})}{\partial \mathbf{t}} = 2n\mathbf{t} - 2\mathbf{X}^T\mathbf{1} + 2s\mathbf{T}^T\mathbf{Y}^T\mathbf{1} = 0, \quad (2.4)$$

$$\mathbf{t} = n^{-1}(\mathbf{X} - s\mathbf{Y}\mathbf{T})^T \mathbf{1}. \quad (2.5)$$

Wstawiając powyższe równanie do funkcji strat otrzymuje się:

$$L(s, \mathbf{T}) = \text{Tr} [\mathbf{J}\mathbf{X} - s\mathbf{J}\mathbf{Y}\mathbf{T}]^T [\mathbf{J}\mathbf{X} - s\mathbf{J}\mathbf{Y}\mathbf{T}], \quad (2.6)$$

gdzie  $\mathbf{J} = \mathbf{I} - n^{-1}\mathbf{1}\mathbf{1}^T$  jest macierzą centrującą.

<sup>1</sup>Zarówno notacja jak i cały algorytm zostały zaczerpnięte z rozdziału 19 w pozycji [2].

<sup>2</sup>Nieocenionym pomocnikiem jest tutaj pozycja [5].

Optymalne  $s$  można wyznaczyć w podobny sposób jak wektor translacji:

$$\frac{\partial L(s, \mathbf{T})}{\partial s} = 2s (\text{Tr } \mathbf{Y}^T \mathbf{J} \mathbf{Y}) - 2 \text{Tr } \mathbf{X}^T \mathbf{J} \mathbf{Y} \mathbf{T} = 0, \quad (2.7)$$

$$s = \frac{\text{Tr } \mathbf{X}^T \mathbf{J} \mathbf{Y} \mathbf{T}}{\text{Tr } \mathbf{Y}^T \mathbf{J} \mathbf{Y}}. \quad (2.8)$$

Uwzględniając powyższe równanie, funkcja strat przyjmuje postać:

$$L(\mathbf{T}) = \text{Tr } \mathbf{X}^T \mathbf{J} \mathbf{X} - \frac{(\text{Tr } \mathbf{X}^T \mathbf{J} \mathbf{Y} \mathbf{T})^2}{\text{Tr } \mathbf{Y}^T \mathbf{J} \mathbf{Y}}. \quad (2.9)$$

Aby wyznaczyć optymalną macierz transformacji należy zminimalizować wyrażenie  $-\text{Tr } \mathbf{X}^T \mathbf{J} \mathbf{Y} \mathbf{T}$  ze względu na  $\mathbf{T}$ . Sytuacja jest bardziej skomplikowana niż w poprzednich przypadkach, ale można w tym celu posłużyć się poniższym algorytmem.

1. Tworzymy macierz  $\mathbf{C}$

$$\mathbf{C} = \mathbf{X}^T \mathbf{J} \mathbf{Y}. \quad (2.10)$$

2. Dokonujemy dekompozycji singularnej (SVD)

$$\mathbf{C} = \mathbf{P} \mathbf{\Phi} \mathbf{Q}^T, \quad (2.11)$$

gdzie  $\mathbf{P}^T \mathbf{P} = \mathbf{Q}^T \mathbf{Q} = \mathbf{I}$ , a  $\mathbf{\Phi}$  jest macierzą diagonalną.

3. Ostatecznie otrzymujemy

$$\mathbf{T} = \mathbf{Q} \mathbf{P}^T. \quad (2.12)$$

Algorytm ten wynika z zastosowania tzw. nierówności Kristofa:

$$-\text{Tr } \mathbf{T} \mathbf{A} \geq -\text{Tr } \mathbf{A}, \quad (2.13)$$

gdzie  $\mathbf{A}$  to macierz diagonalna, a równość zachodzi wyłącznie dla  $\mathbf{T} = \mathbf{I}$ . Korzystając z powyższego twierdzenia i równania 2.11 oraz dokonując cyklicznej permutacji, można zapisać:

$$-\text{Tr } \mathbf{Q}^T \mathbf{T} \mathbf{P} \mathbf{\Phi} \geq -\text{Tr } \mathbf{\Phi}. \quad (2.14)$$

Zatem funkcja strat 2.9 przyjmuje najmniejszą wartość dla  $\mathbf{Q}^T \mathbf{T} \mathbf{P} = \mathbf{I}$ .

## 3. Cele pracy i ujęcie praktyczne

### 3.1 Cele pracy

Celem niniejszej pracy jest stworzenie aplikacji komputerowej, umożliwiającej wstępną, w szczególności wizualną, analizę danych wielowymiarowych. Nałożone wymagania wytyczają trzy odrębne zadania. Aplikacja powinna dokonywać konwersji zestawu danych wielowymiarowych do postaci dwuwymiarowej, dokonywać dopasowania dwóch zestawów danych dwuwymiarowych oraz przedstawiać reprezentację graficzną danych dwuwymiarowych.

### 3.2 Ujęcie praktyczne

Przedstawione w podrozdziałach 1.2.1 i 2.1 ujęcie teoretyczne dobrze wygląda na papierze, jest ścisłe i zwarte. Jednakże praktyczna strona zagadnień informatycznych przeważnie nigdy nie daje się przedstawić równie prosto. Często potrzebne są pewne etapy pośrednie, założenia i uproszczenia. Nie inaczej jest i w tym przypadku. Ogromnie ważnym terminem w tego typu zagadnieniach jest również optymalizacja. Dzisiejsze komputery posiadają coraz to większą moc obliczeniową i dysponują coraz większą pamięcią operacyjną i przestrzenią dyskową. Nie zmienia to jednak faktu, że mimo wszystko aby przeprowadzić obliczenia na coraz większej liczbie danych w rozsądnym czasie, należy korzystać z dostępnych środków świadomie i umiejętnie.

### 3.2.1 Klasyczne skalowanie

Przedstawiony w rozdziale 1.2.1 algorytm klasycznego skalowania składa się z dwóch wymagających numerycznie etapów. Pierwszy etap to wyliczenie i wycentrowanie macierzy kwadratów odległości, a drugi to rozwiązywanie zagadnienia własnego dla wcześniej przygotowanej macierzy.

#### Centrowanie macierzy kwadratów odległości

Centrowanie wierszy i kolumn macierzy wokół zera, sprowadza się do mnożenia trzech macierzy o wymiarach  $n \times n$ , gdzie  $n$  jest liczbą obiektów:

$$\mathbf{B}_\Delta = -\frac{1}{2}\mathbf{J}\Delta^{(2)}\mathbf{J}, \quad (3.1)$$

gdzie macierz centrująca  $\mathbf{J}$  jest tworzona w następujący sposób:

$$\mathbf{J} = \mathbf{I} - n^{-1}\mathbf{1}\mathbf{1}^T. \quad (3.2)$$

Elementy macierzy kwadratów odległości można wyliczać na bieżąco z równania:

$$\Delta_{ij}^{(2)} = \sum_{k=1}^m (y_{ik} - y_{jk})^2, \quad (3.3)$$

gdzie  $\mathbf{Y}$  jest macierzą  $n \times m$  własności obiektów, przy czym na ogół  $n \gg m$ . Wyliczenie pojedynczego elementu scentrowanej macierzy prezentuje się zatem następująco:

$$B_{\Delta_{ij}} = -\frac{1}{2} \sum_{k=1}^n J_{ik} \sum_{l=1}^n J_{lj} \sum_{p=1}^m (y_{kp} - y_{lp})^2. \quad (3.4)$$

Jeśli uwzględnimy, że należy wyliczyć wszystkie  $n^2$  elementów, to nakład wykonanej pracy jest ogromny, a złożoność algorytmu jest rzędu  $O(n^4)$ . Zauważając, że macierz centrująca  $\mathbf{J}$  ma wyłącznie dwie wartości:

$$J_{ij} = \begin{cases} J_{i=j} = 1 - \frac{1}{n} & \text{dla } i = j \\ J_{i \neq j} = -\frac{1}{n} & \text{dla } i \neq j \end{cases}, \quad (3.5)$$

równanie 3.4 można przepisać w postaci:

$$B_{\Delta ij} = -\frac{1}{2} \left\{ \sum_{k=1}^n \sum_{l=1}^n \sum_{p=1}^m (y_{kp} - y_{lp})^2 J_{i \neq j}^2 + \sum_{p=1}^m (y_{ip} - y_{jp})^2 (J_{i=j} - J_{i \neq j})^2 + \right. \\ \left. + \left[ \sum_{k=1}^n \sum_{p=1}^m (y_{ip} - y_{kp})^2 + \sum_{k=1}^n \sum_{p=1}^m (y_{kp} - y_{jp})^2 \right] (J_{i=j} - J_{i \neq j}) J_{i \neq j} \right\} \quad (3.6)$$

Powyższe równanie ma taką samą złożoność, jednak pozwala zaoszczędzić trochę na zapotrzebowaniu na pamięć operacyjną. Jeśli weźmie się pod uwagę fakt, że ilość przetwarzanych danych nie rzadko osiąga  $n \approx 10000$  a nawet więcej, rozmiar jednej macierzy  $n \times n$  o ośmiobajtowych elementach jest rzędu 1GB. Ponadto zauważając, że:

$$J_{i=j} - J_{i \neq j} = 1 \quad (3.7)$$

oraz wyliczając wektor pomocniczy o długości  $n$

$$SD_i = J_{i \neq j} \sum_{k=1}^n \sum_{l=1}^m (y_{kl} - y_{il})^2 \quad (3.8)$$

i wyliczając wartość pomocniczą

$$SSD = J_{i \neq j} \sum_{k=1}^n SD_k, \quad (3.9)$$

wyrażenie na centrowanie macierzy kwadratów odległości można zredukować do znacznie uproszczonej formuły:

$$B_{\Delta ij} = -\frac{1}{2} \left( SSD + SD_i + SD_j + \sum_{k=1}^m (y_{ik} - y_{jk})^2 \right). \quad (3.10)$$

W ostatecznej postaci algorytm centrowania macierzy ma złożoność jedynie rzędu  $O(n^2)$ . Zważywszy, że macierz  $B_{\Delta}$  jest rozmiaru  $n \times n$ , trudno tutaj oczekiwać lepszego rezultatu.

### Zagadnienie własne

Zgodnie z wcześniejszymi informacjami zawartymi w rozdziale 1.2.1, wyznaczenie poszukiwanego rozwiązania jest uwarunkowane rozwiązaniem zagadnienia własnego dla macierzy kwadratów odległości, czyli znalezieniem wartości własnych i odpowiadających im wektorów własnych:

$$B_{\Delta} = Q \Lambda Q^T, \quad (3.11)$$

gdzie  $\mathbf{Q}$  jest macierzą ortogonalną a  $\mathbf{\Lambda}$  to macierz diagonalna.

Oznaczając przez  $r$  wymiar poszukiwanego rozwiązania, z pierwszych  $r$  wartości własnych większych od zera konstruuje się macierz  $\mathbf{\Lambda}_+$  a z odpowiadających im wektorów własnych składana jest macierz  $\mathbf{Q}_+$ . Ostateczne rozwiązanie ma postać:

$$\mathbf{X} = \mathbf{Q}_+ \mathbf{\Lambda}_+^{\frac{1}{2}}. \quad (3.12)$$

Rozwiązanie zagadnienia własnego w ogólności jest żmudne i wymagające ogromnych nakładów obliczeniowych. W tym przypadku sprawa jest trochę uproszczona. Aby otrzymać rozwiązanie o wymiarze  $r$  należy wyznaczyć tylko  $r$  największych wartości własnych i odpowiadających im wektorów własnych. W teorii wymiar rozwiązania powinien być tak dobrany aby była spełniona zależność 1.10. Jednakże w tym przypadku, ponieważ rozwiązanie ma umożliwiać graficzną reprezentację dwuwymiarową, wymiar jest z góry ustalony na  $r = 2$ . A zatem konieczne jest wyznaczenie tylko dwóch największych wartości własnych i dwóch odpowiednich wektorów własnych. Do tego celu wykorzystana została metoda iteracji wektorowych.

### Metoda von Misesa (iteracji wektorowych)<sup>1</sup>

Metoda służy do wyznaczania największej co do modułu wartości własnej i odpowiadającego jej wektora własnego rzeczywistej, symetrycznej macierzy  $\mathbf{A}$ . Oznaczając dominantę wartości własnych przez  $\lambda_1$ , prawdą jest wówczas

$$|\lambda_1| > |\lambda_2| \geq |\lambda_3| \geq \dots \geq |\lambda_n|. \quad (3.13)$$

Dla macierzy  $\mathbf{A}$  istnieje układ  $n$  liniowo niezależnych wektorów własnych  $\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_n$ . Wynika z tego, że:

1. Wektory i wartości własne spełniają zależność

$$\mathbf{A}\mathbf{v}_i = \lambda_i \mathbf{v}_i, \quad i = 1, 2, \dots, n. \quad (3.14)$$

---

<sup>1</sup>Na podstawie [3]

2. Każdy element  $\mathbf{v} \in \mathbb{R}^n$  można przedstawić jako kombinację liniową wektorów własnych  $\mathbf{v}_i$ :

$$\mathbf{v} = c_1 \mathbf{v}_1 + c_2 \mathbf{v}_2 + \cdots + c_n \mathbf{v}_n, \quad c_i = \text{const}; \quad i = 1, 2, \dots, n \quad (3.15)$$

Mnożąc równanie 3.15  $k$  razy przez  $\mathbf{A}$  oraz korzystając z zależności 3.14, otrzymuje się:

$$\begin{aligned} \mathbf{A}^k \mathbf{v} &= c_1 \lambda_1^k \mathbf{v}_1 + c_2 \lambda_2^k \mathbf{v}_2 + \cdots + c_n \lambda_n^k \mathbf{v}_n \\ &= \lambda_1^k \left[ c_1 \mathbf{v}_1 + c_2 \left( \frac{\lambda_2}{\lambda_1} \right)^k \mathbf{v}_2 + \cdots + c_n \left( \frac{\lambda_n}{\lambda_1} \right)^k \mathbf{v}_n \right]. \end{aligned} \quad (3.16)$$

Biorąc pod uwagę zależność 3.13, można zapisać:

$$\frac{\mathbf{A}^k \mathbf{v}}{c_1 \lambda_1^k} \rightarrow \mathbf{v}_1, \quad \text{przy } k \rightarrow \infty, \quad \text{czyli } \mathbf{A}^k \mathbf{v} \approx c_1 \lambda_1^k \mathbf{v}_1. \quad (3.17)$$

Metoda iteracji wektorowych przekłada się na następujący algorytm:

1. Losowy wybór dowolnego wektora początkowego  $\mathbf{v}^{(0)} \in \mathbb{R}^n$
2. Iteracyjne obliczenie  $\mathbf{A}^k \mathbf{v}$ :

$$\mathbf{v}^{(k+1)} = \mathbf{A} \mathbf{v}^{(k)}, \quad k = 0, 1, 2, \dots; \quad \mathbf{v}^{(0)} \text{ dane} \quad (3.18)$$

3. Ostatecznie wynik przyjmuje postać:

$$\mathbf{v}_1 \approx \mathbf{v}^{(k+1)}, \quad \lambda_1 \approx \frac{(\mathbf{v}^{(k)}, \mathbf{v}^{(k+1)})}{(\mathbf{v}^{(k)}, \mathbf{v}^{(k)})} \quad (3.19)$$

Złożoność obliczeniowa powyższego algorytmu jest rzędu  $O(kn^2)$ , gdzie  $k$  jest liczbą iteracji. Dla zestawu danych o  $n \approx 5000$ , empirycznie wyznaczono  $k \approx 100$ . Zbieżność algorytmu jest zależna od wielu czynników i trudno z góry określić niezbędną liczbę iteracji.

Aby obliczyć kolejne wartości i wektory własne można skorzystać z metody nazywanej deflacją. Należy tak zmodyfikować macierz  $\mathbf{A}$  aby przemieścić jedynie wyliczoną wcześniej wartość własną  $\lambda_1$  nie zmieniając pozostałych:

$$\mathbf{A}_1 = \mathbf{A} - \lambda_1 \mathbf{v}_1 \mathbf{v}_1^T. \quad (3.20)$$

Mając tak przesuniętą macierz należy znów wybrać dowolny wektor początkowy  $\mathbf{v}^{(0)} \in \mathbb{R}^n$  i postępować zgodnie z powyższym algorytmem. Z otrzymanych wartości i wektorów własnych konstruowane jest następnie rozwiązanie (3.12).

Należy tutaj jeszcze wspomnieć, że metoda iteracji wektorowych nie gwarantuje znalezienia największej wartości własnej, a jedynie największej co do wartości bezwzględnej. Konieczne zatem może się okazać powtórzenie algorytmu więcej niż tylko zakładana ilość razy.

### Dokładność skalowania

Funkcja strat liczona zgodnie ze wzorem 1.11 wymaga zbyt dużego nakładu pracy, dlatego też błąd klasycznego skalowania jest liczony w trochę zmodyfikowany sposób. Dokładność wyznaczonych transformacji jest wyliczana jako średni kwadrat różnicy kwadratów odległości poszczególnych obiektów. Tak zmodyfikowany sposób oceny błędu, pozwala uniezależnić go od liczby obiektów i porównywać uzyskane dokładności dla zestawów danych o różnej ich liczbie.

### 3.2.2 Analiza prokrustowa

W przypadku analizy prokrustowej, algorytm wyznaczania rozwiązania i poszczególnych transformacji przedstawiony w rozdziale 2.1, w zasadzie nadaje się do praktycznego wykorzystania. Jedynym etapem wymagającym większej uwagi jest operacja dekompozycji singularnej, w skrócie SVD (od angielskiego *Singular Value Decomposition*). Procedura dekompozycji singularnej jest dość złożonym zagadnieniem i dlatego cały zapożyczony algorytm ([4]) został potraktowany jako pewnego rodzaju czarna skrzynka, która wykonuje określone zadanie, bez zagłębianie się zbyt w jej działanie. Rozkład singularny przypomina zagadnienie własne, jednak nie ogranicza się on wyłącznie do macierzy kwadratowych. SVD dla macierzy o wymiarach  $m \times n$  rzędu  $r$ , polega na znalezieniu  $r$  dodatnich wartości singularnych,  $m$  lewych i  $n$  prawych ortonormalnych wektorów singularnych:

$$\mathbf{A} = \mathbf{V} \hat{\mathbf{A}} \mathbf{U}^T. \quad (3.21)$$



Zgodnie z informacjami zawartymi w rozdziale 2.1 algorytm analizy prokrustowej wygląda zatem następująco:

1. Wyliczenie optymalnej macierzy transformacji  $\mathbf{T}$ :

- a. stworzenie macierzy  $\mathbf{C}$

$$\mathbf{C} = \mathbf{X}^T \mathbf{J} \mathbf{Y}, \quad (3.22)$$

- b. dekompozycja SVD

$$\mathbf{C} = \mathbf{P} \mathbf{\Phi} \mathbf{Q}^T, \quad (3.23)$$

- c. macierz transformacji przyjmuje postać

$$\mathbf{T} = \mathbf{Q} \mathbf{P}^T. \quad (3.24)$$

2. Obliczenie czynnika skalującego  $s$

$$s = \frac{\text{Tr } \mathbf{X}^T \mathbf{J} \mathbf{Y} \mathbf{T}}{\text{Tr } \mathbf{Y}^T \mathbf{J} \mathbf{Y}}. \quad (3.25)$$

3. Wyznaczenie wektora przesunięcia  $\mathbf{t}$

$$\mathbf{t} = n^{-1} (\mathbf{X} - s \mathbf{Y} \mathbf{T})^T \mathbf{1}. \quad (3.26)$$

4. Wyznaczenie poszukiwanego rozwiązania  $\mathbf{Z}$

$$\mathbf{Z} = s \mathbf{Y} \mathbf{T} + \mathbf{1} \mathbf{t}^T. \quad (3.27)$$

Na etapie implementacji możliwa jest jeszcze pewna oszczędność na zapotrzebowaniu pamięci. Występująca w równaniach 3.22 i 3.25 macierz centrująca  $\mathbf{J}$  jest rozmiaru  $n \times n$ , jednak zgodnie z równaniem 3.5 ma ona tylko dwie różne wartości. Rozsądnym wydaje się więc napisanie prostej funkcji imitującej tę macierz. Dla  $n \approx 10000$ , oszczędność jest rzędu 1GB. Dodatkowo oszczędza się również na czasie samego tworzenia tej macierzy. Taka optymalizacja nie ma jednak żadnego wpływu na sam algorytm.

### **Dokładność transformacji**

Dokładność analizy prokrustowej jest wyliczana jako średni kwadrat różnicy odległości pomiędzy punktami docelowymi i punktami uzyskanymi z transformacji punktów źródłowych. Takie obliczenia można przedstawić w zapisie macierzowym jako:

$$\Delta_{PA} = n^{-1} \text{Tr} (\mathbf{Z} - \mathbf{Y})^T (\mathbf{Z} - \mathbf{Y}). \quad (3.28)$$

Zgodnie z równaniem 2.3 jest to funkcja strat dzielona przez liczbę punktów. Tak jak w przypadku klasycznego skalowania, modyfikacja ta pozwala uniezależnić błąd od liczby punktów i umożliwia porównywanie jakości dopasowania dla zestawów danych o różnej liczbie obiektów.

## 4. Implementacja

Aplikacja została napisana w języku C++, przy użyciu oprogramowania *Borland C++ Builder 6* dla środowiska *Microsoft Windows*.

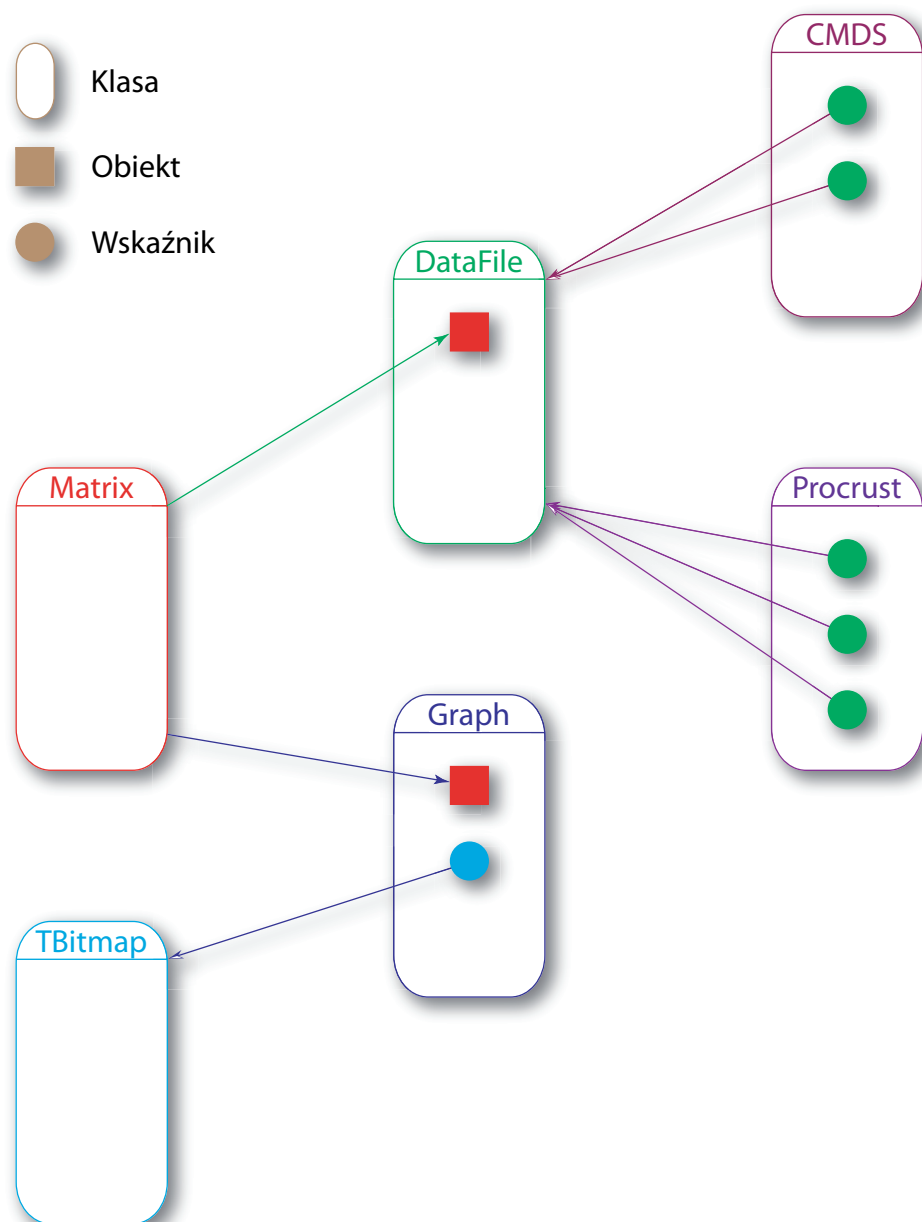
Założenia projektu nakładają wymagania zaimplementowania trzech, w zasadzie odrębnych zagadnień: klasycznego skalowania, analizy prokrustowej i wizualizacji. Można do tego podejść na kilka różnych sposobów. Naturalnym rozwiązaniem byłoby stworzenie trzech odrębnych aplikacji, każdej odpowiedzialnej za swoje zadanie. W myśl tej idei powstały dwa odrębne programy, dokonyujące klasycznego skalowania i analizy prokrustowej, wywoływane z wiersza poleceń. Nie posiadają one wizualizacji, ale za to świetnie nadają się do skryptowego przetwarzania dużej liczby plików. Trzeci program został stworzony z odmienną koncepcją. Została stworzona jedna aplikacja wizualna, wykonująca wszystkie czynności. Dzięki takiemu podejściu możliwe stało się stworzenie odpowiedniego przepływu informacji między poszczególnymi jej częściami, w wyniku czego, powstała aplikacja sprawia wrażenie jedności i integralności całego projektu.

Dla uproszczenia, w dalszej części tego rozdziału, struktury i działanie programu zostały przedstawione wyłącznie w oparciu o aplikację wizualną. Stanowi ona uogólnienie pozostałych aplikacji konsolowych, które są niejako wycinkami aplikacji głównej.

### 4.1 Struktura aplikacji

Stworzona aplikacja opiera się na trzech klasach będących reprezentacją danych, dwóch klasach o charakterze obliczeniowym oraz zestawie funkcji po-

mocniczych. Trzy podstawowe klasy reprezentują: pliki wejścia i wyjścia (klasa *DataFile*), zestawy danych (klasa *Matrix*) oraz reprezentację graficzną danych (klasa *Graph*). Pozostałe dwie klasy *CMDS* oraz *Procrust* są odpowiedzialne za wykonywanie algorytmów odpowiednio klasycznego skalowania i analizy prokrustowej. Powiązania pomiędzy poszczególnymi klasami zostały zobrazowane na rysunku 4.1.



Rysunek 4.1. Zależności między klasami.

### 4.1.1 Klasa *Matrix*

Podstawą implementacji całego projektu jest klasa rzeczywistej macierzy dwuwymiarowej *Matrix* wraz z działającymi na niej funkcjami pomocniczymi. Począwszy od przechowywania danych przez dokonywanie transformacji i tworzenie reprezentacji graficznej a kończąc na obsłudze plików, wszystkie te zadania opierają się na tej podstawowej klasie.

Klasa ta została napisana z myślą o ułatwieniu przeprowadzania obliczeń na macierzach. Głównym składnikiem klasy jest dynamicznie tworzona dwuwymiarowa macierz o ośmiobajtowych elementach rzeczywistych. Pozostałe funkcje składowe wraz z pomocniczymi funkcjami działającymi na obiektach tej klasy, mają za zadanie stworzyć intuicyjny interfejs umożliwiający proste przeprowadzanie podstawowych obliczeń w zapisie macierzowym.

### 4.1.2 Pliki wejścia i wyjścia

Struktura plików wejściowych i wyjściowych składa się z kilku sekcji i taka też jest ich reprezentacja w programie. Za obsługę plików odpowiedzialna jest klasa *DataFile*. Pliki są opisane przez dwie główne struktury: obiekt klasy *Matrix* reprezentujący zastaw danych oraz dynamiczne listy połączone zawierające opis i parametry danych. Dodatkowa struktura będąca wektorem danych tekstowych jest odpowiedzialna za przechowywanie opisu dla każdego wiersza danych (obiektu). Funkcje składowe umożliwiają odpowiedni odczyt i zapis plików oraz manipulację parametrami i opisem danych.

### 4.1.3 Klasy obliczeniowe

Klasy o charakterze obliczeniowym są niewątpliwie istotą tego projektu. Ich głównym zadaniem jest wykonywanie algorytmów klasycznego skalowania i analizy prokrustowej, opisanych w rozdziałach 1.2.1 i 2.1. Podczas inicjalizacji obiektów tych klas, są im przypisywane obiekty klasy *DataFile*, reprezentujące

odpowiednie pliki. Klasa *CMDS* zawiera wskaźniki do pliku z danymi wielowymiarowymi i pliku wyjściowego. Natomiast klasa *Procrust* posiada wskaźniki do plików z danymi wejściowymi i docelowymi oraz do pliku wyjściowego.

#### 4.1.4 Wizualizacja

Za reprezentację graficzną zestawu danych w programie odpowiedzialna jest klasa *Graph*. Sama klasa nie zawiera faktycznego wykresu a jedynie dane i metody niezbędne do jego stworzenia. Podczas procesu inicjalizacji obiektu tej klasy następuje powiązanie go z zewnętrznym komponentem graficznym, na którym ma znajdować się wykres. Oprócz powiązanego komponentu graficznego istotnym jej składnikiem jest obiekt klasy *Matrix* służący do przechowywania danych oraz wektor danych tekstowych z etykietkami wszystkich punktów. Zestaw funkcji składowych dostarcza interfejsu umożliwiającego zmianę parametrów i personalizację wykresu. Klasa dostarcza również metodę do zapisu wykresu w postaci pliku graficznego.

Dodatkową funkcją, zaimplementowaną już poza klasą, jest buforowanie. Buforowanie polega na tworzeniu wykresu na niewidocznym obiekcie graficznym i dopiero po zakończeniu całego procesu rysowania, obszar wykresu kopiowany jest na właściwy komponent. Zadaniem tej funkcji jest zniwelowanie niepożądanego efektu migotania obrazu podczas odświeżania zawartości wykresu.

## 4.2 Przepływ danych i działanie aplikacji

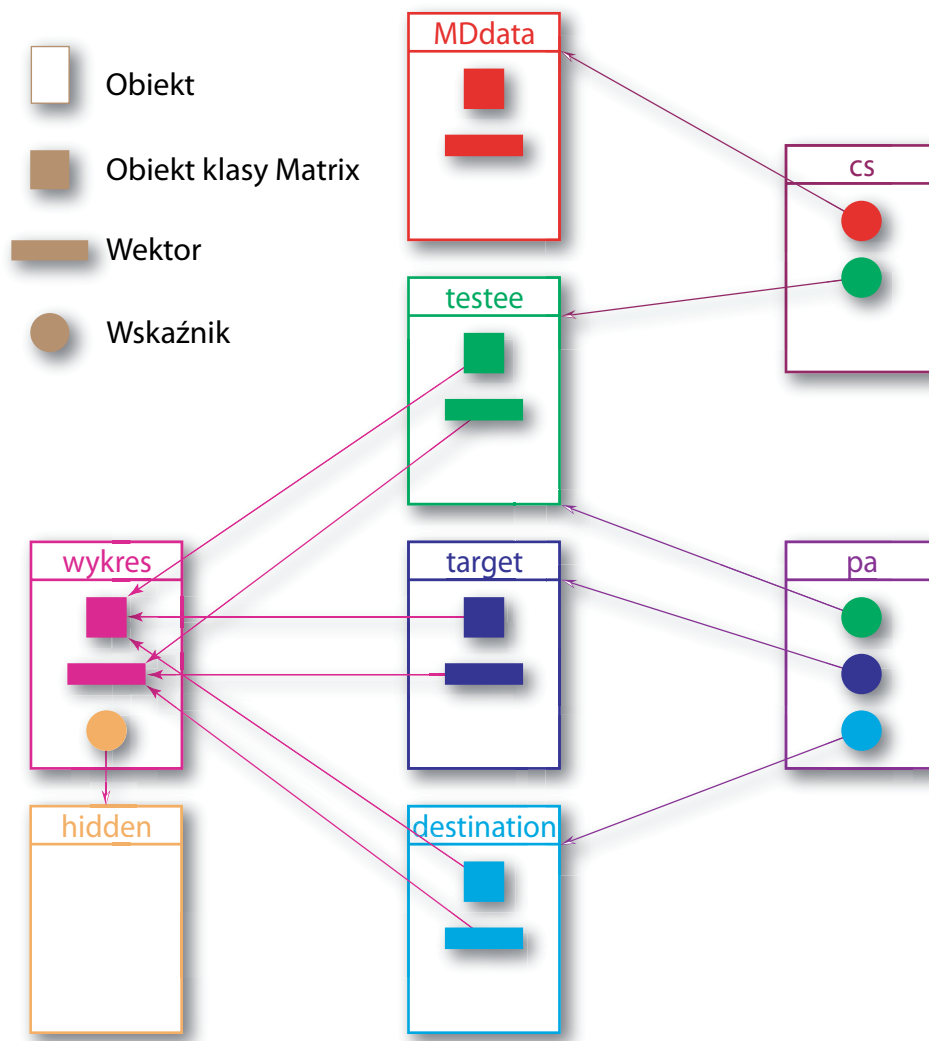
Gdy struktura danych i funkcje są odpowiednio dobrane do zagadnienia, wówczas implementacja całości projektu przypomina już tylko układanie klocków. Trzeba jeszcze tylko wiedzieć co ułożyć i w jakiej kolejności. Program główny korzysta z kilku zasadniczych zmiennych zebranych w tabeli 4.1. Zależności i przepływ danych pomiędzy poszczególnymi obiektami zostały zobrazowane na rysunku 4.2 na stronie 26.

Tabela 4.1. Obiekty w programie głównym

Zmienna	Typ	Opis
MDdata	DataFile	reprezentacja pliku wejściowego z danymi wielowymiarowymi
testee	DataFile	reprezentacja pliku wejściowego/wyjściowego z danymi dwuwymiarowymi
target	DataFile	reprezentacja pliku z dwuwymiarowymi danymi docelowymi
destination	DataFile	reprezentacja pliku wyjściowego z danymi dwuwymiarowymi
cs	CMDS	obiekt odpowiedzialny za cały proces klasycznego skalowania
pa	Procrust	obiekt odpowiedzialny za cały proces analizy prokrustowej
hidden	TBitmap	obiekt przechowujący graficzną postać wizualizacji
wykres	Graph	reprezentacja graficzna danych dwuwymiarowych

Program spełnia trzy funkcje i nic nie stoi na przeszkodzie aby wykorzystywać go tylko do jednej z nich. Ale można go również wykorzystać do wszystkich tych zadań jednocześnie, oczywiście sekwencyjnie. Przebieg takiego działania aplikacji wyglądałby w następujący sposób:

1. Inicjalizacja obiektów:
  - a. zainicjowanie obiektu cs zmiennymi MDdata i testee,
  - b. zainicjowanie obiektu pa zmiennymi testee, target i destination,
  - c. zainicjowanie obiektu wykres zmienną hidden.
2. Wczytanie pliku z danymi wielowymiarowymi (MDdata).
3. Wykonanie klasycznego skalowania:
  - a. wynik jest zapisywany jako testee, który jest jednocześnie obiektem wejściowym dla analizy prokrustowej,
  - b. testee zostaje przesłany do wykresu.
4. Wczytanie pliku docelowego dla analizy prokrustowej (target),
  - a. target jest automatycznie uwzględniany na wykresie.
5. Wykonanie analizy prokrustowej:



Rysunek 4.2. Zależności między obiektami w programie głównym.

- a. wynik jest zapisywany jako destination,
  - b. destination jest automatycznie przesyłany do wykresu.
6. Analiza wizualna danych na wykresie.

### 4.3 Wykorzystanie zasobów systemowych

Odpowiednie wykorzystanie zasobów systemowych jest niezwykle ważne, gdyż zarówno pamięć operacyjna jak i moc obliczeniowa są ograniczone. Często zdarza się tak, że optymalizując czas obliczeń, wykorzystanie pamięci znacząco



wzrasta lub odwrotnie. Należy zatem znaleźć odpowiednią równowagę pomiędzy czasem obliczeń a zużyciem pamięci. O ile czas właściwie nie jest ograniczony to pożądanym jest aby był jak najkrótszy. Natomiast ilość pamięci operacyjnej od razu narzuca surowe ograniczenie. W systemie Microsoft Windows, jeśli nie ma wystarczającej ilości pamięci operacyjnej, wykorzystywana przez program pamięć może być automatycznie mapowana na przestrzeń dyskową, co właściwie usuwa wcześniejsze ograniczenie. Powoduje to jednak ogromne spowolnienie procesu obliczeń i jeśli jest to możliwe, należy tego unikać. Jednakże, jeśli nie ma innego wyjścia, dobrze jest mieć świadomość, że taki mechanizm istnieje.

Ponieważ zadaniem aplikacji jest przetwarzanie licznych zbiorów danych, postanowiono zminimalizować zużycie pamięci, oczywiście w granicach rozsądku. Przybliżone zużycie pamięci dla poszczególnych zadań aplikacji w zależności od rozmiaru danych wejściowych zebrano w tabeli 4.2. Należy tutaj zaznaczyć, że do wielkości zużycia pamięci z tabelki, należy doliczyć około 25MB na działanie samego interfejsu. Dla największego testowanego zestawu danych o wymiarach  $4435 \times 36$  maksymalne zużycie pamięci jest poniżej 200MB.

Tabela 4.2. Przybliżone użycie pamięci

Zadanie	Rozmiar danych	Użycie pamięci* [B]	
		Struktury danych <sup>†</sup>	Etap obliczeń <sup>‡</sup>
Klasyczne skalowanie	$n \times m$	$8(m+2)n$	$8(n+4)n$
Analiza prokrustowa	$n \times 2$	$48n$	$80n$
Wizualizacja	$n \times 2$	$159n$	—

\*Maksymalne wykorzystanie pamięci;

<sup>†</sup>Struktury istniejące po zakończeniu danego etapu i przy założeniu średniej długości etykiety 5 znaków;

<sup>‡</sup>Struktury istniejące tymczasowo w trakcie obliczeń;

Czas wykonywania obliczeń jest w dużej mierze uzależniony od mocy obli-

zeniowej komputera, dlatego też w tabeli 4.3 przedstawiono czasy dla dwóch różnych zestawów komputerowych.

Tabela 4.3. Przykładowe czasy obliczeń

Etap obliczeń	Rozmiar danych	Czasy wykonywania obliczeń* [s]	
		Zestaw testowy A <sup>†</sup>	Zestaw testowy B <sup>‡</sup>
Klasyczne skalowanie <sup>i</sup>	150 × 4	0.04	0.08
Klasyczne skalowanie <sup>f</sup>	150 × 4	0.02	0.03
Klasyczne skalowanie <sup>i</sup>	1869 × 7	5.03	11.51
Klasyczne skalowanie <sup>f</sup>	1869 × 7	4.51	10.62
Klasyczne skalowanie <sup>i</sup>	4435 × 36	37.12	92.29
Klasyczne skalowanie <sup>f</sup>	4435 × 36	36.25	90.23
Analiza prokrustowa	150 × 2	0.01	0.02
Analiza prokrustowa	1869 × 2	0.92	0.92
Analiza prokrustowa	4435 × 2	5.08	5.09

\*Średnia z trzech powtórzeń

<sup>†</sup>Sony VAIO; Intel Core 2 Duo CPU T7700@2.4GHz, 2.00GB RAM; Windows Vista;

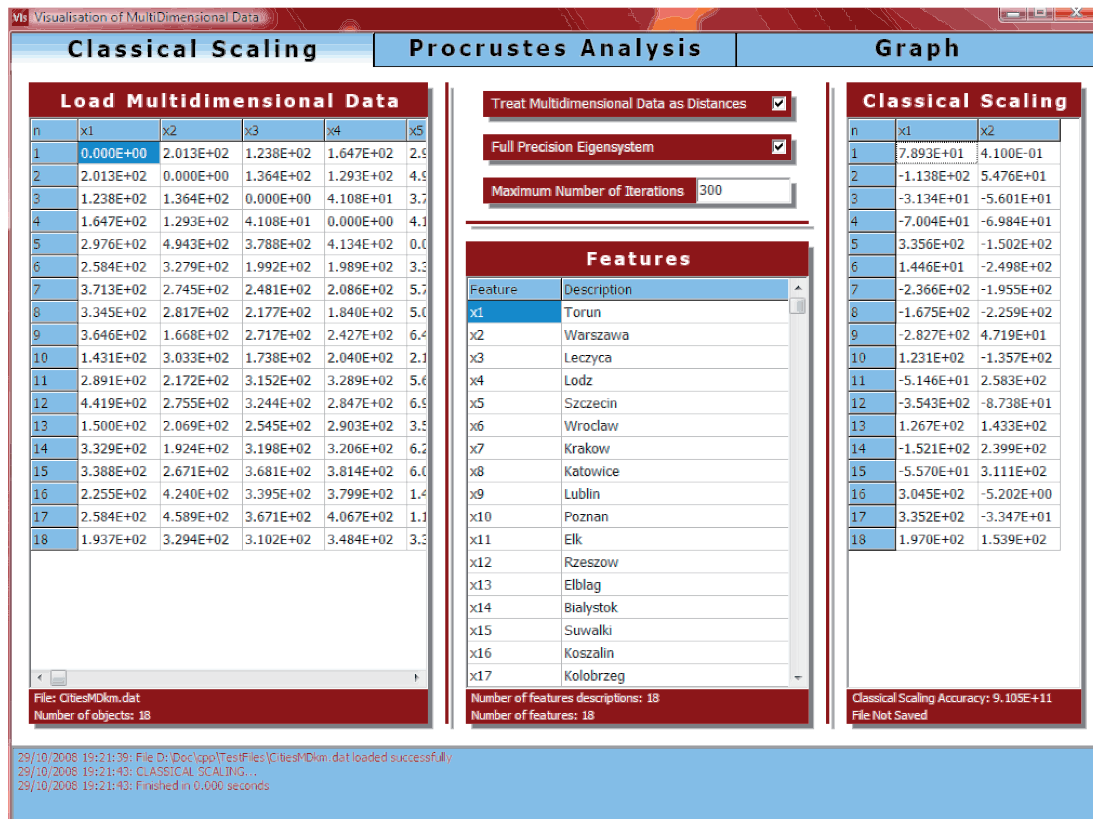
<sup>‡</sup>AthlonXP 2600+@2.1GHz, 512MB RAM; Windows XP;

<sup>i</sup>Pełna dokładność zagadnienia własnego – wyłączona; Liczba iteracji – 100;

<sup>f</sup>Pełna dokładność zagadnienia własnego – wyłączona; Maksymalna liczba iteracji – 300;

## 4.4 Interface i obsługa aplikacji

Interface aplikacji składa się z trzech głównych części. W górnej części znajduje się grupa trzech przycisków odpowiedzialna za wybór aktualnego zagadnienia: klasycznego skalowania (*Classical Scaling*), analizy prokrustowej (*Procrustes Analysis*) lub wizualizacji (*Graph*). Tuż poniżej znajduje się panel, którego zawartość jest uzależniona od aktualnie wybranej funkcji aplikacji. Na samym dole zostało umieszczone okno komunikatów, w którym pojawiają się różne informacje dokumentujące działanie aplikacji.



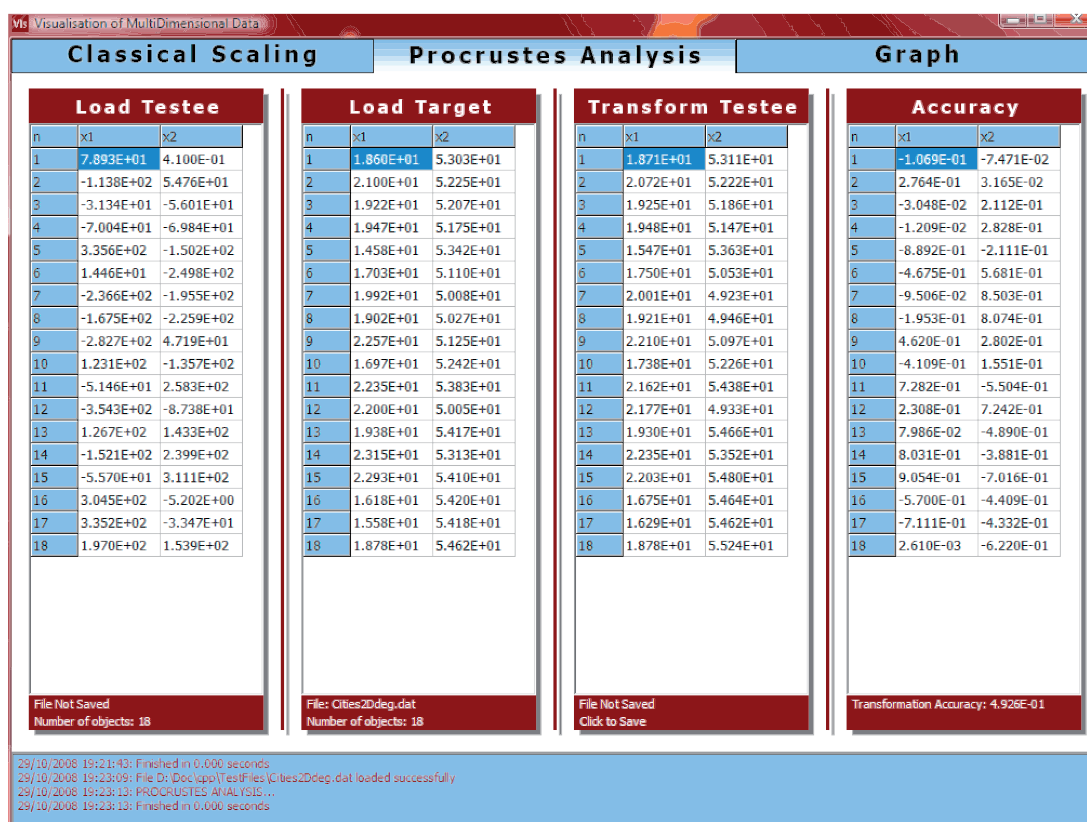
Rysunek 4.3. Panel klasycznego skalowania. Dane wejściowe to odległości między miastami.

#### 4.4.1 Panel klasycznego skalowania

Przedstawiony na rysunku 4.3 panel klasycznego skalowania, jak nazwa sugeruje, służy do przeprowadzania operacji klasycznego skalowania. Wczytywanie danych wielowymiarowych odbywa się poprzez załadowanie odpowiedniego pliku, naciskając przycisk *Load Multidimensional Data*. Dane są wyświetlane poniżej, natomiast nazwy dla poszczególnych wymiarów są wylistowane pod nagłówkiem *Features*. Sam proces klasycznego skalowania inicjuje się wciskając przycisk *Classical Scaling*. Po zakończeniu obliczeń dane są wypisywane w odpowiedniej tabeli. Jeśli pod tabelą z danymi znajduje się napis informujący, że plik nie jest zapisany (*File Not Saved*), klikając go można zachować plik pod wybraną nazwą.

W centralnej części panelu znajdują się opcje mające wpływ na dokładność przeprowadzanych obliczeń na etapie rozwiązywania zagadnienia własnego, opianego w rozdziale 3.2.1 na stronie 16. Jeśli opcja pełnej dokładności (*Full Precision Eigensystem*) nie jest zaznaczona, wówczas należy podać liczbę iteracji,

Zaznaczenie opcji *Treat Multidimensional Data as Distances* powoduje, że dane wejściowe są traktowane jako macierz odległości między obiektami. W przeciwnym razie dane są traktowane jako współrzędne w przestrzeni wielowymiarowej. Opcja ta jest dostępna wyłącznie po załadowaniu danych, dla których liczba obiektów równa jest liczbie wymiarów.



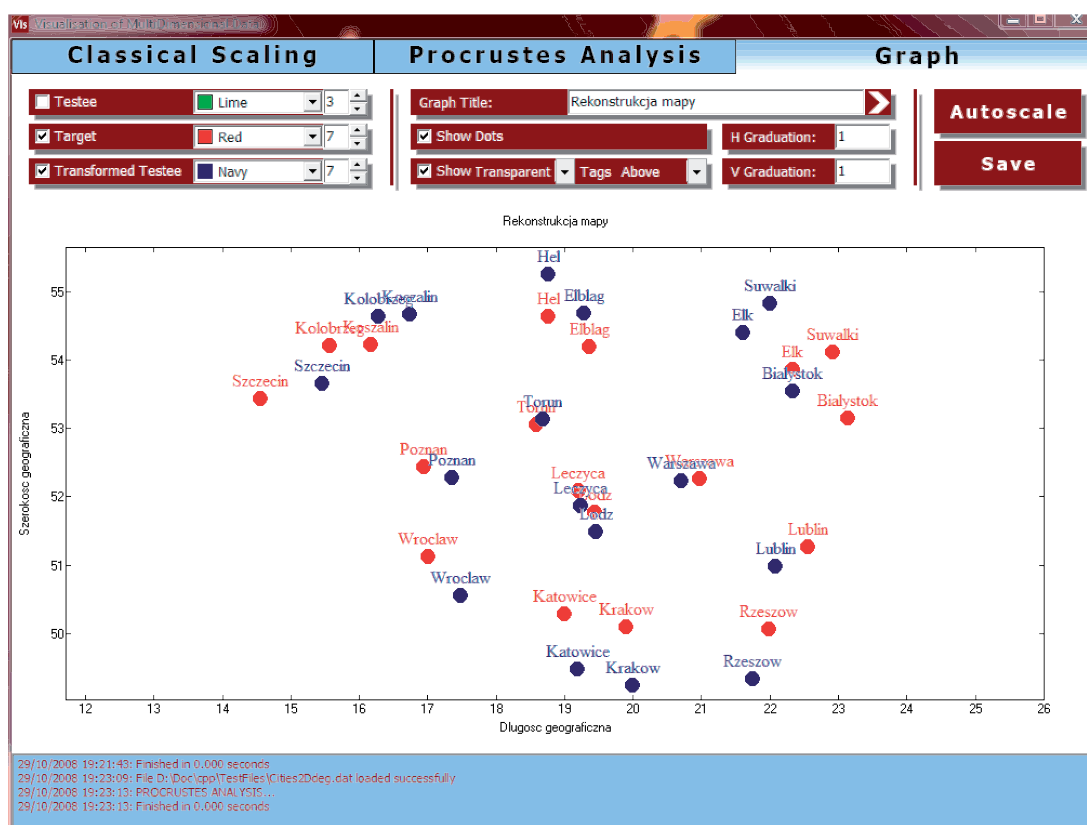
**Rysunek 4.4.** Panel analizy prokrustowej. Dane docelowe to współrzędne geograficzne miast.

### 4.4.2 Panel analizy prokrustowej

Panel analizy prokrustowej został przedstawiony na rysunku 4.4. Dane wejściowe są przesyłane jako wynik klasycznego skalowania lub można je załadować z pliku (przycisk *Load Testee*). Przyciski *Load Target*, *Transform Testee* i *Accuracy* są odpowiedzialne za załadowanie danych docelowych, inicjalizację analizy prokrustowej i wyliczenie dokładności transformacji odpowiednio.

### 4.4.3 Panel wizualizacji

Zamieszczony na rysunku 4.5 panel wizualizacji składa się z obszaru wizualizacji oraz z części zarządzającej. Opcje pozwalają wybrać, które zestawy danych



**Rysunek 4.5.** Panel wizualizacji. Widoczne różnice w dopasowaniu wynikają z faktu, że współrzędne geograficzne to oczywiście współrzędne na sferze.

umieścić na wykresie, przypisać im odpowiednie kolory oraz wielkość. Za rodzaj wyświetlanych informacji są odpowiedzialne funkcje *Show Dots* (punkty)

oraz *Show Tags* (etykiety dla każdego punktu). *H Graduation* i *V Graduation* pozwalają indywidualnie ustalić poziomą i pionową podziałkę wykresu. Możliwa jest również personalizacja wykresu poprzez dodanie własnego opisu wykresu (*Graph Title*), oraz poszczególnych osi (*Horizontal Axis Label* i *Vertical Axis Label*). Przycisk *Autoscale* uruchamia funkcję automatycznie ustawiającą zakres wykresu, tak aby wszystkie dane mogły być wyświetlone. Zapisywanie wykresu do pliku w formacie „jpg” odbywa się po wciśnięciu przycisku *Save*.

Dla obszaru wykresu zostało zaimplementowanych kilka funkcji obsługujących zdarzenia myszy. Obsługiwane zdarzenia zostały zebrane w tabeli 4.4.

Tabela 4.4. Funkcje myszy dla obszaru wykresu

Zdarzenie	Opis
dwuklik lewym przyciskiem	centrowanie i przybliżenie wykresu
dwuklik prawym przyciskiem	centrowanie i oddalenie wykresu
przeciąganie z wciśniętym lewym przyciskiem	przesuwanie obszaru wykresu
przeciąganie z wciśniętym prawym przyciskiem	zaznaczanie i przybliżanie obszaru wykresu

## 5. Podsumowanie

Efektem niniejszej pracy jest powstanie trzech odrębnych aplikacji. Uruchamianych z linii poleceń programów dokonujących klasycznego skalowania i analizy prokrustowej oraz aplikacji z interfejsem graficznym uzupełnionej o moduł wizualizacji dwuwymiarowej. Programy konsolowe zostały stworzone z myślą o przetwarzaniu skryptowym dużej liczby plików. Aplikacja wizualna umożliwia za to kompleksowe przetwarzanie danych wielowymiarowych wraz z końcową wizualizacją danych.

Ważną częścią pracy jest wykonana optymalizacja, zarówno na etapie projektowania algorytmu (rozdział 3) jak i samej implementacji (rozdział 4). W szczególności na etapie klasycznego skalowania udało się zredukować zarówno zapotrzebowanie na pamięć jak i złożoność algorytmu z  $O(n^4)$  do  $O(n^2)$  (rozdział 3.2.1). W całej aplikacji udało się również zachować odpowiednią równowagę pomiędzy wykorzystaniem pamięci operacyjnej i czasu procesora.

### 5.1 Możliwości rozwoju aplikacji

Oczywiście po ukończeniu projektu przychodzi refleksja, że coś możnaby zrobić inaczej lub dodać dodatkowe funkcje, moduły. Stworzona aplikacja jest dość elastyczna i w zasadzie dodanie dodatkowych modułów nie nastroczałoby dużych kłopotów, poza samą implementacją zagadnienia. Jako możliwe drogi rozwoju aplikacji można wymienić, np.:

- Wizualizację 3D – problem polegałby jedynie na implementacji samej wizualizacji trójwymiarowej, najlepiej z wykorzystaniem bibliotek *DirectX*

lub *OpenGL*. Algorytmy klasycznego skalowania oraz analizy prokrustowej są już przystosowane do większej liczby wymiarów.

- ▶ Linia poleceń – aplikacja posiada okno komunikatów, które mogłoby być wykorzystane również jako linia poleceń. Umożliwiłoby to, np. uruchamianie skryptów.
- ▶ Przetwarzanie równoległe – większość obliczeń w programie to działania na dużych macierzach, które świetnie nadają się do zrównoleglenia. Jest to oczywiście funkcja wymagająca gruntownego przebudowania aplikacji, zwłaszcza jej podstaw.
- ▶ Zapisywanie i wznawianie stanu obliczeń – ze względu na długie czasy obliczeń dla dużych zestawów danych funkcja taka byłaby wielce porządana, zwłaszcza dla zagadnienia klasycznego skalowania. Ponieważ najwięcej czasu zabiera wyliczanie wartości i wektorów własnych, funkcja ta powinna być zatem wywoływana na tym etapie. Dodanie tej funkcji wymagałoby pewnych modyfikacji działania aplikacji.



# A. Pliki nagłówkowe

## A.1 Klasa *Matrix*

```
class Matrix
{
private:
    unsigned int row, col;
    double **matrix;

public:
    Matrix(unsigned int row, unsigned int col, double value=0, int fill=0);
    Matrix(const Matrix & m);
    ~Matrix();

    unsigned int nrow() const;
    unsigned int ncol() const;

    void UstawElement(unsigned int row, unsigned int col, double value);
    double PobierzElement(unsigned int row, unsigned int col) const;

    Matrix & operator=(const Matrix & m);
    Matrix & operator+=(const Matrix & m);
    Matrix & operator-=(const Matrix & m);
    Matrix & operator*=(const Matrix & m);
    Matrix & operator*=(double x);

    Matrix & Resize(unsigned int row, unsigned int col);
```

---

```
Matrix & AddRow(double * vec, unsigned int n);
Matrix & AddRows(const Matrix & m);

double * operator[](unsigned int row) const;

//zwraca wskaznik do samej macierzy
double ** pmatrix() const;

Matrix Transpose() const;
double Trace() const;
//max lub min w kolumnach
double ColMax(unsigned int col) const;
double ColMin(unsigned int col) const;
};

Matrix operator+(const Matrix & m1, const Matrix & m2);
Matrix operator-(const Matrix & m1, const Matrix & m2);
Matrix operator*(const Matrix & m1, const Matrix & m2);
Matrix operator*(const Matrix & m, double x);
Matrix operator*(double x, const Matrix & m);

void WypiszM(const Matrix & m);
```

## A.2 Klasa *DataFile*

```
struct file_registry
{
    AnsiString name;
    AnsiString value;
    struct file_registry *next;
};

struct file_section
{
    AnsiString name;
    struct file_registry *list;
    struct file_section *next;
};

typedef struct file_registry file_reg;
typedef struct file_section file_sec;
typedef file_reg *reg_list;
typedef file_sec *sec_list;

class DataFile
{
private:

    //wskaznik do naglowka
    sec_list parameters;
    unsigned int nobjects;
    //tablica z etykietkami
    AnsiString *obj_classes;
    Matrix *data;
    AnsiString filename;

    //sprawdza czy w naglowku istnieje sekcja
    sec_list SecExists(AnsiString name) const;
    //sprawdza czy w sekcji istnieje wpis
```

```
reg_list RegExists(reg_list section, AnsiString name) const;

//analiza i wczytywanie pliku
AnsiString ReadSecName(ifstream &fin);
AnsiString ReadRegName(ifstream &fin);
AnsiString ReadRegValue(ifstream &fin);
AnsiString ReadComment(ifstream &fin);
AnsiString ReadToEOL(ifstream &fin);
AnsiString ReadClass(ifstream &fin, AnsiString separator);
double ReadValue(ifstream &fin, AnsiString separator,
    AnsiString missing_value);
AnsiString ReadValueStr(ifstream &fin, AnsiString separator,
    AnsiString missing_value);

public:

DataFile();
DataFile(AnsiString filename);
DataFile(const DataFile & input_file, const Matrix & output_matrix);
~DataFile();

//zwraca referencje do danych
Matrix & rData();
//zwraca wskaznik do danych
Matrix * pData();
//zwraca wskaznik do etykiet
AnsiString * rClasses();

//czysci obiekt
void Clear();

//laczy naglowek z innymi danymi
void Combine(const DataFile & input_file,
    const Matrix & output_matrix);
//wstawia mowe opisy wymiarow
void ReplaceFeaturesNames();
```

```
//dodaje komentarze do naglowka pliku
void AddComment(AnsiString comment);

//dodaje nowa sekcja do naglowka
sec_list AddSection(AnsiString sec_name);
//wypisuje naglowek na obiekt klasy TMemor
void PrintSections(TMemor *memor);

//dodaje wpisy do sekcji
reg_list AddRegistry(sec_list section, AnsiString reg_name,
    AnsiString reg_value);
reg_list AddRegistry(AnsiString section, AnsiString reg_name,
    AnsiString reg_value);
//wypisuje sekcje
void PrintRegister(sec_list section, TMemor *memor);
void PrintSection(AnsiString section_name, TMemor *memor);
//wypisuje sekcje FEATURES do arkusza
unsigned int PrintFeatures(TStringGrid *tsg);

//kasuje wpisy w sekcji
void DelRegister(sec_list section);
//kasuje naglowek
void DelSections();

//zwraca wartosc wpisu
AnsiString RegValue(AnsiString section, AnsiString registry);
//zwraca nazwe klasy dla numeru obiektu
AnsiString ObjectClassName(unsigned int object_number);

void LoadFile(AnsiString filename);
void SaveFile(AnsiString filename);
};
```

## A.3 Klasa *Graph*

```
class Graph
{
private:

    Graphics::TBitmap *image;
    TCanvas *canvas;
    //dane (x,y,wielkosc,kolor)
    Matrix *matrix;
    //etykiety
    AnsiString *vec;

    double x_min;
    double x_max;
    //podzialka x
    double x_graduation;
    //polozenie znacznika na osi x
    double x_tick_position;
    double y_min;
    double y_max;
    double y_graduation;
    double y_tick_position;

    //przelozenie punktow na odleglosci
    double x_scale;
    double y_scale;

    //odstep wykresu od krawedzi plotna
    int border_offset;

    AnsiString GraphTitle, XAxis, YAxis;
    bool tags;
    bool dots;
    unsigned int tags_position;
    bool transparent_tags;
```

```
//kasuje dane i wpisuje nowe
void FromMatrix(Matrix & m, AnsiString * v);
//dodaje dane
void AddMatrix(Matrix & m, AnsiString * v);
//dodaje etykiety
void AddVec(AnsiString * v, unsigned int n);

public:

Graph();
Graph(Graphics::TBitmap * img);
~Graph();

void Init(Graphics::TBitmap * img);

//dodaje opisy osi
void SetLabels(AnsiString title, AnsiString x, AnsiString y);
//ustawia wyswietlanie etykiet
void SetTags(bool TagsVisible, unsigned int TagsPosition,
             bool TagsTransparent);
//ustawia wyswietlania punktow
void SetDots(bool DotsVisible);
//ustawia zakres
void SetRange(double xmin, double xmax, double ymin, double ymax);
//ustawia podzialke(polozenie_podzialkiX,odleglosc_miedzy_podzialkamiX,
void SetScale(double xtick, double xscale,
              double ytick, double yscale);
//zwraca zakres
void GetRange(double* xmin, double* xmax, double* ymin, double* ymax);
//zwraca podzialke
void GetGraduation(double* xgrad, double* ygrad);

//automatycznie centruje i skaluje
void AutoScale();
//wyswietla w prawidlowych proporcjach
```

```
void Proportional();

//rysuje bez dodawania danych
void DrawCircle(int x, int y, int r, TColor color);
void DrawFrameOnTop(int xmin, int xmax, int ymin, int ymax);
void DrawPoint(double x, double y, int r, TColor color);
void DrawTag(double x, double y, AnsiString label, int size,
    TColor color);
void DrawTxt(double x, double y, AnsiString label, int size,
    TColor color);
//czysci wykres
void ClearGraph();

//dodaje punkt
void AddPoint(double x, double y, int r, TColor color);
//kasuje punkty
void ErasePoints();
//rysuje punkty
void DrawGraph();
//rysuje osie
void DrawAxes();
//rysuje caly wykres
void Plot();

void Pan(double x, double y);
void Zoom(int left, int right, int top, int bottom);
void ZoomIn(int x, int y);
void ZoomOut(int x, int y);

void FromMatrix(Matrix & m, int r, TColor color, AnsiString * v);
void AddMatrix(Matrix & m, int r, TColor color, AnsiString * v);

//zapisuje jako jpg
void ToFile(AnsiString filename);
};
```



## A.4 Klasa *CMDS*

```
class CMDS
{
private:

    DataFile *MDfile;
    DataFile *OUTfile;
    //okresla czy dane wejsciowe maja byc traktowane jak macierz odleglosci
    bool Dist;
    unsigned int NumberOfObjects;
    unsigned int NumberOfFeatures;
    bool FullPrecisionEigensystem;
    unsigned int NumberOfIterations;

    inline const double SQR(const double &a) {return a*a;}
    //wylicza kwadrat odleglosci dla podanych obiektow
    double Dist2(const Matrix & m, unsigned int i, unsigned int j);
    //mnozy macierz x wektor na wskaznikach
    void MultiplyMxV(const Matrix & M, const Matrix & V, Matrix & MxV);
    void MDdataToB(Matrix & B);
    void DistToB(Matrix & B);
    double EigenSystem(const Matrix & B, double & EVal, Matrix * EVec);
    void cmds();
    double Accuracy();

public:

    CMDS(DataFile &MDfile, DataFile &OUTfile);
    ~CMDS();
    double ClassicalScaling(bool FullPrecisionEigensystem=true,
        unsigned int NumberOfIterations=300, bool SquaredDist=false);
    //zwraca wskaznik do podpietego pliku
    DataFile * pMDfile();
    DataFile * pOUTfile();
};
```

## A.5 Klasa *Procrust*

```
class Procrust
{
private:
    DataFile *Testee;
    DataFile *Target;
    DataFile *Destination;

    inline const double MAX(const double &a, const double &b)
        {return b > a ? (b) : (a);}
    inline const double MIN(const double &a, const double &b)
        {return b < a ? (b) : (a);}
    inline const double SIGN(const double &a, const double &b)
        {return b >= 0 ? (a >= 0 ? a : -a) : (a >= 0 ? -a : a);}
    //twierdzenie pitagorasa
    double pythag(const double a, const double b);
    //dekompozycja singularna
    void svdcmp(Matrix &a, Matrix &w, Matrix &v);
    void proc(const Matrix &Y, const Matrix &X, Matrix &YtoX);
    //centruje macierz
    Matrix center_matrix(const Matrix &m);
    //symuluje macierz centrujaca o zadanej liczbie elementow n
    double SimJ(unsigned int n, unsigned int row, unsigned int col);
    void proc_accuracy();

public:
    AnsiString ProcError;
    double Accuracy;
    Procrust(DataFile &Testee, DataFile &Target, DataFile &Destination);
    ~Procrust();
    double ProcrustesAnalysis();
    //zwraca wskaznik do podpietego pliku
    DataFile * pTestee();
    DataFile * pTarget();
    DataFile * pDestination();};
```

# Bibliografia

- [1] Torgerson, W. S., *Classical Scaling: I. Theory And Method*, vol.17, no.4, Psychometrika, Grudzień 1952
- [2] Borg, I., Groenen, P., *Modern Multidimensional Scaling: Theory And Applications*, Nowy York, Springer-Verlag, 1997
- [3] Bronsztejn, I. N., Siemindiajew, K. A., Musiol, G., Mühligh, H., *Nowoczesne Kompendium Matematyki*, Warszawa, Wydawnictwo Naukowe PWN, 2004
- [4] Press, W. H., Teukolsky, S. A., Vetterling, W. T., Flannery, B. P., *Numerical Recipes - The Art of Scientific Computing, Third Edition*, Nowy York, Cambridge University Press, 2007
- [5] Petersen, K., B., Pedersen, M., S., *The Matrix Cookbook*, Strona internetowa <http://matrixcookbook.com>