Dariusz K. Chojecki
Szczecin

O programowaniu w środowisku R na przykładzie algorytmu obliczeniowego dla sezonowości zjawisk (metoda średnich jednoimiennych okresów)

W badaniach nad przeszłością populacji istotne miejsce zajmuje analiza sezonowości ruchu urodzeń (poczęć), małżeństw i zgonów, która doczekała się już szeregu opracowań i nadal jest pogłębiana[1], aczkolwiek należy zauważyć, że zagadnienia związane z cyklicznością napływu i odpływu ludności, z racji niedostatków źródłowych, dla epoki przedstatystycznej czekają na lepsze oświetlenie. W sezonowości zjawisk demograficznych uwidacznia się wpływ klimatu, obyczaju, religii, gospodarki i szeregu innych czynników powiązanych ze sobą. Jest to więc zjawisko ciekawe pod względem badawczym, dobrze udokumentowane i z reguły rozpatrywane w kontekście wyników uzyskanych na podstawie metody średnich jednoimiennych okresów[2], bo najlepiej kwalifikującej się do oceny addytywnych wahań w czasie bez wyraźnego trendu[3] (zjawisko charakterystyczne dla starego porządku demograficznego). Rzeczona metoda stanowi tu tylko pretekst do zapoznania czytelnika z możliwościami programowania w środowisku R. Jest to jednakże „nie byle jaki" pretekst, gdyż język interesującego nas niekomercyjnego programu[4] zyskał status *lingua statistica*. Pozwala on tworzyć własne rozwiązania obliczeniowe dla specyficznych problemów, w tym także związanych z demografią

[1] Zob. podsumowanie wyników badań: Cezary Kuklo, *Demografia Rzeczypospolitej przedrozbiorowej*, Warszawa 2009, s. 298–303, 351–353, 409–413.

[2] Zob. też Edmund Piasecki, *Problem wartości poznawczej wskaźników sezonowości zgonów*, „Przeszłość Demograficzna Polski" (dalej: PDP) 13, 1981, s. 49–65; Konrad Wnęk, *Metody badania korelacji sezonowości zgonów ze zjawiskami klimatycznymi*, PDP 20, 1997, s. 36, 47.

[3] Por. Mieczysław Sobczyk, *Statystyka opisowa*, Warszawa 2010, s. 195, 200.

[4] Na polskim rynku wydawniczym jest już kilka publikacji propagujących środowisko R. John M. Quick, *Analiza statystyczna w środowisku R dla początkujących*, Gliwice 2012 (napisany dla historyków zajmujących się aspektami militarnymi); zwięźle i krótko: Łukasz Komsta, *Wprowadzenie do środowiska R* (www.cran.r-project.org/doc/contrib/Komsta-Wprowadzenie.pdf - tylko pdf); ciekawie: Przemysław Biecek, *Przewodnik po pakiecie R*, wyd. 3, Wrocław 2014 (www.biecek.pl/R/

historyczną, a co ważne dzielić się tymi rozwiązaniami z innymi użytkownikami, wymieniając się kodem programowym w zwykłym pliku tekstowym[5] czy tworząc pakiety — tak zwane paczki, które zawierają, aby użyć przenośni, moc prezentów dla badaczy różnych dziedzin nauki.

Żeby móc analizować kod programowy (tu: dłuższa „wypowiedź" informatyczna), należy najpierw poznać funkcje, które go tworzą. Przyjmując taką optykę postrzegania spraw, nie powinnien dziwić czytelnika fakt, że droga dojścia do przedstawionego kodu jest długa i odmierzana małymi krokami. Każdy krok jest zaplanowany z myślą o osobie, która do tej pory nie miała do czynienia z programowaniem i środowiskiem R (piszący te słowa też nie zalicza się do ekspertów[6]). Swoistym podsumowaniem tego „inicjującego" artykułu są dwa bloki kodów programowych funkcji *MSJO1* (dla S_i) i *MSJO2* (dla g_i) przeznaczonych do pomiaru relatywnych i absolutnych wahań sezonowych zgodnie z poniższymi wzorami[7].

$$S_i = \frac{\bar{y}_i \cdot d}{\sum_{i=1}^{\mathrm{d}} \bar{y}_i} \cdot 100$$

gdzie: S_i — wskaźnik sezonowości dla i-tego podokresu cyklu sezonowości, d — liczba podokresów (na przykład półrocza — 2, kwartały — 4, miesiące — 12), $\bar{y}_i$ — średnia jednoimiennych (pod)okresów — na przykład dla poszczególnych miesięcy — obliczona z badanych cykli rocznych.

Suma wskaźników w wypadku sezonowości miesięcznej wynosi 1200, kwartalnej — 400, półrocznej — 200 (średni poziom zjawiska przyjęty za 100).

$$g_i = \bar{y} \cdot \left(\frac{S_i}{100} - 1\right)$$

gdzie: g_i — absolutne poziomy wahań sezonowych wyrażone w jednostkach zjawiska, $\bar{y}$ — średnia ogólna, s_i — wskaźnik sezonowości dla i-tego podokresu cyklu sezonowości.

Suma absolutnych wahań sezonowych wynosi 0 (średni poziom zjawiska przyjęty za 0).

Szczególnie w wypadku wartości miesięcznych, z powodu różnej długości miesięcy, zachodzi potrzeba przeliczeń na równą liczbę dni w danej jednostce czasu. Ich prostą formułę przedstawia poniższy wzór:

PrzewodnikPoPakiecieRWydanieIIIinternet.pdf - obszerna część); specjalistycznie: Marek Walesiak, Eugeniusz Gatnar, *Statystyczna analiza danych z wykorzystaniem programu R*, Warszawa 2012.

[5] Warto mieć na uwadze, że w notatniku kod programowy musi zostać pozbawiony znaków „+" na początku wierszy, by mógł zadziałać po wklejeniu do konsoli R.

[6] Na forum Polskiej Grupy Użytkowników R (https://groups.google.com/forum/#!forum/polska-grupa-uzytkownikow-r) otrzymałem cenne informacje, które pozwoliły mi rozwiązać dwa problemy.

[7] M. Sobczyk, *Statystyka* [3], s. 200 n.

$$y_{t_0} = \frac{y_t \cdot t_0}{z}$$

gdzie: y_{t_0} — „standaryzowany", przeliczony poziom zjawiska, y_t — faktyczny poziom zjawiska w danej jednostce czasu, t_0 — liczba dni w danej jednostce czasu przyjęta za podstawę porównań, z — rzeczywista liczba dni w danej jednostce czasu.

Przygotowanie do pracy

Po otarciu konsoli środowiska R czyścimy informacje o programie, używając skrótu klawiaturowego Ctrl + L. Następnie sprawdzamy lokalizację bieżącego folderu roboczego, wpisując po znaku zachęty „>" funkcję: getwd(). Jej wynik, po zatwierdzeniu enterem, stanowi ścieżka dostępu, która może być różna od podanej niżej.

```
> getwd()
[1] "C:/Users/DC/Documents"
```

Zapis ten oznacza, że program będzie pobierał domyślnie dane z folderu roboczego o nazwie „Documents" lub je w nim zapisywał. Oczywiście dane mogą być przechowywane w różnych lokalizacjach, na przykład na pulpicie w folderze o nazwie „StatDem". Aby ten ostatni stał się domyślnym, należy posłużyć się funkcją setwd() i jako jej argument[8] podać określoną ścieżkę dostępu.

```
> setwd("C:/Users/DC/Desktop/StatDem")
```

Żeby sprawdzić zawartości folderu, to jest znajdujące się w nim nazwy plików, stosujemy funkcję:

```
> list.files()
```

Jej wynikiem może być lista nazw plików, a wypadku gdy folder jest pusty — komunikat:

```
character(0)
```

Załóżmy, że w folderze roboczym znajdują się pliki z danymi, na których chcemy wykonać stosowne obliczenia. Jeśli korzystamy z programu Excel, to zbiory naszych danych powinny być zapisane w formacie .csv (rozdzielone przecinkami). Konwersja pliku o rozszerzeniu xlsx na csv sprowadza się do wybrania następujących poleceń: zapisz jako; inne formaty; csv (rozdzielany przecinkami). Należy

[8] Argument czy argumenty funkcji są podawane w okrągłych nawiasach. Przywołana funkcja: getwd() — jest bezargumentową.

pamiętać, że zmianę formatu można wykonać tylko dla ramki danych — zestawienia tabelarycznego — przechowywanych w jednej zakładce (arkuszu). Jeśli w pliku .xlsx jest więcej zakładek, to przed konwersją informacje z każdej zakładki muszą być zapisane w odrębnych „jednozakładkowych" plikach arkusza kalkulacyjnego.

Niech w folderze StatDem znajduje się plik z przykładowymi danymi w preferowanym formacie o nazwie „zgonyNiemSzczecin1889_1891.csv". Do wyświetlenia przechowywanych w nim informacji posłużmy się funkcją: read.csv2(). Kolumny wyświetlanej ramki powinny mieć nagłówki, innymi słowy nazwy, gdyż drugi atrybut tej funkcji — header — jest domyślnie ustawiony[9] na TRUE (zob. też wprowadzenie i uwagi w tekście Radosława Poniata). Zastosowanie tej funkcji jest najodpowiedniejsze w wypadku importu danych zapisanych w polskojęzycznej wersji Excela. Importowaną ramkę — zmienną — nazwijmy *mojeDane*[10].

```
> mojeDane <- read.csv2("zgonyNiemSzczecin1889_1891.csv")
> mojeDane
```

	Rok	Miesiac	ZgonyNiemowlat
1	1889	1	71
2	1889	2	47
3	1889	3	58
4	1889	4	75
5	1889	5	82
6	1889	6	217
7	1889	7	185
8	1889	8	90
9	1889	9	121
10	1889	10	67
11	1889	11	63
12	1889	12	103
13	1890	1	66
14	1890	2	93
15	1890	3	89
16	1890	4	82
17	1890	5	69
18	1890	6	131

[9] read.csv2(file, header=TRUE, ...)

[10] Nazwy zmiennych nadawane przez użytkownika nie mogą mieć polskich znaków diakrytycznych, spacji. Ponadto nie mogą zaczynać się od liczb i znaków specjalnych. W ich interpretacji ważna jest wielkość liter. W artykule rzeczone nazwy pisane są kursywą.

```
 19        1890          7          185
 20        1890          8          145
 21        1890          9           87
 22        1890         10           74
 23        1890         11           72
 24        1890         12           85
 25        1891          1           81
 26        1891          2           62
 27        1891          3           77
 28        1891          4           73
 29        1891          5           84
 30        1891          6          109
 31        1891          7          266
 32        1891          8          210
 33        1891          9          153
 34        1891         10           96
 35        1891         11           78
 36        1891         12           84
>
```

Liczba porządkowa wierszy ramki jest wygenerowana automatycznie. Dostęp do określonych elementów ramki uzyskujemy za pomocą „kwadratowej" notacji nawiasowej. I tak, aby wyświetlić przykładowo wszystkie elementy trzeciej kolumny należy napisać:

```
> mojeDane[,3]
```

[...] (Pominięte wyniki wyświetlenia z racji ich ilości)

Aby wyświetlić ostatni wiersz:

```
> mojeDane[36,]
    Rok Miesiac ZgonyNiemowlat
36 1891      12             84
>
```

Aby wyświetlić czwartą wartość w trzeciej kolumnie:

```
> mojeDane[4,3]
[1] 75
>
```

Ogromną zaletą pracy w środowisku R jest to, że właściwie nie musimy wyświetlać całej ramki danych, by na niej pracować. Wystarczy odwoływać się do jej ogólnej nazwy (tu: *mojeDane*) czy, jeśli zachodzi taka potrzeba, do nazw nagłówków[11], do których wyświetlenia można użyć funkcji: names()

```
> naglowki <- names(mojeDane)
> naglowki
[1] "Rok"          "Miesiac"      "ZgonyNiemowlat"
```

Liczba w nawiasie kwadratowym jest indeksem — pozycją danego elementu w wektorze składającym się tu z trzech elementów. A zatem "Miesiac" powinien mieć wartość indeksu 2. Sprawdźmy:

```
> names(mojeDane[2])
[1] "Miesiac"
```

albo krócej, bo odwołując do nazwy wektora

```
> naglowki[2]
[1] "Miesiac"
```

Wynik jest jednoelementowym wektorem, dlatego też przy łańcuchu znaków (napisie) "Miesiac" widnieje wartość indeksu równa 1.

Krótkie wprowadzenie do programowania

Wprawdzie w środowisku R istnieje kilkaset, o ile nie kilka tysięcy gotowanych funkcji, ale jego siła polega właśnie na możliwości tworzenia na ich podstawie kolejnych rozwiązań dopasowanych do własnych potrzeb badawczych i dzielenia się nimi z innymi użytkownikami, także w dziedzinie badań demograficznych. Nasze krótkie wprowadzenie w tematykę rozpocznijmy od opisu funkcji użytkownika, czyniąc to na prostych przykładach. Składnia funkcji wygląda następująco:

```
> nazwaFunkcji <- function(argument1, argument2, ...) {
+ instrukcja
+ }
>
```

Niech nasza funkcja oblicza pole trapezu:

```
> poleTrapezu <- function(bokA, bokB, wysokoscH) {
+ 0.5*(bokA + bokB)*wysokoscH
```

[11] Stosowanie w nagłówkach polskich znaków diakrytycznych może prowadzić do powstawania problemów informatycznych.

```
+ }
>
```

Nazwa funkcji powinna w jakiejś mierze oddawać jej przeznaczenie, aczkolwiek w tym względzie nie ma zasad, poza podanymi wyżej, w przypisie 10. Nasza funkcja posiada trzy argumenty oddzielone od siebie przecinkiem (znak separacji!). Ich poszczególne nazwy są pisane bez spacji. W tym wypadku mają one za zadanie przenosić wartości liczbowe do instrukcji, w której zostanie wykonane określone działanie, tu — obliczenie pola trapezu. Ważne jest, by takimi samymi nazwami argumentów posługiwać się w części instrukcji. Symbol „+" na początku każdego wiersza jest znakiem przeniesienia (przejście enterem o wiersz niżej); będzie on rozpoczynał każdy kolejny wiersz do momentu zakończenia pisania funkcji. Jeśli pisanie danej funkcji nie daje się z jakiś powodów zakończyć — na przykład niezamknięcie cudzysłowia czy nawiasu w odpowiednim miejscu — to należy użyć przycisku ucieczki (Esc), by je przerwać. Oczywiście, daną funkcję można zapisać w jednym wierszu, ale traci ona na przejrzystości, im dłuższy jest jej zapis (w dalszej części symbol „↑" będzie oznaczał, że chodzi o ten sam wiersz polecenia, który w edytorze tekstu, z powodu braku miejsca, uległ zawinięciu). Spacje wewnątrz funkcji nie odgrywają roli, aczkolwiek odpowiednie ich stosowanie może wpływać na przejrzystość kodu programowego.

Aby wyświetlić treść funkcji, należy napisać jej nazwę i zatwierdzić klawiszem Enter.

```
> poleTrapezu
function(bokA, bokB, wysokoscH) {
0.5*(bokA + bokB)*wysokoscH
}
>
```

W powyższym wypadku sprawdzenie działania funkcji odbywa się poprzez podstawienie za argumenty funkcji odpowiednich wartości liczbowych. Niech bok A ma 40 jednostek, bok B — 60, a wysokość H — 20.

```
> poleTrapezu(40,60,20)
[1] 1000
>
```

Funkcja *poleTrapezu*(), mimo że posiada trzy argumenty, jest przeznaczona do obliczeń powierzchni tylko jednej figury płaskiej. W celu zwiększenia „zadaniowości" funkcji można wykorzystać instrukcję warunkową typu if ... else ... Jej składnia wygląda następująco:

```
> if(warunek) {
+ instrukcja1
+ } else {
```

```
+ instrukcja2
+ }
>
```

Instrukcja1 jest wykonywana wtedy, gdy warunek jest prawdziwy (wartość logiczna TRUE), natomiast instrukcja2, gdy warunek jest niespełniony (FALSE). Warto zapamiętać, że często stosowany operator porównania wymaga dwóch znaków równości. Ponadto łańcuchy znaków zapisujemy w cudzysłowie, bez użycia którego byłyby one traktowane przez interpreter jako nazwy zmiennych czy funkcji, co z kolei prowadziłoby do powstawania błędów. Prosty, poprawny przykład:

```
> if(2 + 2 == 5) {
+ "To prawda"
+ } else {
+ "To nieprawda"
+ }
[1] "To nieprawda"
>
```

Oczywiście, że to nieprawda, skoro lewa strona równania pod względem wyniku nie jest taka sama jak prawa:

```
> 2 + 2 == 5
[1] FALSE
>
```

Mając podstawową wiedzę o instrukcji warunkowej, napiszmy własną funkcję *pola*, która będzie obliczać powierzchnię prostokąta bądź trapezu w zależności od naszej potrzeby. Wykorzystajmy nazwy argumentów te same co w funkcji *poleTrapezu* i poszerzmy ich listę o jeszcze jeden argument, mianowicie tryb. Przyjmijmy, że jeżeli wartość tego ostatniego będzie równa 1, to funkcja *pola* obliczy powierzchnię pierwszej z wymienionych figur płaskich, 2 — drugiej, a gdy większa od 2, to wyświetli odpowiedni komunikat tekstowy — łańcuch znaków. Ponieważ w tym wypadku mamy do czynienia z trzema opcjami, to posłużymy się zagnieżdżeniem instrukcji warunkowej, innymi słowy, osadzeniem jej w części instrukcja2. Według schematu rzecz powinna wyglądać następująco (pogrubiona zagnieżdżona instrukcja warunkowa):

```
> if(warunek1) {
+ instrukcja1
+ } else {
+ if(warunek2) {
+ instrukcja2
+ } else {
+ instrukcja3
+ }
```

```
+ }
>
```

Praktyka:

```
> pola <- function(tryb, bokA, bokB, wysokoscH) {
+ if(tryb == 1) {
+ bokA * bokB
+ } else {
+ if(tryb == 2) {
+ 0.5*(bokA + bokB)*wysokoscH
+ } else {
+ "Argument tryb wymaga wartości 1 (pole kwadratu) albo 2 (pole trapezu)"
+ }
+ }
+ }
>
```

Przetestujmy działanie utworzonej funkcji *pola*. Niech bok A ma 20 jednostek, bok B — 30, wysokość H — 40.

```
> pola(1,20,30,40)
[1] 600
> pola(2,20,30,40)
[1] 1000
> pola(3,20,30,40)
[1] "Argument tryb wymaga wartości 1 (pole kwadratu) albo 2 (pole trapezu)"
> pola(,20,30,40)
Error in tryb == 1 : 'tryb' is missing
```

W ostatnim wypadku w funkcji pola nie podano wartości dla pierwszego argumentu. Czy zawsze trzeba go podawać? Nie. Można bowiem zadeklarować domyślną wartość argumentu. Do tego celu używamy jednego znaku równości. Jeśli chcielibyśmy, żeby funkcja obliczała pole prostokąta bez podania wartości przed pierwszym przecinkiem, wówczas argument tryb powinien wyglądać następująco:

```
> pola2 <- function(tryb = 1, bokA, bokB, wysokoscH) {
[...] (pozostała treść (ciało) funkcji bez zmian)
}
>
```

Sprawdzenie:

```
> pola2(,20,30,40)
[1] 600
> pola2(1,20,30,40)
[1] 600
```

```
> pola2(2,20,30,40)
[1] 1000
```

Często zachodzi potrzeba wykonania określonych operacji na wektorze wartości logicznych, na „ciągach" danych. Stosowanie w tej sytuacji instrukcji warunkowej typu if ... else nie jest wskazane, gdyż zwraca ona tylko wartość przypisaną do jednego warunku logicznego. Jeśli zatem chcemy pracować na całym wektorze i przekształcać wartości jego elementów *en bloc* w zależności od warunku, to powinniśmy skorzystać z funkcji: ifelse(). Składnia tej funkcji jest następująca:

```
> ifelse(warunek, instrukcja1, instrukcja2)
```

Instrukcja1 jest wykonywana dla tych elementów wektora, dla których spełniony jest warunek (TRUE), a instrukcja2 dla pozostałych (FALSE).

```
> jakiesDane <- c(20,25,50,40,30,25,35,20)
> jakiesDane
[1] 20 25 50 40 30 25 35 20
```

Na podstawie tego ośmioelementowego wektora stwórzmy wektor wartości logicznych, w którym TRUE będzie odpowiadać wartościom wyższym od średniej arytmetycznej w zbiorze danych. Do wyznaczenia przeciętnej użyjmy gotowej funkcji: mean().

```
> srednia <- mean(jakiesDane)
> srednia
[1] 30.625
> prawdaZeWieksze <- jakiesDane > srednia
> prawdaZeWieksze
[1] FALSE FALSE  TRUE  TRUE FALSE FALSE  TRUE FALSE
>
```

Jak widać, wartość logiczna TRUE występuje w wektorze *prawdaZeWieksze* na trzecim, czwartym i siódmym miejscu, co odpowiada pozycji ponadprzeciętnych wartości w wektorze *jakiesDane*, wynoszących kolejno 50, 40 i 45. Dla tych elementów, używając funkcji ifelse(), przypiszmy w nowym wektorze łańcuch znaków „większa", a dla pozostałych — o wartościach niższych od przeciętnej — „mniejsza".

```
> mniejszaWieksza <- ifelse(jakiesDane > srednia, "większa", "mniejsza")
> mniejszaWieksza
[1] "mniejsza" "mniejsza" "większa"  "większa"  "mniejsza" "mniejsza" "większa" "mniejsza"
>
```

Co ważne, z pomocą funkcji ifelse() można także wykonywać odpowiednie przeliczenia. Przyjmijmy, że wartości większe od średniej z wektora *jakiesDane*

mają zostać podniesione dziesięciokrotnie, natomiast mniejsze — tylekroć obniżone. Wyniki tych działań zapiszmy w nowym wektorze o nazwie *pomnozonePodzielone*.

```
> pomnozonePodzielone <- ifelse(jakiesDane > srednia, jakiesDane * 10, jakiesDane / 10)
> pomnozonePodzielone
[1]   2.0   2.5 500.0 400.0   3.0   2.5 350.0   2.0
```

Podobnie jak w instrukcji if... else..., tak i w funkcji ifelse() można dokonywać zagnieżdżeń w części instrukcja2, co przedstawia poniższy schemat, w którego zapisie pogrubiono zagnieżdżoną funkcję:

```
> ifelse(warunek1, instrukcja1, ifelse(warunek2, instrukcja2, instrukcja3))
```

Zanim to uczynimy, najpierw zmodyfikujmy nieznacznie wektor *jakiesDane*, dodając doń element o wartości 15. Powiększony w ten sposób zbiór informacji zapiszmy pod nazwą *jakiesDane2*.

```
> jakiesDane
[1] 20 25 50 40 30 25 35 20
> jakiesDane2 <- c(15, jakiesDane)
> jakiesDane2
[1] 15 20 25 50 40 30 25 35 20
>
```

Wykonajmy następujące działanie: operując na wektorze *jakiesDane2* przypiszmy jego wartością mniejszym od dwudziestu pięciu łańcuch znaków „mniejsze od 25", równym dwadzieścia pięć — „równe 25", większym od dwudziestu pięciu — „większe od 25". Wyniki zapiszmy w nowym wektorze pod nazwą *mniejszeRowneWieksze*.

```
> mniejszeRowneWieksze <- ifelse(jakiesDane2 == 25, "równe 25", ifelse(jakiesDane2 < 25,
↑"mniejsze od 25", "większe od 25"))
> mniejszeRowneWieksze
[1] "mniejsze od 25" "mniejsze od 25" "równe 25"      "większe od 25"  "większe od 25"
↑"większe od 25"  "równe 25"      "większe od 25"  "mniejsze od 25"
>
```

Zgodnie z zapisem warunków funkcja ifelse() najpierw sprawdza, czy prawdą jest, że dany element w wektorze *jakiesDane2* ma wartość 25, jeżeli tak, to zastępuje ten element łańcuchem znaków „równe 25", a jeżeli nie, to w części instrukcja2 zagnieżdżoną funkcją ifelse() rozpatruje kolejny warunek, to znaczy, czy prawdą jest, że dany element w wektorze *jakiesDane2* ma wartość mniejszą od 25. Jeżeli tak jest, to zastępuje ten element łańcuchem znaków „mniejsze od 25". W wypadku gdy ani pierwszy, ani drugi warunek nie jest spełniony, funkcja odwołuje się do części instrukcja2 w zagnieżdżonej funkcji ifelse(), czyli do „większe od 25" i zastępuje

tym ostatnim łańcuchem znaków pozostałe wartości z wektora *jakiesDane2*. W tym przykładzie kolejność warunków nie ma znaczenia.

Ostatnim zagadnieniem, niezbędnym do zrozumienia prezentowanych dalej funkcji, jest stosowanie operatorów logicznych, to jest sumy logicznej kojarzonej ze spójnikiem „lub" (znak tak zwanej rury „|") oraz iloczynu logicznego kojarzonego ze spójnikiem „i" (znak „&"). Istnieją także inne operatory logiczne, ale nie ma tu potrzeby ich omawiania.

Z pomocą funkcji ifelse() stwórzmy nowy wektor danych o nazwie *sumaLogiczna*. Niech powstanie on na podstawie wektora *jakieśDane2*, którego część elementów zostanie przekształcona. Elementy o wartościach równych 15 lub 30 lub 40 (TRUE) będą dziesięciokrotnie zwiekszone, natomiast reszta pozostanie bez zmian.

```
> jakiesDane2
[1] 15 20 25 50 40 30 25 35 20
> sumaLogiczna <- ifelse(jakiesDane2 == 15 | jakiesDane2 == 30 | jakiesDane2 == 40,
↑jakiesDane2 * 10, jakiesDane2)
> sumaLogiczna
[1] 150  20  25  50 400 300  25  35  20
```

Należy zwrócić uwagę, że zapis warunków w powyższej funkcji jest „tasiemcowaty" z racji powtarzania nazwy zmiennej, operatora logicznego i porównania. Problem ten można obejść, tworząc specjalny wektor składający się z testowanych wartości, poprzedzony specjalnym zestawem znaków %in% (zawiera). Poniższy, zmodyfikowany zapis warunku zinterpretujmy następująco: sprawdź czy wektor *jakiesDane2* zawiera elementy o wartości 15 lub 30 lub 40, a jeśli tak, to przypisz im wartość logiczną TRUE (wykonanie działania w instrukcja1), pozostałym zaś FALSE (wykonanie działania w intrukcja2).

```
> sumaLogiczna2 <-  ifelse(jakiesDane2 %in% c(15,30,40), jakiesDane2 * 10, jakiesDane2)
> sumaLogiczna2
[1] 150  20  25  50 400 300  25  35  20
```

Drugi z operatorów logicznych używany jest do „łowienia" wartości, dla których zachodzą równocześnie określone warunki; w poniższym przykładzie tylko te elementy wektora *jakiesDane2* mają atrybut TRUE, które są większe lub równe 25 i zarazem mniejsze lub równe 40, czyli elementy o wartościach zawierających się w przedziale wyznaczonym powyższymi liczbami. Zgodnie z zapisem instrukcji funkcji ifelse() tylko one są mnożone razy dziesięć, podczas gdy reszta pozostaje bez zmian w nowo utworzonym wektorze o nazwie *iloczynLogiczny*.

```
> iloczynLogiczny <- ifelse(jakiesDane2 >= 25 & jakiesDane2 <= 40, jakiesDane2 * 10,
↑jakiesDane2)
> iloczynLogiczny
```

```
[1]  15  20 250  50 400 300 250 350  20
>
```

Opis pozostałych funkcji wykorzystywanych w kodzie programowym

Wartość minimalna w wektorze: min()

```
> jakiesDane2
[1] 15 20 25 50 40 30 25 35 20
> jakiesDane2najmniejsza <- min(jakiesDane2)
> jakiesDane2najmniejsza
[1] 15
>
```

Wartość maksymalna w wektorze: max()

```
> jakiesDane2najwieksza <- max(jakiesDane2)
> jakiesDane2najwieksza
[1] 50
>
```

Suma wartości elementów wektora: sum()

```
> jakiesDane2suma <- sum(jakiesDane2)
> jakiesDane2suma
[1] 260
>
```

Ponieważ argumentami funkcji subset() i tapply() są wektory, ich działanie ukażmy na przykładzie pewnej ramki danych utworzonej funkcją:

```
> data.frame(wektor1, wektor2, ...)
```

Niech w ramce danych znajdzie się nowy wektor *plecBadanych* i dobrze nam zanany wektor *jakiesDane2*.

```
> plecBadanych <- c("kobieta","kobieta","mężczyzna","mężczyzna","kobieta","mężczyzna",
↑"kobieta","mężczyzna","kobieta")
> plecBadanych
[1] "kobieta"   "kobieta"   "mężczyzna" "mężczyzna" "kobieta"   "mężczyzna" "kobieta"
↑"mężczyzna" "kobieta"
> jakasRamka <- data.frame(plecBadanych, jakiesDane2)
> jakasRamka
```

```
  plecBadanych jakiesDane2
1      kobieta         15
2      kobieta         20
3    mężczyzna         25
4    mężczyzna         50
5      kobieta         40
6    mężczyzna         30
7      kobieta         25
8    mężczyzna         35
9      kobieta         20
>
```

Funkcją subset() można tworzyć wektory — podzbiory danych spełniających określony warunek.

```
> subset(ramka, warunek)
```

Odwołując się do *jakasRamka*, utwórzmy na podstawie *jakiesDane2* wektor, który będzie prezentował tylko informacje liczbowe dla kobiet (zobacz wcześniejsze uwagi na temat „kwadratowej" notacji nawiasowej).

```
> jakiesDane2kobiety <- subset(jakasRamka[,2], jakasRamka[,1] == "kobieta")
> jakiesDane2kobiety
[1] 15 20 40 25 20
>
```

Utwórzmy też nowy wektor, w którym znajdą się dane o płci osób mających ponadprzeciętne wartości zmiennej *jakiesDane2* (więcej niż 28).

```
> jakiesDane2plecPonad <- subset(jakasRamka[,1], jakasRamka[,2] > mean(jakasRamka[,2])
> jakiesDane2plecPonad
[1] mężczyzna kobieta   mężczyzna mężczyzna
Levels: kobieta mężczyzna
>
```

Wielce przydatną funkcją w obliczaniu określonych charakterystyk statystycznych dla określonych kategorii czy wariantów zmiennej jest: tapply()

```
> tapply(wektor, indeks, funkcja)
```

Argument wektor odnosi się do danych, na których są wykonywane obliczenia, indeks — do kategorii bądź wariantów, dla których te wyliczenia są przedstawiane, funkcja — do określonego sposobu działania.

Mając na uwadze powyższe obliczmy dla kobiet i mężczyzn średnie i maksymalne wartości zmiennej *jakiesDane2*.

```
> jakiesDane2plecSrednia <- tapply(jakasRamka[,2], jakasRamka[,1], mean)
> jakiesDane2plecSrednia
plecBadanych
  kobieta mężczyzna
       24         35
> jakiesDane2plecNajwieksze <- tapply(jakasRamka[,2], jakasRamka[,1], max)
> jakiesDane2plecNajwieksze
plecBadanych
  kobieta mężczyzna
       40         50
>
```

Opis kodu programowego dla metody średnich w jednoimiennych okresach (MSJO)

Ogólne informacje o przygotowanej funkcji:

```
> MSJO1(nazwa ramki, granica dolna, granica górna, tryb)
```

1. (punkty odnoszą się do wierszy kodu programowego)

Funkcja *MSJO1* (Metoda średnich w jednoimiennych okresach 1) oblicza relatywne wahania zjawiska. Działa na trójkolumnowej ramce posiadającej nagłówki (zobacz ramkę *mojeDane*). W pierwszej kolumnie rzeczonej ramki muszą znajdować się dane o okresach, w drugiej — o podokresach, w trzeciej — o poziomie zjawiska. Nagłówki kolumn mogą mieć dowolne nazwy. Wymagany jest liczbowy typ danych; dla miesięcy w zakresie od 1 do 12 (możliwość przeprowadzenia standaryzacji — przeliczeń na jednakową liczbę dni), dla kwartałów — od 1 do 4, dla półroczy — od 1 do 2.

Funkcja posiada cztery argumenty, przy czym podstawienie danych do pierwszego jest konieczne (nazwa ramki), do pozostałych zaś nie. Drugi i trzeci argument funkcji, to jest dolna i górna granica przedziału czasowego, są ustawione domyślnie na pierwszy rok i ostatni. Czwarty argument funkcji: tryb obliczeń, ustawiony jest domyślnie na 1. Dla tej wartości nie są wykonywane przeliczenia na równą liczbę dni w miesiącu, podczas gdy dla wartości 2 — tak. UWAGA: w obliczeniach sezonowości na podstawie danych półrocznych lub kwartalnych nie należy wybierać trybu 2 (dla tego rodzaju danych brak przeliczeń na równą liczbę dni).

Przykłady wprowadzania danych do funkcji:

```
> MSJO1(mojaRamka)
> MSJO1(mojaRamka, , , )
> MSJO1(mojaRamka, , , 1)
```

```
> MSJO1(mojaRamka, , , 2)
> MSJO1(mojaRamka, 1889, 1891, )
> MSJO1(mojaRamka, 1889, 1891, 1)
> MSJO1(mojaRamka, 1889, 1891, 2)
> MSJO1(mojaRamka, 1889, , )
> MSJO1(mojaRamka, 1889, , 2)
> MSJO1(mojaRamka, ,1891, 2) itp.
```

Uwaga: użycie wartości większej od 2 w ostatnim argumencie funkcji albo użycie wartości 2 w wypadku innych danych niż miesięczne powoduje wyświetlenie komunikatu:

```
[1] "W argumencie tryb wprowadzono nieprawidłową wartość. W wypadku danych
↑kwartalnych i półrocznych nie należy podawać wartości 2 (brak procedury
↑standaryzacyjnej)" (zob. pkt. 14).
```

2.

Utworzenie wieloelementowego wektora wartości podokresów, na przykład miesięcy, dla dokonanego przez użytkownika wyboru przedziału czasowego.

3.

Utworzenie wieloelementowego wektora wartości poziomu zjawiska dla dokonanego przez użytkownika wyboru przedziału czasowego.

4.

Utworzenie jednoelementowego wektora o liczbie podokresów — we wzorze: d (zob. wyżej, s. 76).

5.

Obliczenie średnich arytmetycznych w jednoimiennych (pod)okresach, na przykład dla poszczególnych miesięcy albo kwartałów, albo półroczy — we wzorze: $\bar{y}_i$ (zob. wyżej, s. 76).

6 — 8/9.

Obliczenie relatywnych wahań sezonowych na podstawie niestandaryzowanych danych (tryb 1).

8/9 — 13.

Obliczenie relatywnych wahań sezonowych na podstawie standaryzowanych danych, ale tylko dla miesięcznych wartości (tryb 2). Tu też utworzenie za pomocą funkcji ifelse() wieloelementowego wektora standaryzowanych danych — poziomy zjawiska przeliczone na trzydzieści dni, wykorzystanego do dalszych obliczeń wskaźnika.

Ciało drugiej funkcji *MSJO2* (absolutne wahania sezonowe) właściwie nie różni się zasadniczo. Należy tylko zwrócić uwagę, że w wierszu 6 i 13 pojawia się wektor średniej ogólnej, i że w wierszu 8 i 14 mamy zmodyfikowany kod pod kątem obliczeń wahań absolutnych.

```
1) > MSJO1 <- function(ramka, granicaDolna = min(ramka[,1]), granicaGorna =
↑ max(ramka[,1]), tryb = 1) {
2) + wektorPodokresWybor <- subset(ramka[,2], ramka[,1] >= granicaDolna & ramka[,1] <=
↑ granicaGorna)
3) + wektorPoziomWybor <- subset(ramka[,3], ramka[,1] >= granicaDolna & ramka[,1] <=
↑ granicaGorna)
4) + liczbaPodokresy <- max(wektorPodokresWybor)
5) + sredniaJednoimiennePodokresy <- tapply(wektorPoziomWybor, wektorPodokresWybor,
↑ mean)
6) + if(tryb == 1) {
7) + sredniaJednoimiennePodokresy * liczbaPodokresy /
↑ sum(sredniaJednoimiennePodokresy) * 100
8) + } else {
9) + if(tryb == 2 & liczbaPodokresy == 12) {
10) + ifelse(wektorPodokresWybor %in% c(1,3,5,7,8,10,12), wektorPoziomWybor * 30/31,
↑ ifelse(wektorPodokresWybor == 2, wektorPoziomWybor * 30/28, wektorPoziomWybor)) ->
↑wektorPoziomWyborStandaryzacja
11) + sredniaJednoimiennePodokresyStandaryzacja <-
↑ tapply(wektorPoziomWyborStandaryzacja, wektorPodokresWybor, mean)
12) + sredniaJednoimiennePodokresyStandaryzacja * liczbaPodokresy /
↑ sum(sredniaJednoimiennePodokresyStandaryzacja) * 100
13) +} else {
14) + "W argumencie tryb wprowadzono nieprawidłową wartość. W wypadku
↑ danych kwartalnych i półrocznych nie należy podawać wartości 2 (brak procedury
↑ standaryzacyjnej)"
15) + }
16) + }
17) + }
18) >
```

```
1) MSJO2 <- function(ramka, granicaDolna = min(ramka[,1]), granicaGorna = max(ramka[,1]),
↑ tryb = 1) {
2) + wektorPodokresWybor <- subset(ramka[,2], ramka[,1] >= granicaDolna & ramka[,1] <=
↑ granicaGorna)
3) + wektorPoziomWybor <- subset(ramka[,3], ramka[,1] >= granicaDolna & ramka[,1] <=
↑ granicaGorna)
4) + liczbaPodokresy <- max(wektorPodokresWybor)
5) + sredniaJednoimiennePodokresy <- tapply(wektorPoziomWybor, wektorPodokresWybor,
↑ mean)
6) + sredniaOgolna <- mean(wektorPoziomWybor)
7) + if(tryb == 1) {
8) + sredniaOgolna * ((sredniaJednoimiennePodokresy * liczbaPodokresy /
↑ sum(sredniaJednoimiennePodokresy)) - 1)
9) + } else {
10) + if(tryb == 2 & liczbaPodokresy == 12) {
11) + ifelse(wektorPodokresWybor %in% c(1,3,5,7,8,10,12), wektorPoziomWybor * 30/31,
↑ ifelse(wektorPodokresWybor == 2, wektorPoziomWybor * 30/28, wektorPoziomWybor)) ->
↑ wektorPoziomWyborStandaryzacja
12) + sredniaJednoimiennePodokresyStandaryzacja <-
↑ tapply(wektorPoziomWyborStandaryzacja, wektorPodokresWybor, mean)
```

```
13) + sredniaOgolnaStandaryzacja <- mean(wektorPoziomWyborStandaryzacja)
14) + sredniaOgolnaStandaryzacja * ((sredniaJednoimiennePodokresyStandaryzacja *
↑ liczbaPodokresy / sum(sredniaJednoimiennePodokresyStandaryzacja)) - 1)
15) + } else {
16) + "W argumencie tryb wprowadzono nieprawidłową wartość. W wypadku
↑ danych kwartalnych i półrocznych nie należy podawać wartości 2 (brak procedury
↑ standaryzacyjnej)"
17) + }
18) + }
19) + }
20) >
```