Wprowadzenie do środowiska programistycznego R

(na podst. http://math.illinoisstate.edu/dhkim/rstuff/rtutor.html)

1) R jako kalkulator

```
> 2 + 3 * 5      # zwróć uwage na kolejność wykonywania działań
> log (10)       # logarytm natualny o podstawie e=2.718282
> 4^2            # 4 do potęgi drugiej
> 3/2            # dzielenie
> sqrt (16)      # pierwiastek kwadratowy
> abs (3-7)      # wartość bezwzględna wyrażenia 3-7
> pi             # liczba pi
> exp(2)         # funkcja wykładnicza
> 15 %/% 4       # liczba całkowita z reszty dzielenia
> # Tworzenie komentarza wykonuje się poprzez wstawienie znaku '#'
```

2) Tworzenie obiektów

Operatorem przypisania w R jest wyrażenie `<-` lub `=`

Wyrażenie znajdujące się po prawej stronie operatora jest przypisane do wyrażenia znajdującego się po lewej stronie. Zapoznajmy się zatem z wybranymi funkcjami matematycznymi dostępnymi w R.

Uwaga: W R wielkość znaków MA ZNACZENIE, tzn. wyrażenie `abc i ABC`, mogą być 2 różnymi obiektami o innej zawartości.

```
> x=log(2.843432)*pi
> x
[1] 3.283001
> sqrt(x)
[1] 1.811905
> floor(x)        # największa liczba całkowita mniejsza niż x
(zaokrąglanie w dół)
[1] 3
> ceiling(x)      # najmniejsza liczba całkowita większa niż x
(zaokrąglanie w górę)
[1] 4
```

3) Typy obiektów

Wektor

Podstawowym typem obiektu jest wektor, który tworzymy z wykorzystaniem funkcji `c`

```
> x<-c(1,3,2,10,5)    #tworzy wektor x o zdefiniowanych 5
komponentach
> x
[1]  1  3  2 10  5
> y<-1:5              #tworzy wektor liczb całkowitych w
kolejności narastającej
> y
[1] 1 2 3 4 5
> y+2                 #dodaje liczbę 2 do wszystkich elementów
```

```
obiektu y
[1] 3 4 5 6 7
> 2*y                  #mnoży x2 wszystkie elementy obiektu y
[1]  2  4  6  8 10
> y^2                  #podnosi każdy element obiektu y do potęgi
2-giej
[1]  1  4  9 16 25
> 2^y                  #podnosi liczbę 2 do potęgi każdego z
kolejnych elementów zbioru y
[1]  2  4  8 16 32
> y                    #sam y nam się nie zmienia
[1] 1 2 3 4 5
> y=y*2
> y                    #chyba, że użyjemy operatora przypisania
[1]  2  4  6  8 10
```

Jeśli chcemy zawrzeć więcej niż 1 polecenie w 1 linii używamy do tego celu średnika:

```
> x<-c(1,3,2,10,5); y<-1:5
```

Wygodnym uproszczeniem są operacje wykonywane bezpośrednio na obiektach:

```
> x+y
[1]  2  5  5 14 10
> x*y
[1]  1  6  6 40 25
> x/y
[1] 1.0000000 1.5000000 0.6666667 2.5000000 1.0000000
> x^y
[1]     1     9     8 10000  3125
```

Podstawą pracy w R są funkcje gdzie najpierw podaje się nazwę funkcji, a następnie w nawiasie elementy jakie mają być przez funkcje wykonane:

```
> sum(x)            #suma wszystkich elementów of elements in x
[1] 21
> cumsum(x)         #skumulowana suma kolejnych elementów  wektora
[1]  1  4  6 16 21
[1] 1 7 3
> max(x)            #maximum
[1] 10
> min(x)            #minimum
[1] 1
```

Sortowanie elementów z użyciem komendy `sort()` :

```
> x
[1]  1  3  2 10  5
> sort(x)               # kolejność rosnąca
[1]  1  2  3  5 10
```

Funkcje mogą mieć domyślnie zadeklarowane parametry lub możemy je określać po przecinku:

```
> sort(x, decreasing=T)  # kolejność malejąca
[1] 10  5  3  2  1
```

Ekstrakcja poszczególnych elementów wektora jest niezmiernie ważna w kontekście pracy z rzeczywistymi danymi:

```
> x
[1]  1  3  2 10  5
> length(x)           # liczba elementów obiektu x
[1] 5
> x[3]                # 3-ci element obiektu x
[1] 2
> x[3:5]              # od 3-go do 5-go elementu obiektu x
[1]  2 10  5
> x[-2]               # wszystkie elementy x z wyjątkiem 2-giego
[1]  1  2 10  5
> x[x>3]              # wyświetla tylko te elementy które
spełniają określoną zależność
[1] 10  5
```

Kolejne przydatne funkcje to `seq(), rep() i which().` Sprawdźmy zatem ich działanie:

```
> seq(10)
 [1]  1  2  3  4  5  6  7  8  9 10
> seq(0,1,length=10)
 [1] 0.0000000 0.1111111 0.2222222 0.3333333 0.4444444 0.5555556
0.6666667
 [8] 0.7777778 0.8888889 1.0000000
> seq(0,1,by=0.1)
 [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
> rep(1,3)
[1] 1 1 1
> c(rep(1,3),rep(2,2),rep(-1,4))
[1]  1  1  1  2  2 -1 -1 -1 -1
> rep("Small",3)
[1] "Small" "Small" "Small"
> c(rep("Small",3),rep("Medium",4))
[1] "Small"  "Small"  "Small"  "Medium" "Medium" "Medium" "Medium"
> which(x>5 & x <7) # podaje numery indeksów spełniających warunek
w nawiasie
```

Macierze

W najprostszej postaci (2-wymiarowej) macierze to po prostu tabele z danymi. Tworzenie macierzy można wykonywać np. poprzez złączenie kolumn lub rzędów o jednakowej długości. W zależności od tego w jaki sposób chcemy macierz utworzyć wykorzystujemy funkcje cbind(), (skrót od "column bind") lub rbind() (skrót od „row bind”):

```
> x
[1]  1  3  2 10  5
> y
[1] 1 2 3 4 5
> m1<-cbind(x,y);m1
      x y
[1,]  1 1
[2,]  3 2
[3,]  2 3
[4,] 10 4
[5,]  5 5
> t(m1)                 # transpozycja macierzy m1
  [,1] [,2] [,3] [,4] [,5]
x    1    3    2   10    5
y    1    2    3    4    5
> m1<-t(cbind(x,y))     # operacje można także zagnieżdżać
> dim(m1)               # podaje wymiary macierzy
[1] 2 5
> m1<-rbind(x,y)        # rbind() jest funkcją dla łączenia po
wierszach. Wynik powinien być taki sam jak dla komendy:
t(cbind()).
```

Oczywiście można macierz utworzyć również z wykorzystaniem funkcji 'matrix()':

```
> m2<-matrix(c(1,3,2,5,-1,2,2,3,9),nrow=3);m2
     [,1] [,2] [,3]
[1,]    1    5    2
[2,]    3   -1    3
[3,]    2    2    9
```

Zwróć uwagę, że poszczególne elementy są użyte do wypełnienia w pierwszej kolejności 1-szej kolumny, potem 2-giej, itd. Jeśli chcemy użyć danych do wypełnienia po rzędach, musimy komputer o tym poinformować za pomocą opcji byrow=T:

```
> m2<-matrix(c(1,3,2,5,-1,2,2,3,9),ncol=3,byrow=T);m2
     [,1] [,2] [,3]
[1,]    1    3    2
[2,]    5   -1    2
[3,]    2    3    9
```

Wyciąganie poszczególnych elementów z macierzy wymaga zadeklarowania jej wszystkich wymiarów:

```
> m2
     [,1] [,2] [,3]
[1,]    1    3    2
[2,]    5   -1    2
[3,]    2    3    9
> m2[2,3]            #element macierzy m2 w 2-gim rzędzie i 3-ciej
kolumnie
[1] 2
```

```
> m2[2,]              #cały 2-gi rząd
[1]  5 -1  2
> m2[,3]              #cała 3-cia kolumna
[1] 2 2 9
> m2[-1,]             #cała macierz m2 bez pierwszego rzędu
     [,1] [,2] [,3]
[1,]    5   -1    2
[2,]    2    3    9
> m2[,-1]             #i bez pierwszej kolumny
     [,1] [,2]
[1,]    3    2
[2,]   -1    2
[3,]    3    9
# stwórz nową macierz o dowolnej nazwie bez pierwszej kolumny i
pierwszego rzędu
```

Operacje arytmetyczne na macierzach wykonuje się analogicznie do operacji na wektorach:

```
> m1<-matrix(1:4, ncol=2); m2<-matrix(c(10,20,30,40),ncol=2)
> 2*m1                 # iloczyn skalarny
     [,1] [,2]
[1,]    2    6
[2,]    4    8
> m1+m2                # dodawanie macierzy
     [,1] [,2]
[1,]   11   33
[2,]   22   44
> m1*m2                # mnożenie analogicznych elementów
     [,1] [,2]
[1,]   10   90
[2,]   40  160
```

Zwróc uwagę, że formuła: m1*m2 NIE JEST mnożeniem macierzy w sensie matematycznym. Do tego służy funkcja %*% :

```
> m1 %*% m2
     [,1] [,2]
[1,]   70  150
[2,]  100  220
```

Ramka danych

Ramka danych jest ostatnim typem danych, który muszą Państwo poznać przed rozpoczęciem prawdziwej pracy w środowisku R.

Ramka danych jest zwykle macierzą składającą się z kolumn, ale może zawierać różne typy danych (np. dane w formie liczbowej, tekstowej, itd.). Nie będziemy się zatrzymywać na kwestii tworzenia ramek danych, tylko wykorzystamy domyślnie wgrane dane o nazwie 'cars'

```
data(cars)

names(cars) # wyświetlenie nazw kolumny
```

Jeśli chcemy wyświetlić tylko pierwszą kolumnę możemy zrobić wywołanie analogiczne jak dla macierzy [] lub użyć symbolu '$'

```
cars$speed
```

Mocną stroną R jest tworzenie wykresów wysokiej jakości, które z powodzeniem mogą być wykorzystane przy publikacjach i opracowaniach. Spróbujmy zatem zastososowania poniższych komend:

```
hist(cars$dist) # tworzy histogram

# zapisz histogram w postaci obiektu 'a' i wyświetl jego zawartość

plot(cars, xlab = "Speed (mph)", ylab = "Stopping distance (ft)",
     las = 1)
lines(lowess(cars$speed, cars$dist, f = 2/3, iter = 3), col =
"red")
title(main = "cars data")
plot(cars, xlab = "Speed (mph)", ylab = "Stopping distance (ft)",
     las = 1, log = "xy")
title(main = "cars data (logarithmic scales)")
lines(lowess(cars$speed, cars$dist, f = 2/3, iter = 3), col =
"red")
summary(fm1 <- lm(log(dist) ~ log(speed), data = cars))

# Możemy także spróbować zrobić prostą wizualizację 3d dla modelu DEM

data(volcano)
z <- 2 * volcano        # Exaggerate the relief
x <- 10 * (1:nrow(z))   # 10 meter spacing (S to N)
y <- 10 * (1:ncol(z))   # 10 meter spacing (E to W)
## Don't draw the grid lines :  border = NA
persp(x, y, z, theta = 135, phi = 30, col = "green3", scale =
FALSE, ltheta = -120, shade = 0.75, border = NA, box = FALSE)
```

Programowanie w R

Dotychczas nasza nauka nie wykraczała zbytnio poza możliwości, które możemy osiągnać w dowolnym arkuszu kalkulacyjnym. Prawdziwa moc zaklęta w językach programowania polega na zastosowaniu instrukcji sterujących, pętli oraz funkcji programistycznych, które pozwalają na automatyzację wykonywanych operacji.

Instrukcje sterujące/warunkowe:

```
#Instrukcja warunkowa 'if':
> if (warunek){
+    instrukcja1
+    }
> else{
+    instrukcja2
+    }
```

można także zastosować instrukcję warunkową 'if' bez opcji 'else', np.:

```
> x = 2

> if(x==2){
    print("Faktycznie, x=2")
    }

> y=4
> if(x==3 & y==4){
    print("Faktycznie, x=3 i y=4")
    }
    else{
    print("x jest różny od 3, y jest różny od 4")
    }
```

Zadanie:
Napisz instrukcję warunkową „if-else” sprawdzających wybrane dowolnie 3 poniższe wyrażenia logiczne (dla dowolnie zdefiniowanych liczb):

`x == y` "x is equal to y"

`x != y` "x is not equal to y"

`x > y` "x is greater than y"

`x < y` "x is less than y"

`x <= y` "x is less than or equal to y"

`x >= y` "x is greater than or equal to y"

Drugą instrukcją sterującą, podobną do 'if' jest instrukcja **'ifelse':**

```
# Instrukcja 'ifelse' (odpowiednik excelowego 'jeżeli', ale możemy go tworzyć dla obiektów dowolnie długich, a nie dla jednej komórki)
```

```
> x = 1:10
> ifelse(x<5 | x>8, x, 0)
```

PĘTLE:

1. FOR

Najpopularniejsza pętla w językach programowanie to tzw. pętla 'for'. Wszystko co znajduje się pomiędzy znakami '{}' jest wykonywane dla każdej zdefiniowanej wartości dowolnego parametru , taką ilość razy jaka jest długość prawej strony wyrażenia:

```
# Pętla for:
> for (k in 1:5){
+ print(k)
+ }
```

Najlepiej jednak pokazać użyteczność na przykładzie (np. posługując się zbiorem danych z reanalizy temperatury powierzchni morza wg Reynoldsa (2007):

```
load("sst.Rdata") # wczytujemy przykładowe dane


for (dzien in (1:10)) {
  jpeg(filename=paste(czas[dzien],"jpg",sep="."))
  image(lon,lat,sst[,,dzien],main=czas[dzien],xlab="lon",ylab="lat")
  box()
  dev.off()
}
```

2. WHILE

W modelowaniu często nie znamy dokładnego rozwiązania wielu układów równań różniczkowych. Często nie mają one analitycznego rozwiązania. Stąd też wiele metod wykorzystuje tzw. metody iteracyjne, gdzie obliczenia wykonywane są aż do uzyskania pewnej satysfakcjonującej nas wielkości minimalnego błędu. Tego typu zależność będziemy wykorzystywać w jednym z najprostszych modeli bilansu energetycznego Ziemi. Spróbujmy zatem wykorzystać inną właściwość pętli 'while', która wykonuje się tak długo dopóki zadeklarowany warunek nie zostanie spełniony. Wyobraźmy sobie, że mamy dużą ilość plików, które mają analogiczny ciąg serii danych (rok, miesiąc, dzień, godzina) i chcemy je połączyć w jedną tabelę:

```
setwd(„C:\modelowanie\pliki ”) # ustawiamy katalog gdzie przechowujemy te pliki:
plik=dir(pattern=".txt")
 # pobieramy nazwy plików, które znajdują się w katalogu
dl=length(plik)
   # sprawdzamy ile jest plików w tym katalogu
wynik=NULL    # tworzymy pusty obiekt na wyniki
i=1
```

```
while(i<=dl) {
  dane=read.table(plik[i], header=T)
  wynik=cbind(wynik,dane$ff)
  i=i+1
}
colnames(wynik)=plik
wynik=cbind(dane[,1:4],wynik)
write.table(wynik, file="wynik.csv", sep=";", row.names=F)
```

FUNKCJE

Ostatnią rzeczą, którą każdy początkujący programista musi poznać są funkcje. Wszystkie polecenia, po których do tej pory wywoływaliśmy () są w rzeczywistości właśnie funkcjami, przyjmującej pewne parametry. Jeśli wywołamy jakieś polecenie będące funkcją bez nawiasów, wówczas wyświetli nam się jej kod źródłowy.

```
> log
```

Ogólna postać funkcji wygląda następująco:

```
nazwafunkcji = function(argument1, argument2, ...) {
        wyrażenia zawarte w funkcji
} # koniec funkcji
```

następnie wywołujemy funkcję wpisując jej nazwę i argumenty. Sprawdźmy jak to działa na konkretnym przykładzie:

```
funkcja = function(r,x) {
        r*x*(1-x)
}
```

i przetestujmy jej działania na wartościach r=2, x=5 :

```
funkcja(2,5)
```

Jeśli korzystamy z jakiegoś schematu postępowania bardzo często warto zawrzeć je właśnie w postaci funkcji.