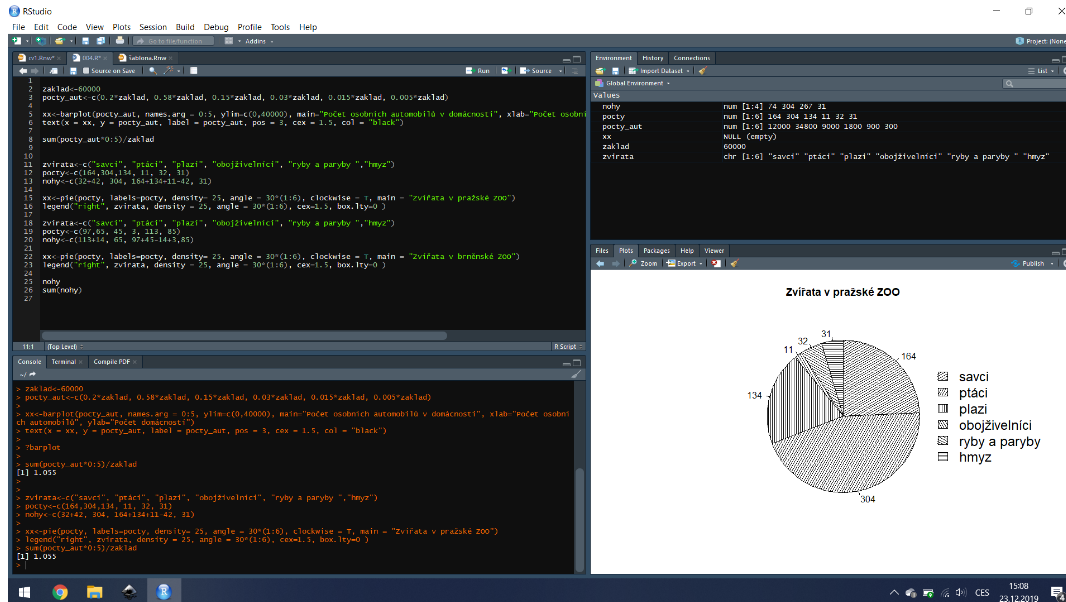
1 RStudio

R je programovací jazyk, zatímco RStudio je prostředí, které budeme používat. RStudio vypadá následovně:



Prostředí RStudia je rozděleno na 4 části:

- **vlevo nahoře** jsou skripty – zde se píše kód do souboru (obvykle `nazev.R`), který pak můžeme spouštět. Buď řádek po řádku (zkratka Ctrl+Enter), nebo označením části kódu myší a opět stejnou zkratkou.
- **vlevo dole** je konzole – zde můžete psát kód a rovnou jej pomocí Enter spustit. Hodí se zejména pro ryché výpočty a hledání chyb. Nevýhoda je, že se neukládá, narozdíl od skriptů. V této části také uvidíte výstupy ze svých programů a také chybové nebo varovné hlášky.
- **vpravo nahoře** se necházejí pokročilé věci. Pro nás je zajímavá hlavně karta Global Enviroment, kde vidíte objekty, které existují v paměti. Na Obrázku například vidíte, že existuje objekt `nohy`, který je numerický vektor o 4 prvcích, konkrétně 74, 304, 267 a 31.
- **vpravo dole** je několik karet. Files slouží pro navigaci v souborech v počítači. Plots ukazuje výsledky vykreslování různých grafů. Packages nainstalované balíčky. Dále budeme často používat Help, který obsahuje nápovědu k různým příkazům.

Vaše RStudio pravděpodobně nevypadá úplně stejně jako to na obrázku. Pomocí Tools › Global Options si můžete vzhled RStudia přizpůsobit. Zejména doporučuji přepnou v Appearance theme na nějaké tmavé, které méně unavuje oči.

2 Working directory, balíčky, nápověda a komentáře

2.1 Working directory (pracovní adresář)

Před tím, než začneme psát první program v R, je dobré vytvořit si složku, ve které budete své soubory (Skripty, výsledné grafy a jiné soubory) uchovávat.
Vytvořte si tedy novou složku (např. SMLF) někde v dokumentech svého počítače. Poté vpravo dole, v kartě files se do této složky proklikjte a když v ní budete, pomocí ozubeného kolečka (je u nej nápis More › Set as working directory. Na konzoli se objeví něco jako

```r
setwd("~/práce/SMLF")
```

Pokud nevíte, v jakém adresáři jste (který je nastavený jako pracovní), můžete se zeptat

```r
getwd()

## [1] "C:/Users/janbo/Documents/práce/SMLF"
```

Cesta u vás bude jiná. Tímto příkazem můžete nastavit pracovní složku i bez pracného proklikávání, tento příkaz bude na začátku každého skriptu, který budete psát.

2.2 Instalace a spouštění balíčků

Při programování není důležité pokaždé znova vynalézat kolo. Při programování v R máte přístup ke spoustě externích balíčků. Jedním z užitečných balíčků je např. `xlsx`, který umožňuje nařítat excelová data. Instalaci tohoto balíču provedete příkazem

```r
install.packages("readxl")
```

Nezpoměňte na uvozovky! Po nainstalování balíčku jej stačí zprovoznit zavoláním

```r
library("readxl")

## Warning: package 'readxl' was built under R version 3.5.3
```

Je dobré všechny volané balíčky mít hned na začátku skriptu.

2.3 Nápověda

Nápovědu k dané funkci zobrazíte pomocí `?nazev_funkce`. Pokud například nevíte, jak funguje funkce `sum`, můžete se zeptat pomocí

```r
?sum
```

Nápověda se zobrazí vpravo dole, kde si přečtete, co funkce dělá, jakou má syntaxi, argumenty a to nejužitečnější – příklady použití.

2.4 Komentáře

Komentáře jsou část vašeho kódu, který se nevyhodnocuje – slouží pro vaše poznámky a přehlednost. Cokoliv, co je ve skriptu za symbolem `#` se nevyhodnocuje jako příkaz, slouží jen pro přehlednost. Například

```r
# Ukázka komentáře. Tento text nic nedělá, je na řádku za #
sum(1:5) #Sečteme čísla 1 až 5. Kód před # se vyhodnotí:

## [1] 15

# Rko nám řeklo, kolik to je.
```

Další časté využití komentářů je při hledání chyb – není nutné odmazat kód, stačí jej zakomentovat (pokud nevím, zda se k němu budu chtít zase vrátit).

3 Proměnné

V R můžeme do proměnných přiřazovat hodnoty a dále s nimi pracovat.

```
jmeno <- "Jan" #pomocí <- do proměnné jmeno uložíme hodnotu Honza.
               #U textových řetězců je třeba použít uvozovky.
prijimeni = "Novak" # Rovnitko funguje taky,
                    # ale doporučuji používat raději šipku <-
vek <- 24 #Stejně tak můžeme přiřazovat číselné hodnoty
# V konzoli si můžete vypsat hodnotu v proměnné jejím zavoláním
jmeno

## [1] "Jan"

# Ve skriptech se používá print(), který vypíše hodnotu svého argumentu.
print(jmeno)

## [1] "Jan"
```

3.1 Datové typy

Rko rozlišuje 5 datových typů – datový typ proměnné rozhoduje o tom, co lze s ním dělat. Tyto typy jsou:

- **logical** pouze pravda/nepravda, tj. `TRUE` a `FALSE`, zkráceně `T` a `F`
- **numeric** reálná čísla. Dále se dělí na
  - **integer** celé číslo, např. `8L` značí 8
  - **double** reálné číslo, např. `8.5`. Obsahuje desetinnou tečku
- **complex** komplexní čísla, např. `1 + 2i` , ty potřebovat nebudeme
- **character** textové řetězce, např. `"SMLF"` nebo `"8"`. Cokoliv v uvozovkách.
- **raw** binární data, s těmi se nesetkáme

Datový typ rozhoduje o tom, co s daným objektem můžeme dělat. Čísla lze sčítat, násobit, mocnit. . . , což např. s textovými řetězci nelze dělat.
Může se stát, že potřebujeme zjistit, jakého typu je daný objekt. K tomu slouží `class`.

```
x <- 8
class(x)

## [1] "numeric"

# Aha, v x je uloženo reálné číslo.
y <- "8"
class(y)

## [1] "character"

# Použil jsem uvozovky, takže v y je textový řetězec.
x+y

## Error in x + y: non-numeric argument to binary operator
```

```r
# Chyba, číslo s textem nelze sečíst.

# Pokud mám hypotézu, jakého typu je nějaká proměnná, mohu se zeptat přímo:
is.numeric(y)

## [1] FALSE

# Obdobně funguje is.integer(), is.character() apod.
```

Taky se může hodit převádět text na číslo a naopak:

```r
x <- "123"
class(x)

## [1] "character"

# V x je textový řetězc
x <- as.numeric(x)
print(x)

## [1] 123

class(x)

## [1] "numeric"

# Nyní je z x číslo. A zase zpátky na řetězec
x <- as.character(x)
print(x) # Uvozovky

## [1] "123"

class(x)

## [1] "character"
```

3.2 Speciální hodnoty

Zejména v chybových hláškách se můžeme setkat s následujícími speciálními hodnotami:

- `NA` Not Availible (chybějící hodnota), obvykle ji potkáte, když chybí hodnoty v načteném datovém souboru.
- `NaN` Not a Number je obvykle výsledkem nesmyslné matematické operace
- `NULL` používáme, když chceme vytvořit objekt, už víme, jak se jmenuje, ale nevíme, co do něj uložit.

```r
log(-1) #Nelze logaritmovat záporné číslo

## Warning in log(-1): NaNs produced

## [1] NaN

1/0 # Dělení nulou v R produkuje nekonečno

## [1] Inf
```

4 Vektory

4.1 Inicializace a prodloužení vektoru

Většinou si nevystačíme s proměnnými, které obsahují jen jednu hodnotu, ale budeme potřebovat vektory. Klíčovou vlastností vektoru je, že všechny hodnoty v něm musí mít stejný typ. Vektory vytvoříme pomocí `c()`, kde v závorce jsou jednotlivé prvky odděleny čárkou.

```
logic_vektor <- c(TRUE, TRUE, FALSE)
class(logic_vektor)

## [1] "logical"

num_vektor <- c(10, pi, 7-4)
class(num_vektor)

## [1] "numeric"

char_vektor <- c("Aleš", "Bětka", "Cyril")
class(char_vektor)

## [1] "character"

smes_vektor <- c( 5, TRUE, "Dan") # Tohle by nemělo existovat
class(smes_vektor) # Podle všeho jsou všechny hodnoty char

## [1] "character"

print(smes_vektor) # Ejhle, Rko vše převedlo na textové řetězce

## [1] "5"    "TRUE" "Dan"
```

Pokud chceme vektor prodloužit (přidat k němu další hodnotu nebo spojit dva vektory, stačí použít znova `c()` – jako zřetězení). Opět musí sedět datové typy.

```
vektor_A <- c( "Aleš", "Bětka")
vektor_A <- c(vektor_A, "Cecilka") # Prodloužíme vektor_A o Cecilku
print(vektor_A) # povedlo se

## [1] "Aleš"    "Bětka"   "Cecilka"

vektor_B <- c("Xaver", "Ýgar", "Zora")
spojeni <- c(vektor_A, vektor_B) # Spojení vektorů do jednoho v daném pořadí.
print(spojeni)

## [1] "Aleš"    "Bětka"   "Cecilka" "Xaver"   "Ýgar"    "Zora"
```

Často budeme chtít vytvořit vektor celočíselných hodnot. Toho dosáhneme pomocí `from : to`.

```
1 : 10 # Celočíselná rostoucí posloupnost.

##  [1]  1  2  3  4  5  6  7  8  9 10

-3 : 3 # Může být i klesající.

## [1] -3 -2 -1  0  1  2  3
```

Další často používaná situace je vytvoření posloupnosti s krokem jiným než jedna `seq(from =, to =, by =)`, případně s celkovým počtem prvků `seq(from =, to =, length.out =)`

```
seq( from = 0, to = 1, by = 0.25) # by určuje velikost kroku

## [1] 0.00 0.25 0.50 0.75 1.00

seq( from = 0, to =1, length.out = 6) # length.out celkový počt prvků

## [1] 0.0 0.2 0.4 0.6 0.8 1.0
```

Poslední možností, jak inicializovat vektor je opakování stejné hodnot `rep(co, kolikrat)`.

```
rep(0, 10) # vektor 10 nul

##  [1] 0 0 0 0 0 0 0 0 0 0

rep(NULL, 10) # vektor 10 null

## NULL

# Používá se, když konkrétními hodnotami jej budeme teprve plnit,
# ale víme, že jich bude 10
```

4.2 Manipulace s hodnotami vektoru

Pokud se chceme odkázat na konkrétní hodnotu (hodnoty) vektoru, použijeme `vektor[index]`.

```
vektor<-letters # Vektor písmen anglické abecedy
vektor[3] # hodnota na 3. místě vektoru

## [1] "c"

vektor[c(1,9)] # hodnota na 1. a 9. místě

## [1] "a" "i"

vektor[10:5] # hodnoty na 10. až 5. místě

## [1] "j" "i" "h" "g" "f" "e"

vektor[30] # 30. hodnota vektoru

## [1] NA

# Neexistuje
vektor[2]<-"z" #Přepsání hodnoty na 2. pozici
```

Výběr lze provést i pomocí logického vektoru

```
vektor<-letters[1:5] # Vektor prvních 5 písmen anglické abecedy
vyber<-c(T,F,F,F,T) # Loický vektor délky 5
vektor[vyber] # Vybere hodnoty na místech, kde je T (TRUE)

## [1] "a" "e"
```

Často chceme zjistit délku vektoru pomocí funcke `length()`.

```
length(letters) # Kolik písmen má anglická abeceda?

## [1] 26

length(seq(0, 1000, by=0.001)) # Kolik prvků má tato posloupnost?

## [1] 1000001
```

Pokud chceme vypsat z `vektor`u prvních `n` hodnot, použijeme `head(vektor, n)`. Pro posledních `n` hodnot funguje obdobně `tail(vektor, n)`.

```
vektor <- seq(0, 10, by = 1/3 )
head(vektor, 5) # Prvních 5 hodnot

## [1] 0.0000000 0.3333333 0.6666667 1.0000000 1.3333333

tail(vektor, 1) # Poslední hodnota

## [1] 10
```

4.3 Matematické operace s vektory

Máme-li dva vektory numerického typu, můžeme s nimi provádět numerické operace. Pokud je jeden z vektorů kratší, zřetězí se, dokud není dost dlouhý.

```
a<-c(2,5,0,-4, 1)
b<-0:4

a+b # součet

## [1]  2  6  2 -1  5

a-b # rozdíl

## [1]  2  4 -2 -7 -3

a*b # součin

## [1]   0   5   0 -12   4

a/b # podíl

## [1]       Inf  5.000000  0.000000 -1.333333  0.250000

a%/%b # celočíselnné dělení (dělení bez zbytku)

## [1] Inf   5   0  -2   0

a %% b # modulo, tj. zbytek po dělení

## [1] NaN   0   0   2   1

a**b # Umocnění, buď pomocí ** nebo symbolem ^

## [1]   1   5   0 -64   1
```

```r
# Pokud je jeden vektor kratší, prodlouží se
c(1, 2 ,3) * c(-1, 1)

## Warning in c(1, 2, 3) * c(-1, 1): longer object length is not a multiple of shorter
object length

## [1] -1  2 -3

# Konstanta je taky vektor

2*c(1,2,3)

## [1] 2 4 6

# Na vektory můžeme aplikovat i matematické funkce

sin(c(pi, pi/2, 0, pi/6))

## [1] 1.224606e-16 1.000000e+00 0.000000e+00 5.000000e-01
```

4.4 Statistické výpočty s vektory

V Rku můžeme jednoduše spočítat průměr, směrodatnou odchylku apod.

```r
vektor<-c(2,4,1,2,2,0,1,9)
sum(vektor) # Součet

## [1] 21

mean(vektor) # Průměr

## [1] 2.625

var(vektor) # Rozptyl

## [1] 7.982143

sd(vektor) # Směrodatná odchylka

## [1] 2.825269

median(vektor) # Medián

## [1] 2

table(vektor) # Tabulka hodnot např. pro určení modu nebo konstrukci histogramu

## vektor
## 0 1 2 4 9
## 1 2 3 1 1
```

5 Matice

5.1 Inicializace a prodloužení matice

Matici vytvoříme tak, že vektor seskládáme do matice příkazem `matrix(vektor, nrow = , ncol =, byrow = T)`. Parametr `byrow` kontroluje, zda budeme plnit matici po řádcích či po sloupcích.

```
vektor <- 1:15
matrix(vektor, nrow = 3, ncol = 5, byrow = T) # Naskládáno po řádcích.

##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    2    3    4    5
## [2,]    6    7    8    9   10
## [3,]   11   12   13   14   15

matrix(vektor, nrow = 3, ncol = 5, byrow = F) # Naskládáno po sloupcích.

##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    4    7   10   13
## [2,]    2    5    8   11   14
## [3,]    3    6    9   12   15

# Co když vektor není dost dlouhý?
matrix(vektor, nrow = 5, ncol = 5) # byrow = F je default

##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    6   11    1    6
## [2,]    2    7   12    2    7
## [3,]    3    8   13    3    8
## [4,]    4    9   14    4    9
## [5,]    5   10   15    5   10

# Vektor se zopakuje, toho lze využít na vytvoření matice stejných hodnot, např:
matrix(0, nrow = 2, ncol = 3)

##      [,1] [,2] [,3]
## [1,]    0    0    0
## [2,]    0    0    0
```

Speciální je jednotková matice `diag(n)`.

```
diag(2)

##      [,1] [,2]
## [1,]    1    0
## [2,]    0    1
```

Můžeme chtít slepit dvě matice – a to buď po řádcích příkazem `rbind()` nebo po sloupcích – `cbind()`. Musí však sedět dimenze.

```
maticeA <- matrix("A", nrow = 4, ncol = 3)
maticeB <- matrix("B", nrow = 2, ncol = 3)
# Spojíme matice "pod sebe"
rbind(maticeA, maticeB)

##      [,1] [,2] [,3]
```

```
## [1,] "A"  "A"  "A"
## [2,] "A"  "A"  "A"
## [3,] "A"  "A"  "A"
## [4,] "A"  "A"  "A"
## [5,] "B"  "B"  "B"
## [6,] "B"  "B"  "B"

# Spojení vedle sebe nebude fungovat
cbind(maticeA, maticeB)

## Error in cbind(maticeA, maticeB): number of rows of matrices must match (see arg
2)
```

Nejčastější použití je pro připojení dalšího řádku nebo sloupce (můžeme připojit vektor k matici).

```
maticeA <- matrix("A", nrow = 3, ncol = 3)
vektorB <- rep("B", 3)
# Přidání řádku
rbind(maticeA, vektorB)

##         [,1] [,2] [,3]
##         "A"  "A"  "A"
##         "A"  "A"  "A"
##         "A"  "A"  "A"
## vektorB "B"  "B"  "B"

# Přidání sloupce
cbind(maticeA, vektorB)

##                  vektorB
## [1,] "A" "A" "A" "B"
## [2,] "A" "A" "A" "B"
## [3,] "A" "A" "A" "B"
```

5.2 Manipulace s hodnotami matice

Pokud se chceme odkázat na konkrétní hodnotu (hodnoty) vektoru, použijeme `matice[r,s]`, kde `r` je řádek a `s` je sloupec.

```
# Vytvoříme matici prvních 15 písmen abecedy
matice <- matrix(letters[1:15], nrow = 3, ncol =5 , byrow = T)
print(matice)

##      [,1] [,2] [,3] [,4] [,5]
## [1,] "a"  "b"  "c"  "d"  "e"
## [2,] "f"  "g"  "h"  "i"  "j"
## [3,] "k"  "l"  "m"  "n"  "o"

matice[2,3] # Co je na 2. řádku a 3. sloupci

## [1] "h"

matice[1, ] # Pokud necháme prázdné místo u sloupce, vybereme 1. řádek

## [1] "a" "b" "c" "d" "e"
```

```
matice[ ,4] # Celý 4. sloupec

## [1] "d" "i" "n"

matice[2, 2:4] # 2-4 hodnota na 2. řádku

## [1] "g" "h" "i"

matice[4,1]

## Error in matice[4, 1]: subscript out of bounds

# Neexistuje
matice[2,3]<-"z" #Přepsání hodnoty
print(matice)

##      [,1] [,2] [,3] [,4] [,5]
## [1,] "a"  "b"  "c"  "d"  "e"
## [2,] "f"  "g"  "z"  "i"  "j"
## [3,] "k"  "l"  "m"  "n"  "o"
```

Často nás u matice bude zajímat její rozměr – dimenze:

```
# Vytvoříme matici prvních 15 písmen abecedy
matice <- matrix(letters[1:15], nrow = 3, ncol =5, byrow = T)
dim(matice) #Vrátí vektor c(počet řádků, počet sloupců)

## [1] 3 5
```

5.3 Matematické operace matice

Standardní operace `+,-, *, /` apod. jsou definovány po prvcích (pro stejně velké matice). Samozřejmě existuje i maticový součin `%*%` a další operace, ale o nich, až když je budeme potřebovat.

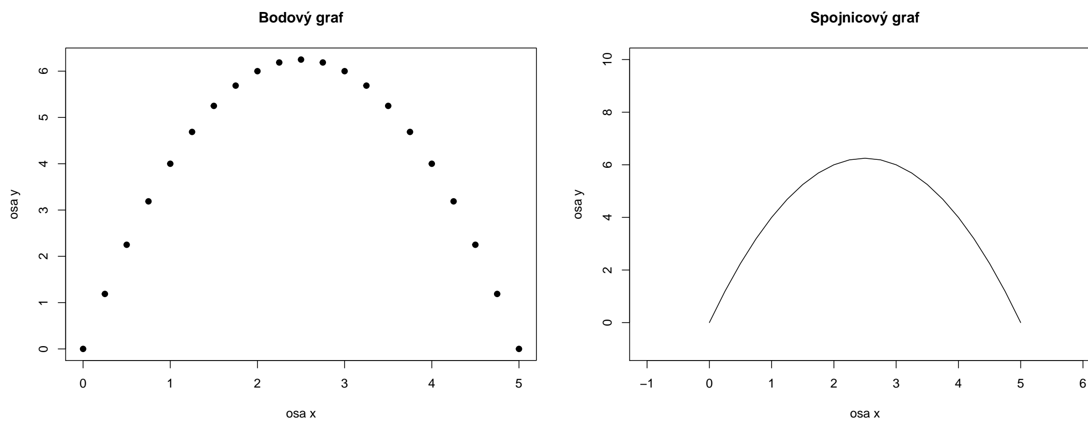
6 Grafy

Základním grafem je bodový graf (scatterplot), který lze vhodně modifikovat. Příkaz je

```
plot(x, y, type = "p", xlim = c(0,1), ylim = c(0,1),
    xlab = "osa x", ylab = "osa y", main = "Název grafu)
```

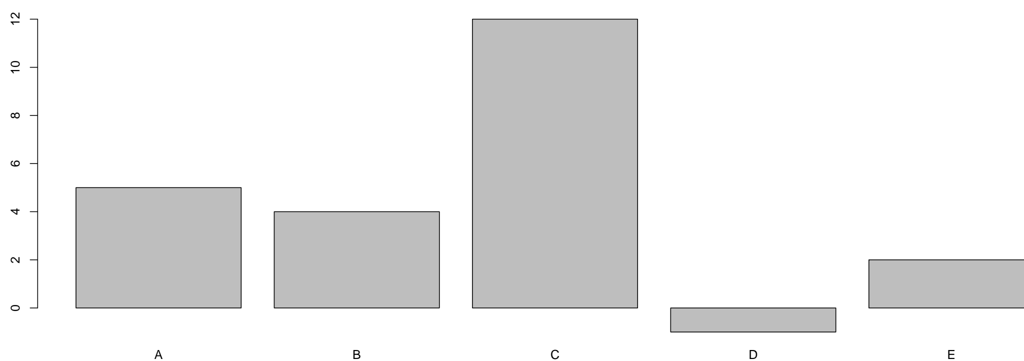
1. `x` je vektor x-ových souřadnic bodů
2. `y` je vektor y-ových souřadnic bodů
3. `type="p"` typ grafu. Možnost `"p"` jsou points (bodový graf), `"l"` značí lines (spojnicový graf) atd. všechny možnosti najdete v nápovědě.
4. `xlim=c(0,1)` a `ylim=c(0,1)` rozsah grafu na ose x a na ose y
5. `xlab="osa x"` popis osy x, podobně funguje popis osy y a název celého graf (`main=""`)

```r
par(mfrow=c(1,2)) # Grafy budou vykresleny v rastru 2x1
x<-seq(0,5, by = 0.25)
y<- -x**2 + 5* x
plot(x, y, type = "p", xlab = "osa x", ylab= "osa y",
     main = "Bodový graf", pch = 19) # pch nastavuje tvar bodů, viz help
plot(x, y, type = "l", xlab = "osa x", ylab= "osa y",
     main = "Spojnicový graf", xlim = c(-1, 6), ylim = c(-1, 10))
```



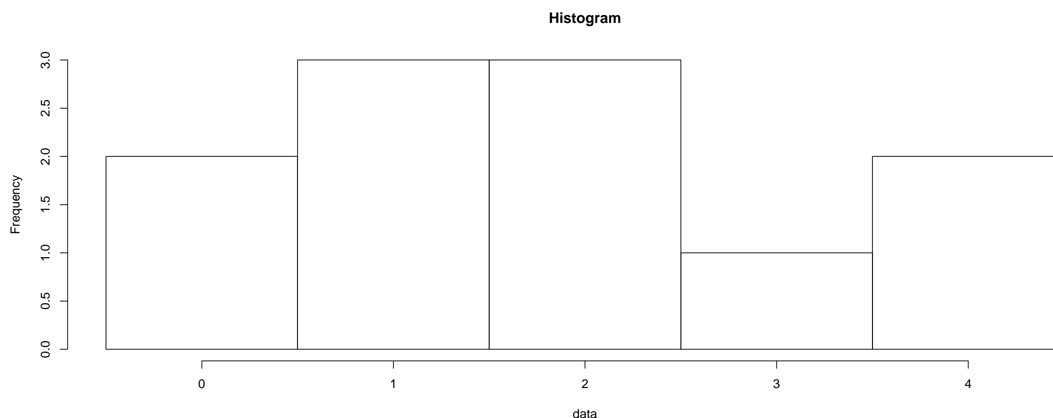
Další často používané grafy jsou `barplot`, který udělá sloupcový graf. Opět, podrobnější nastavení v nápovědě.

```r
par(mfrow=c(1,1)) # Pouze jeden obrázek
vysky<-c(5, 4, 12, -1, 2)
barplot(vysky,  names.arg = c("A", "B", "C", "D", "E"))
```



Koláčový graf používat nebudeme a nedělejte to. Ještě by nás mohl zajímat histogram. Opět, spousta nastavení je v nápovědě `?hist`.

```r
par(mfrow=c(1,1)) # Pouze jeden obrázek
data <- c(1, 0 ,2 ,2 ,2 , 3, 4, 1, 1, 0, 4 )
hist(data, breaks = seq(-0.5, 4.5, by = 1), main = "Histogram")
```



7 Podmínky

Podmínka v jazyce R má synatax

```
if(podminka){
  vykona_se_pokud_je_podminka_splnena
} else {
  vykona_se_pokud_neni_podminka_splnena
}
```

Podmínka je nějaký výraz, jehož hodnota je logická – `TRUE` nebo `FALSE`. Obvykle se setkáme s rovností `==`, případně nějakou nerovností `>, <, <=, >=`. Nerovná se (negaci) provdeme pomocí `!=`. Samozřejmě, že `<>` mají smysl jen pro numerické hodnoty, `==` můžeme porovnávat i textové řetězce.
Část `else {}` je nepovinná. Konkrétním příkladem může být rozhodnutí, zda-li je `n` kladné, záporné nebo nula.

```
n <- 5
if(n == 0){ # Test, zda je n=0 se provede pomocí ==
  print("n je 0")
} else { # Pokud není, vyhodnotí se tato část
  if( n > 0 ){ # Pokud je n > 0
    print("n je kladné") # Vypíše se text
  } else { # Jinak
    print("n je záporné") # Se vypíše toto
  }
}

## [1] "n je kladné"

# Ukázalo se, že 5 je kladné číslo
```

Pokud použijeme porovnání na vektory, je výsledkem vektor logických hodnot. Pokud chceme z tohoto vektoru udělat jedinou hodnotu, můžeme použít `any()`, který vrací `TRUE`, pokud aspoň jeden vstup je pravdivý, nebo `all()`, který vrací `TRUE`, pokud jsou všechny hodnoty pravdivé.

```
x <- 1:10
x > 4 # Které hodnoty x jsou větší než 4?
```

```
##  [1] FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE

# Toto nebude v podmínce fungovat. Potřebujeme jen jednu hodnotu T/F
any(x > 4) # Je v x aspoň jedna hodnota > 4 ?

## [1] TRUE

all(x > 4) # Jsou v x všechny hodnoty > 4 ?

## [1] FALSE
```

V podmínce můžeme využívat taky výrazy „nebo“ (`|`), případně „a současně“ (`&`).

```
a <- 2
b <- 3
if( a > 2 | b > 2){
  print("ANO")
}else{
  print("NE")
}

## [1] "ANO"

if(a > 2 & b > 2){
  print("ANO")
}else{
  print("NE")
}

## [1] "NE"
```

8 Cykly `for` a `while`

V Rku jsou k dispozici 2 cykly: `for` a `while`. Obě způsobují opakování série nějakých příkazů, liší se tím, že u cyklu `for` dopředu víme, kolikrát proběhne.
Syntax cyklu `for` je

```
for(prvek in vektor){ # postupně projdi hodnoty vektoru
  udelej_neco # tohle s nimi udělej
}
```

Konkrétní příklady:

```
# Sečteme čísla 1 až 10
soucet <- 0 # Na zacatku je soucet 0
for (cislo in 1:10){ # Projdeme hodnoty 1-10
  soucet <- soucet + cislo} # K součtu přičteme další číslo
print(soucet)

## [1] 55
```

```
# Sečteme čísla 1 až 10
jmena <- c("Aleš", "Cyril", "Luboš", "Xena") #
for(jmeno in jmena){ # Postupně projdeme jmena
 print(jmeno) # A vypíšeme je
}

## [1] "Aleš"
## [1] "Cyril"
## [1] "Luboš"
## [1] "Xena"
```

Zde jsme využívali cyklus `for`, protože jsme věděli, že potřebujeme projít všechny hodnoty z nějaké množiny (vektoru). Cyklus `while` naopak využíváme, pokud nevíme, kolik iterací bude třeba. Syntax pro `while` je:

```
while(podminka){ # Pokud podminka plati
  udelej_neco # Udelej neco
}
```

Konkrétně:

```
x <- 1:1000 # Ve vektoru x jsou čísla 1-1000
soucet <-0 # Jejich soucet je na začátku nula
index <-1 # Začneme přičítáním prvního prvku vektoru
while(soucet < 1000){ # dokud je soucet menší než tisíc
  soucet <- soucet + x[index] # součet zvedneme o další prvek
  index <- index + 1 # a posuneme index o 1
}
# Na konec, když už máme víc jak 1000, si vypíšeme kolik ten součet je
# a kolik členů vektoru x jsme museli sečíst
print(c(soucet, index - 1)) #Proč je zde index - 1 ?

## [1] 1035   45
```

Další využití cyklu `while` je konstrukce `while(TRUE)`. Tento cyklus bude fungovat navždy, dokud nedostane příkaz konec – `break`.

```
x <- 1:1000 # Ve vektoru x jsou čísla 1-1000
soucet <-0 # Jejich soucet je na začátku nula
index <-1 # Začneme přičítáním prvního prvku vektoru
while(TRUE){ # takovýto program poběží donekonečna, pokud nedostane příkaz break
  soucet <- soucet + x[index] # součet zvedneme o další prvek
  if(soucet > 1000){ # podmínku implementujeme zde
    print(c(soucet, index)) # Když je splněna, vypíše výsledek
    break # a cyklus ukončí
  }
  index <- index + 1 # Jinak posuneme index o 1 a opakujeme
}

## [1] 1035   45

# Na konec, když už máme víc jak 1000, se vypíšem kolik ten součet je
# a kolik členů vektoru x jsme museli sečíst
```

9 Funkce

V Rku často voláme funkce. Např.

```
sum(1:10) # Funkce sum() sečte všechny hodnoty (vektor) svého argumentu

## [1] 55

sin(pi/2) # Funkce sin() funguje jako v matematice

## [1] 1

as.integer("5") # as.integer() převede textový řetězec "5" na číslo 5

## [1] 5

# Funkce může mít více argumentů (vstupů) a vracet vektor nebo matici
seq(from = -1, to =1, by =0.5)

## [1] -1.0 -0.5  0.0  0.5  1.0

# Nebo ohlásit chybu případně warning.
log(-1)

## Warning in log(-1): NaNs produced

## [1] NaN

# Zejména warningy jsou nebezpečné, protože váš kód "funguje"
# Ale nemusí fungovat tak, jak zamýšlíte...
```

Pokud si s nějakou funkcí nevíte rady, stačí zavolat funkci nápovědy k této funkci.

9.1 Vlastní funkce

Pokud se stane, že bychom nějaký kus kódu chtěli používat vícekrát, vyplatí se vytvořit funkci, kterou pak budeme opakovaně volat.
Potřebujeme vytvořit funkci, která má na vstupu vektor celých čísel a vrátí nám součet pouze sudých hodnot. Základní kostra funkce vypadá takto:

```
secti_sude<-function(vektor){ # Funkce má jméno secti_sude a jeden vstup - vektor
  prikazy # Serie příkazů, jak ze vstupu vytvořit výsledek
  return(vysledek) # Co má funkce vrátit
}
```

Stačí vymyslet posloupnost příkazů, jak ze vstupu – vektoru čísel vytvořit výsledek (součet sudých čísel). Jeden z možných postupů je

1. Projít jednotlivé hodnoty vektoru.
2. U každé hodnoty zjistit, zda je sudá.
3. Pokud ano, přičíst ji k výsledku, jinak nedělat nic.
4. Nakonec vrátit výsledek.

Náročný úkol vytvoření celé funkce jsme takto rozbili na menší podúkoly, které vyřešíme samostatně.
1. Chceme projít jednotlivé hodnoty vektoru vstupu. To dokážeme cyklem `for`

```
for(hodnota in vektor){
  co_s_ni_dal?
}
```

2. Cyklus, který projde hodnoty vektoru máme nachystaný, dále budeme testovat sudost. To můžeme udělat např. tak, že se podíváme na zbytek po dělení 2 (operace `%%`).

```
if( (hodnota %% 2) == 0){ # Test rovnosti pomocí ==
    co_s_tou_sudou_hodnotou?
} else {
    co_s_lichou_hodnotou?
  }
```

3. Pokud je hodnota sudá, tak ji přičteme k výsledku, jinak neděláme nic.

```
vysledek <- vysledek + hodnota
```

Nyní stačí dát celý náš kód dohromady:

```
secti_sude<-function(vektor){
  vysledek <- 0 # Na začátku je součet sudých hodnot 0
  for(hodnota in vektor){ # Projdeme hodnoty vektoru
    if( (hodnota %% 2) == 0){ # Pokud je sudá
      vysledek <- vysledek + hodnota # Tak ji přičteme k výsledku
  } # Část else je zbytečná, tak ji smažeme
}
  return(vysledek) # Co má funkce vrátit
}
```

Pokud tento kód spustíme, v Global Enviroment se objeví nová funkce, kterou můžeme volat.

```
data <- c(1, -4, 0, 2, 3, 6) # Vstupní vektor
# Víme, že výsledek by měl být -4+0+2+6=4
secti_sude(data)

## [1] 4
```

Co se ale stane, když nebudeme „hodní“ a funkci dáme špatný vstup?

```
spatny_vstup <- c(4, 2, "A", 0, -3)
secti_sude(spatny_vstup)

## Error in hodnota%%2: non-numeric argument to binary operator

# Získáme chybu.
```

Je dobré, pokud nějakou funkci píšu si ošetřit (zkontrolovat), jestli je vstup dobrého typu. Upravíme naši funkci:

```
secti_sude<-function(vektor){
  if( is.numeric(vektor) ){ # Přidáme podmínku, vektor musí být numeric
    print("Vstup OK") # Vizuální potvrzení, není nutné
  vysledek <- 0 # Na začátku je součet sudých hodnot 0
  for(hodnota in vektor){ # Projdeme hodnoty vektoru
    if( (hodnota %% 2) == 0){ # Pokud je sudá
      vysledek <- vysledek + hodnota # Tak ji přičteme k výsledku
  } # Část else je zbytečná, tak ji smažeme
}
  return(vysledek) # Co má funkce vrátit
  } else {
    print("Vstup není numeric, hlupáku!") # Můžeme se uživateli i vysmát.
    return(NA) # Vrátí NA - neznámou hodnotu
  }
}
```

Vyzkoušíme naši novou funkci:

```
data <- c(1, -4, 0, 2, 3, 6) # Vstupní vektor
secti_sude(data)

## [1] "Vstup OK"
## [1] 4

spatny_vstup <- c(4, 2, "A", 0, -3)
secti_sude(spatny_vstup)

## [1] "Vstup není numeric, hlupáku!"
## [1] NA
```

Nakonec uděláme ještě jednu úpravu – modifikujeme tuto funkci tak, aby sčítala buď sudé, nebo liché, dle naší volby. Přidáme na vstupu druhý argument (parametr) `sude = T`. Znamená to, že parametr `sude` má defaultní hodnotu `TRUE`, čili pokud jej uživatel nezadá, bude se sčítat sudé

```
secti_sude_nebo_liche<-function(vektor, sude = T){
  # Následující if/else přiřadí hodnotě podmínky 0 nebo 1 -- což je
  # zbytek po dělení sudého resp. lichého čísla 2
  if(sude){
    hodnota_podminky <- 0
    print("Sčítáme sudé.")
  } else {
      hodnota_podminky <- 1
      print("Sčítáme liché")
    }

  if( is.numeric(vektor) ){
    print("Vstup OK")
  vysledek <- 0
  for(hodnota in vektor){
    if( (hodnota %% 2) == hodnota_podminky){  # !!!
      vysledek <- vysledek + hodnota
```

```
    }
}
  return(vysledek)
  } else {
    print("Vstup není numeric, hlupáku!")
    return(NA)
  }
}
```

Funkce bobtná, nabaluje se na ni další kód. Proto je důležité psát funkce přehledně, volit vhodné názvy pro funkce i proměnné (používat buď podtžítka_ nebo camelCase). A jak nyní funkce funguje?

```
data <- c(1, -4, 0, 2, 3, 6)
secti_sude_nebo_liche(data, sude = F) # Sečte liché hodnoty)

## [1] "Sčítáme liché"
## [1] "Vstup OK"
## [1] 4
```

Obsah

1 RStudio **1**

2 Working directory, balíčky, nápověda a komentáře **1**
2.1 Working directory (pracovní adresář) . . . 1
2.2 Instalace a spouštění balíčků . . . 2
2.3 Nápověda . . . 2
2.4 Komentáře . . . 2

3 Proměnné **3**
3.1 Datové typy . . . 3
3.2 Speciální hodnoty . . . 4

4 Vektory **5**
4.1 Inicializace a prodloužení vektoru . . . 5
4.2 Manipulace s hodnotami vektoru . . . 6
4.3 Matematické operace s vektory . . . 7
4.4 Statistické výpočty s vektory . . . 8

5 Matice **9**
5.1 Inicializace a prodloužení matice . . . 9
5.2 Manipulace s hodnotami matice . . . 10
5.3 Matematické operace matice . . . 11

6 Grafy **11**

7 Podmínky **13**

8 Cykly `for` a `while` **14**

9 Funkce **16**
9.1 Vlastní funkce . . . 16