Česká verze stránek není od roku 2013 aktualizována. Aktuální verzi této stránky najdete v její anglické verzi zde.

Vytvoření cyklu a definice funkce

Vytvoření cyklu, který bude schopný danou činnost x-krát zopakovat, a nadefinování funkce se naučíme na náhodné chůzi (*random walk*).

O co jde

Nejdříve stručný popis toto, o co ve skriptu půjde. Na začátku stojíme v bodě nula ($x_1 = 0$). V každém následujícím kroku (x_2, x_3, ...) vyrazíme šikmo doprava nebo šikmo doleva (rovně to nejde). Rozhodnutí o směru chůze je náhodné a můžeme ho udělat například tak, že si hodíme mincí (pana - doprava, orel - doleva).

Tento proces bude mít následující numerickou podobu: pokud se rozhodneme (respektive mince nás rozhodne) vyrazit vlevo, k současné pozici ($x_1 = 0$) přičteme jedničku ($x_2 = 1$); pokud vyrazíme doprava, jedničku odečteme ($x_2 = -1$).

V následujícím kroku postupujeme úplně stejně - náhodně se rozhodneme pro chůzi doprava nebo doleva, přičemž úplně ignorujeme minulé rozhodnutí o směru chůze, kterou jsme se dostali do současného bodu [1]. Takto postupujeme po určený počet kroků *k*, takže konečná pozice bude x_k.

Otázkou je, kam mě tento náhodný rozhodovací proces zanese, když udělám, dejme tomu, 1000 kroků. Člověk by tak nějak intuitivně očekával, že bude sice zatáčet doprava a doleva, ale ve výsledku, díky tomu že ten proces zatáčení bude náhodný, půjde více méně rovně. Je tomu ale opravdu tak?[2]

Vytvoření skriptu

Krok za krokem

Nejdříve si nadefinuji prázdný vektor, do kterého budu postupně ukládat informace o mojí současné pozici:

```
steps.rw <- vector (mode = 'numeric', length = 1000)
```

První argument mi říká, že vektor bude obsahovat čísla (alternativou by bylo `'character'` nebo `'logical'`), druhý argument určuje délku vektoru, tedy počet jeho elementů (pozor, správně je `length`, špatně `lenght`).

A teď začnu postupovat vpřed. Jak náhodně vybrat, jestli postupovat vpravo nebo vlevo? Pro náhodný

výběr je zde funkce `sample`, která má dva hlavní argumenty: `x` - z čeho vybírat, a `size` - kolik elementů vybrat. Pokud tedy napíšu

```
sample (x = c(-1,1), size = 1)

[1] -1
```

vybere se mi náhodně jedna hodnota z vektoru, obsahujících čísla `-1` a `1`, v mém případě to je hodnota `-1`.

Vektor `steps.rw` obsahuje samé nulové hodnoty, o čemž se snadno přesvědčíme:

```
steps.rw
   [1] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0
  [79] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0
  ...
  ...
```

První element tohoto vektoru je tedy nulový, což odpovídá mojí pozici v bodu nula. Druhý element vypočtu takto:

```
steps.rw [2] <- steps.rw [1] + sample (c(-1,1), 1)
```

Všimněte si, že ve funkci `sample` nevypisuji názvy jednotlivých argumentů (`x` a `size`) - není to potřeba, pokud hodnoty argumentů uvádím v definovaném pořadí (viz sekce `Usage` v eRkové nápovědě, do které se v případě funkce `sample` dostanu pomocí `?sample`). Další elementy můžu nadefinovat takto:

```
steps.rw [3] <- steps.rw [2] + sample (c(-1,1), 1)
steps.rw [4] <- steps.rw [3] + sample (c(-1,1), 1)
steps.rw [5] <- steps.rw [4] + sample (c(-1,1), 1)
steps.rw [6] <- steps.rw [5] + sample (c(-1,1), 1)
steps.rw [7] <- steps.rw [6] + sample (c(-1,1), 1)
steps.rw [8] <- steps.rw [7] + sample (c(-1,1), 1)
steps.rw [9] <- steps.rw [8] + sample (c(-1,1), 1)
steps.rw [10] <- steps.rw [9] + sample (c(-1,1), 1)
```

Použití cyklu

Zřejmě tušíte, že takto se daleko nedostanu - pro 1000 kroků bych musel řádek tisíckrát nakopírovat a pozměnit. Tady se právě bude hodit aplikovat cyklus, pro který je funkce `for`:

```
for (k in seq (2, 1000))
  {
  steps.rw [k] <- steps.rw [k-1] + sample (c(-1,1),1)
```

```
  }
```

Skript mi říká zhruba toto: do proměnné k postupně přiřaď hodnoty od 2 do 1000, a s každou z těchto hodnot udělej operaci, kdy na k-tou pozici ve vektoru `steps.rw` ulož hodnotu, kterou vypočteš z `k-1` té pozice vektoru `steps.rw` přičtením náhodně vybrané hodnoty `-1` nebo `1`.

```
steps.rw

   [1]   0  -1   0  -1   0  -1  -2  -3  -4  -3  -2  -3  -4  -3  -4  -5  -4
-5  -4  -5  -4  -5  -6  -7  -8  -9 -10  -9 -10 -11 -10  -9 -10 -11 -10  -9
-10 -11 -10
  [40] -11 -10 -11 -10  -9 -10  -9  -8  -7  -6  -5  -4  -3  -2  -1   0   1
2   3   4   3   2   1   2   3   2   1   2   3   4   5   6   5   4   3   2
3   2   3
  [79]   4   5   6   7   8   7   8   7   6   7   8   7   8   7   8   7   8
7   6   5   6   5   4   5   6   7   8   9  10   9   8   7   8   9  10  11
10   9   8
 [118]   9   8   9  10  11  10  11  12  11  12  13  12  13  14  13  14  15
16  15  16  15  16  15  14  13  12  11  12  11  10  11  12  11  12  13  14
13  14  15
 [157]  16  17  16  17  18  17  16  15  14  13  12  11  10  11  12  13  12
11  10   9   8   9   8   9  10   9   8   9  10   9   8   7   8   9  10   9
8   7   8
...
...
```

Ještě lépe, když si to celé nakreslím:

```
plot (x = steps.rw, type = 'l')
```

Obrázek zde nenajdete, připravil bych vás o okamžik překvapení. Navíc je jasné, že obrázek bude pro každou náhodnou chůzi vypadat jinak. Argument x ve funkci `plot` obsahuje hodnoty, které chci nakreslit. Teoreticky bych měl zadat argumenty dva, tedy x a y, což by byly dva vektory o stejné délce, první udávající pozici bodu na ose x, druhý na ose y. Pokud argument y chybí (jako v našem případě), výsledek je následujicí: hodnoty z argumentu x se kreslí na vertikální osu, a na horizontální se kreslí jejich pořadí ve vektoru (proto má výsledný obrázek osu x popsanou jako Index).

Definice a použití funkce

Pokaždé, když daný cyklus o tisíci krocích provedu znovu, dostanu jiný tvar mé náhodné chůze. Možná by se hodilo vidět, co se stane, když cyklus náhodné chůze zopakuji třeba 25 krát. Mohl bych jednoduše cyklus o 1000 krocích zanořit do dalšího cyklu, který by tentokrát definoval počet opakování celé procedury. Půjdu na to ale ještě jinak. Z celého skriptu, který mi vytvoří jednu náhodnou chůzi (o daném počtu kroků, které musím udělat od bodu nula) vytvořím funkci.

```
random.walk <- function (no.steps)
    {
    steps.rw <- vector (mode = 'numeric', length = no.steps)
    for (k in seq (2, no.steps))
```

```
      {
      steps.rw [k] <- steps.rw [k-1] + sample (c(-1,1),1)
      }
    }
```

Nově vytvořená funkce se jmenuje `random.walk` a vyžaduje jediný argument s názvem `no.steps`, tedy počet kroků. V prvním kroku funkce vytvoří prázdnou proměnnou steps.rw, která představuje numerický vektor o délce `no.steps`. Ve druhém kroku se tato proměnná začíná plnit hodnotami z cyklu, který jsme použili už před chvílí - cyklus začíná pozicí 2 a končí pozicí o hodnotě `no.steps`.

Zkusíme funkci použít:

```
random.walk (1000)
```

Pokud jsem někde neudělal chybu, funkce proběhne, ale nevrátí mi žádný výsledek. Proměnná `steps.rw` v rámci funkce skutečně vznikla, ale zůstala ve funkci "uvězněna". Je potřeba, aby ji funkce "poslala ven", což zařídím tak, že na konci definice funkce napíšu její jméno, stejně jako když se chci podívat na to, co proměnná obsahuje:

```
random.walk <- function (no.steps)
    {
    steps.rw <- vector (mode = 'numeric', length = no.steps)
    for (k in seq (2, no.steps))
      {
      steps.rw [k] <- steps.rw [k-1] + sample (c(-1,1),1)
      }
    steps.rw
    }
```

Tady jen malá odbočka: pokud nechci funkci definovat celou znovu, tak jak jsem to udělal nyní, mohu ji jen otevřít pro editaci příkazem `edit`:

```
random.walk <- edit (random.walk)
```

Editovanou verzi je třeba přiřadit znovu do proměnné stejného (případně jiného) jména, aby se mi zeditované změny uložily - pokud použiju pouze `edit (random.walk)`, po editaci se mě eRko sice zeptá na uložení změn, ale vlastní funkce se nezmění. Analogií je funkce `fix (random.walk)`, která se naopak používá jako taková a změny se automaticky uloží do definice původní funkce [3)].

Znovu se podívám, jak funkce pracuje (použiju menší počet kroků, aby výsledný vektor nebyl tak dlouhý pro výpis):

```
random.walk (100)

  [1]   0  -1   0   1   2   1   0  -1  -2  -1   0  -1   0  -1  -2  -3  -4
-5  -4  -5  -4  -3  -4  -5  -6  -5  -6  -7  -8  -9 -10  -9  -8  -9  -8  -7
-8  -9 -10
 [40] -11 -12 -13 -14 -15 -14 -13 -12 -11 -12 -11 -12 -13 -14 -13 -12 -13
-12 -13 -12 -11 -12 -13 -12 -11 -10 -11 -12 -11 -10  -9  -8  -9  -8  -9 -10
-11 -12 -13
 [79] -12 -13 -14 -13 -14 -13 -12 -13 -14 -13 -14 -13 -12 -13 -14 -13 -14
```

```
-13 -14 -15 -16 -15
```

Celé to můžu případně nakreslit:

```
plot (random.walk (1000), type = 'l')
```

A teď onen slíbený obrázek, ve kterém se mi nakreslí několik náhodných chůzí najednou. Udělám nejdříve úpravu grafického rozhraní (budeme mu říkat dejme tomu kreslící plátno) tak, že ho rozdělím na 25 polí (5 řádků, 5 sloupečků), což se provede nastavením parametru `mfrow`. Dále chci, aby vlastní obrázek neměl žádné okraje (*margins*), kam se normálně kreslí popisky os a jednotlivé hodnoty - to provedu nastavením parametru `mar` na samé nuly. O detailech a dalších grafických parametrech se dozvíte v nápovědě k funkci `par`.

```
par (mfrow = c (5,5))
par (mar = c (0,0,0,0))
```

A nyní vytvořím cyklus, který mi těch 25 obrázků postupně nakreslí:

```
for (i in seq (1, 25))
    {
    plot (random.walk (1000), type = 'l', ann = F, axes = F)
    box ()
    }
```

V tomto cyklu ve skutečnosti proměnnou `i` na nic nepotřebujeme, její definice nám pouze zajistí, že se cyklus zopakuje právě 25 krát. V kreslící funkci `plot` se objevily další dva argumenty: `ann = F` znamená, že se nebude vykreslovat annotace os (jejich názvy), `axes = F` zase zajistí, že se nebudou osy ani jednotlivé značky vůbec vykreslovat. Výsledný graf by pak ale “visel úplně ve vzduchu”, proto je nakonec ještě doplněna funkce `box ()`, která kolem grafu nakreslí krabici (v místech, kde původně byly osy).

Souhrn skriptu pro random walk

```
set.seed (212121)  # pokud nastavíte seed stejné jako já, dostanete úplně
stejný obrázek

random.walk <- function (no.steps)
{
  steps.rw <- vector (mode = 'numeric', length = no.steps)
  for (k in seq (2, no.steps))
  {
    steps.rw [k] <- steps.rw [k-1] + sample (c(-1,1),1)
  }
  steps.rw
}


par (mfrow = c (5,5))
par (mar = c (0,0,0,0))
```

```
for (i in seq (1, 25))
{
  plot (random.walk (1000), type = 'l', ann = F, axes = F)
  box ()
}
```

Vlastní obrázek najdete zde.

Skript pro 3D random walk

```
library (rgl) # pokud neni knihovna nainstalovana, pouzijte install.packages
("rgl")

no.steps <- 10000
steps.rw <- matrix (0, ncol = 3, nrow = no.steps)
for (k in seq (2, no.steps))
  {
  steps.rw[k,] <- steps.rw[k-1,]
  which.to.add <- sample (1:3, 1)
  steps.rw[k, which.to.add] <- steps.rw[k, which.to.add] + sample (c (-1,1),
1)
  }

rgl.linestrips (steps.rw)

# pokud chcete proces kresleni animovat, pouzijte nasledujici cyklus:
# for (i in seq (2, no.steps)) rgl.linestrips (steps.rw[1:i,])
```

1)
Toto není pro člověka úplně intuitivní - minulé rozhodnutí má tendenci zohledňovat v rozhodnutích následujících. Příklad: jdu do kantýny, kde vaří v podstatě jen samé blafy, takže je celkem jedno, co si člověk dá. Zároveň se ani nenamáhají vyvěšovat jídelní lístek - vím jen, že na výběr je jídlo A, B, C a D. První den si dám Áčko. Katastrofa. Druhý den přijdu zas, a i když vím, že se můžu rozhodnout úplně náhodně, Áčko si nedám, zkusím Béčko. Taky katastrofa. Příští den si dám radši zase Áčko, protože to předevčírem byl sice blivajs, ale pořád o trošku menší než to včerejší Béčko. Zklamu se, tentokrát je to ještě horší, zítra si dám radši Céčko a od příštího týdne budu radši hladem.
2)
Máte teď v podstatě dvě možnosti - (a) vzít minci do hrsti a vyrazit na pole, nebo (b) vzít mozek do hrsti a spustit eRko. Osobně doporučuji možnost (a), která je sice podstatně pracnější, ale daleko zdravější a možná i zábavnější.
3)
Funkce `edit` a `fix` se dají s úspěchem použít i na editaci objektů typu `matrix` a `data.frame` - RStudio, pokud ho používáte, však editaci matic a datových rámců funkcí `edit` nepodporuje.