Univerzita Jana Evangelisty Purkyně
v Ústí nad Labem

Přírodovědecká fakulta

Prezentace výsledků a vizualizace algoritmů v prostředí R s využitím balíčků knitr a Shiny

Bakalářská práce

Vypracoval: Vladimír Ctibor
Vedoucí práce: RNDr. Jiří Škvor, Ph.D.

Studijní program: Informační systémy
Studijní obor: Aplikovaná informatika

Ústí nad Labem 2018

Zde vložte zadání

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně a použil jen pramenů, které cituji a uvádím v přiloženém seznamu literatury.

Byl jsem seznámen s tím, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona c. 121/2000 Sb., ve znění zákona č. 81/2005 Sb., autorský zákon, zejména se skutečností, že Univerzita Jana Evangelisty Purkyně v Ústí nad Labem má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona, a s tím, že pokud dojde k užití této práce mnou nebo bude poskytnuta licence o užití jinému Subjektu, je Univerzita Jana Evangelisty Purkyně v Ústí nad Labem oprávněna ode mne požadovat přiměřený příspěvek na úhradu nákladu, které na vytvoření díla vynaložila, a to podle okolností až do jejich skutečné výše.

V Ústí nad Labem dne 3. května 2018 Podpis: .....

Děkuji vedoucímu práce RNDr. Jiřímu Škvorovi, Ph.D.
za neocenitelné rady a pomoc při tvorbě bakalářské práce.

Abstrakt

Cílem práce je popsat využití programovacího jazyka R k prezentaci dat, algoritmů a výsledků měření či simulačních experimentů. K tomuto účelu je použit framework pro tvorbu reaktivních webových aplikací *Shiny* a knihovna *knitr*, která je určená k vytváření reprodukovatelných reportů. Použití těchto nástrojů je demonstrováno na řešení několika úloh z různých oblastí, jako jsou obecně analýza dat, numerické metody, výpočetní geometrie či molekulární simulace.

Klíčová slova

R, knitr, Shiny, analýza dat, numerické metody, simulace

Abstract

Objective of this work is to describe utilization of R programming language for presentation of data, algorithms, measurement results and simulation experiments. Frameworks *Shiny* and *knitr*, used for creation of reactive web applications and reproducible reports, respectively, are presented for this purpose. Usage of these tools is demonstrated on several exercises from different domains, such as general data analysis, computational geometry or molecular simulations.

Key words

R, knitr, Shiny, data analysis, numerical analysis, simulations

Obsah

Úvod

Žijeme v době, kdy lidstvo produkuje více dat než kdykoliv dříve v historii. Jejich efektivní analýza a prezentace je nezbytná schopnost napříč obory a profesemi. Grafická znázornění v různých formách od nepaměti slouží jako prostředek vizualizace libovolných dat, prostředek mnohem působivější, než je forma, ve které s nimi běžně pracujeme, což jsou zpravidla dlouhé řady čísel a složitých matematických výrazů. Vizualizace ovšem nemusí být jen nástrojem pro prezentaci konečných výsledků, své uplatnění nalezne i při analýze.

Moderní počítačové systémy jsou mocným nástrojem nejen ve fázi výzkumu, ale mohou posloužit také k prezentaci výsledků, přičemž nám dávají dříve nepředstavitelné možnosti. Především již nejsme omezeni statickou povahou tištěných materiálů. V dnešní době můžeme prezentovat naši práci interaktivně. Můžeme vytvořit prostředí, které komukoliv umožní zkoumat naše data skutečně do hloubky, „hrát“ si s nimi, měnit vstupní parametry a pozorovat, jakým způsobem jsou ovlivněny výstupy.

S prezentací výsledků, především vědeckých bádání, a měření pak souvisí koncept reprodukovatelného výzkumu. Jde o myšlenku, která říká, že výsledkem vědeckého výzkumu by neměl být jen odborný článek, ale kompletní prostředí, které umožní přesnou reprodukci dosažených výsledků. Zopakovatelnost vědeckého výzkumu je obzvláště důležitá, chceme-li je aplikovat, například jako průmyslový proces, při stavbě nových strojů a podobně. Bohužel se nedá říci, že jsou veškeré existující vědecké práce perfektně reprodukovatelné. Svou roli nepochybně hraje skutečnost, že vytvořit a publikovat práci v takové podobě, která umožní snadnou zopakovatelnost, není triviální úkol a ne každý zná nástroje pro to nezbytné.

Cílem této práce je prozkoumat a ukázat možnosti dvou nástrojů, které mohou posloužit pro řešení výše uvedených problémů a umožnit dosáhnout chvályhodných cílů, kterými jsou snadno reprodukovatelný výzkum a působivá, interaktivní vizualizace.

Těmito nástroji jsou *Shiny* a *knitr*. V obou případech se jedná o rozšiřující knihovny pro programovací jazyk R. *Shiny* slouží k vytváření webových aplikací. Umožňuje v pouhých pár řádcích kódu realizovat dobře vypadající, interaktivní aplikaci bez potřeby hlubší znalosti webových technologií a jazyků. Balík *knitr* nám dává možnost volat programy psané v některém z mnoha programovacích jazyků z textových dokumentů. Díky tomu je možné článek, popisující výsledky nějakého měření či experimentu, přímo propojit se zdrojovým kódem, který jsme použili k provedení těchto měření či experimentů.

Pro účely demonstrace těchto nástrojů bylo vybráno sedm úloh z rozličných oborů. Těmito úlohami a obory jsou analýza dat (k-means, word cloud a Schulzeho metoda), numerické metody (konkrétně pro řešení nelineárních rovnic), výpočetní geometrie (detekce rozhraní pomocí Delaunayovy triangulace) a molekulární simulace (pro ukázku Monte Carlo simulace byl zvolen Isingův model jako zástupce mřížkových modelů, pro ukázku molekulárně dynamické simulace byl zvolen systém tuhých disků). Tato práce je rozdělena na dvě části, teoretickou a praktickou. Teoretická část obsahuje stručný popis jazyka R (historie, instalace,

vývojová prostředí a syntaxe jazyka), dále pak popis knihoven *Shiny* a *knitr*. Druhá polovina teoretické části je věnována jednotlivým úlohám. Pro každou úlohu je popsán problém, který řešíme, používané algoritmy a praktické využití těchto konceptů. Následně praktická část práce obsahuje samotnou implementaci těchto úloh. Implementace každé úlohy je rozdělena na tři části. První částí je společný backend, což je *R* skript, který obsahuje kolekci funkcí pro řešení dané úlohy. Funkce obsažené v tomto backendu jsou využívány jednak v textu této práce pro demonstraci knihovny *knitr* ve druhé části popisu implementace každé úlohy a jednak v poslední z těchto tří částí, ve které je představena *Shiny* aplikace, která obsahuje interaktivní vizualizaci dané problematiky.

Část I.

Teoretická část

1. Jazyk a prostředí R

R je software pro provádění statistických výpočtů a analýzu dat. Jedná se o integrované prostředí pro manipulaci s daty, pro výpočty s důrazem na širokou škálu technik používaných ve statistice, a vytváření grafických výstupů pro prezentaci experimentů a výsledků.

Hlavní součástí tohoto prostředí je stejnojmenný programovací jazyk. Jedná se o open source implementaci jazyka S. Ačkoli mezi nimi existují rozdíly, většina programů napsaných v S by měla být spustitelná pod R bez potřeby úprav. S se stal populárním nástrojem pro řešení problémů ve statistice, a R přináší možnost podílet se na této aktivitě za použití svobodného software. R je součástí projektu GNU, licencovaný pod podmínkami GNU General Public License. [1]

Autoři R kladou velký důraz na rozšířitelnost tohoto prostředí. Zakomponování plnohodnotného programovacího jazyka umožňuje vytváření vlastních funkcí. Ty se pak dají volat stejně snadno jako vestavěné funkce. Co je ovšem důležitější, R, podobně jako řada dalších moderních jazyků, obsahuje integrovaného správce balíčků. Díky tomu je možné nainstalovat libovolnou rozšiřující knihovnu jediným příkazem. Knihovny se stahují z úložiště CRAN (Comprehensive R Archive Network), což je celosvětová síť http a ftp serverů poskytujících zdrojové kódy, binární soubory a dokumentaci pro samotné R i rozšiřující balíčky. V současné době je na CRANu hostováno přes jedenáct tisíc rozšiřujících knihoven pro R. [2] V neposlední řadě je možné propojit programy psané v R s programy či funkcemi psanými v jiných jazycích – samotné R obsahuje funkce pro volání kódu napsaného v C a Fortranu, dále pak existují projekty jako například *R.Net*, umožňující propojení R s programy psanými v jazycích rodiny CLR, především C# a F#.

R je možné nainstalovat na naprostou většinu v dnešní době používaných systémů. Jedná se o UNIXové systémy, především Linux, MacOS X a FreeBSD, dále pak Windows a MacOS. R je tedy snadno dostupné pro každého zájemce. V základní instalaci je k dispozici pouze textové rozhraní, ale existuje široká nabídka grafických nástrojů pro práci s R.

1.1. Historie

R je implementací jazyka S, vyvinutého mezi lety 1975 a 1976 v Bell Laboratories Rickem Beckerem, Allanem Wilksem a Johnem Chambersem. S bylo původně určené pro operační systém GCOS, ale v roce 1979 se primární platformou stal UNIX. V roce 1988 vydaná verze číslo 3 přinesla mnoho změn, především nahrazení maker funkcemi, změna způsobu předávání funkcí mezi funkcemi vyšších řádů a rozšíření konceptu objektů. Revize verze 3, provedená v roce 1991, zavedla datový typ dataframe. V roce 1989 vyšla zatím nejnovější revize S, S4, zaměřená především na rozšíření objektově orientovaných aspektů jazyka. Třídy typu S4 se značně liší od tříd typu S3, přičemž v R jsou dostupné oba typy. [1, 3]

Počátky R se datují do první poloviny devadesátých let. Původně se jednalo o jazyk určený pro výuku základů statistiky na univerzitě v Aucklandu na Novém Zélandě. Jeho autory jsou Ross Ihaka a Robert Gentleman. V roce 1992 padlo rozhodnutí použít syntaxi jazyka S. V roce 1994 je dokončena počáteční verze a zveřejněna na Internetu pod licencí GPL. V roce 1997 se R oficiálně stává součástí projektu GNU. 29. února 2000 je jazyk prohlášen za dostatečně kompletní, aby mohla být vydána verze 1.0. Verze 2.0 vyšla v roce 2004. V současné době je jazyk ve verzi 3. Jádro jazyka je implementováno v jazycích C, C++ a Fortran, velká většina funkcí je ovšem napsána přímo v *R*. [1, 4]

1.2. Instalace

Přesný způsob instalace R záleží na cílovém systému. Existují dvě hlavní cesty, kterými se lze vydat. Buď se rozhodneme pro kompilaci ze zdrojových souborů, nebo použijeme již zkompilované binární soubory. Z CRANu lze stáhnout binární soubory pro Windows, Mac OS, Mac OS X a Linuxové distribuce Ubuntu, Debian, Suse a RHEL. Zde bude popsána instalace na systém Windows 10 a Linuxovou distribuci Ubuntu 17.04.

1.2.1. Windows

V případě instalace na systém Windows stačí stáhnout požadovaný .exe soubor (32bit nebo 64bit). R je také k dispozici jako balíček pro Windows package manager *Chocolatey*. Pokud používáte *Chocolatey*, dá se R nainstalovat jednoduchým příkazem `choco install r.project`.

1.2.2. Ubuntu

Pro instalaci R na Ubuntu je možné z CRANu stáhnout instalační balíček *.deb*. Nejprve zvolíme verzi Ubuntu, na kterou chceme R nainstalovat, na CRANu jsou jednotlivé verze označené podle oficiálních kódových názvů, pro Ubuntu 17.04 tedy zvolíme složku Zesty. Dále se musíme rozhodnout, jestli chceme 32 bitovou nebo 4 bitovou verzi R. Instalační balíčky mají název ve tvaru `r-base-core_[verze R][kódové jméno verze Ubuntu]_[architektura procesoru].deb`, tudíž pokud chceme nejnovější verzi 64 bitového R (v současné době 3.4.0) pro Ubuntu 17.04, stáhneme balíček s názvem `r-base-core_3.4.0-1zesty_amd64.deb`. Tento soubor můžeme otevřít v *Software Center* pro grafickou instalaci, nebo jednoduše nainstalovat příkazem `sudo dpkg -i r-base-core_3.4.0-1zesty_amd64.deb`. [5]

Alternativně můžeme přidat některý z mirrorů CRANu do seznamu zdrojů balíčků pro package manager apt. Toho dosáhneme modifikací souboru `/etc/apt/sources.list`, kam připíšeme `deb https://mirrors.nic.cz/R/bin/linux/ubuntu zesty/`. Následně provedeme aktualizaci dostupných programů příkazem `sudo apt update` a instalaci příkazem `sudo apt install r-base`. Výhodou tohoto mírně složitějšího způsobu je zjednodušení aktualizací, o které se stará program *apt* – provedeme-li manuální instalaci stažením souboru *.deb*, musíme si aktualizace zařizovat sami. [6]

1.3. Vývojové nástroje

Ačkoli programy lze psát i v tom nejjednodušším textovém editoru, existuje řada vylepšení, která umožňují vytvářet kvalitnější a komplexnější kód s vynaložením menšího množství úsilí. Toto tvrzení je platné pro libovolný programovací jazyk, tedy i pro *R*. V této části jsou představeny některé programy, které nám mohou práci s *R* výrazně usnadnit.

1.3.1. Textové rozhraní

Samotné R, pokud nedoinstalujeme žádný z grafických vývojových nástrojů, poskytuje pouze textové rozhraní. V unixových systémech jej spustíme příkazem `R`. Tím se dostaneme do běžného REP (read-eval-print) cyklu, kde můžeme zadávat příkazy (platné výrazy jazyka R) a dostáváme okamžitou zpětnou vazbu v podobě výstupů vyhodnocení daných výrazů, případně chybových hlášek a varování. REP cyklus prostředí R rozumí idiomům známým z unixovských shellů, tedy můžeme používat šipky nahoru a dolů pro procházení historie příkazů, tab pro dokončení výrazu a Ctrl+C pro ukončení vyhodnocování příkazu. Dále je interpreter R schopen rozpoznat, jestli jsme zadali sémanticky kompletní výraz – pokud ano, po odentrování jej okamžitě vyhodnotí a vrátí výsledek. Pokud jsme ovšem zadali a odentrovali neúplný výraz, interpreter před zahájením vyhodnocování počká na dokončení výrazu a toto čekání vyjádří změnou promptu. Tím je možné komplikovanější příkazy rozdělit na více řádků. Stejně tak je možné zadat více výrazů na jeden řádek tím, že jednotlivé příkazy oddělíme středníkem. REP cyklus opustíme zavoláním funkce `q()`.

Při spouštění R je možné nastavit řadu přepínačů, například `--silent` pro vypnutí úvodní zprávy, `--encoding` pro nastavení kódování textu čteného ze standardního vstupu či `--file` pro čtení příkazů ze zdrojového souboru. Úplný seznam přepínačů lze získat standardním `man R`, či předáním přepínače `--help`.

Chceme-li spouštět skripty napsané v R, použijeme příkaz `Rscript`. Tedy pokud máme zdrojový soubor `file.R`, můžeme jej nechat vykonat příkazem `Rscript file.R`. Alternativní možností je umístit na první řádek zdrojového souboru tak zvaný shebang v podobě `#!/usr/bin/Rscript` (Za token #! se píše absolutní cesta ke spustitelnému souboru Rscript, a ta se může na různých systémech lišit!) a skript pak můžeme spouštět známým způsobem `./file.R`. Jiný způsob spouštění skriptů je přímo z prostředí R zavoláním funkce `source()`, jejímž jediným argumentem je textový řetězec obsahující absolutní či relativní cestu k souboru, který chceme vykonat.

Ačkoliv se na první pohled může čistě textové rozhraní zdát velmi strohé, obsahuje plnohodnotné prostředí R a dává k dispozici, alespoň v principu, zcela stejný rozsah funkcionality jako jakékoliv grafické vývojové prostředí. Ačkoli ve většině případů budeme chtít raději použít grafické prostředí, naprostá většina těchto programů dává k dispozici i textovou konzoli, která umožňuje práci s R tímto způsobem. REP cyklus je velmi užitečný nástroj při vývoji skriptů, kde umožňuje rychlé experimentování a ověřování s okamžitou zpětnou vazbou, a je tedy vhodné naučit se využívat jej efektivně. Nakonec, pokud chceme provést jen pár příkazů či zavolat několik funkcí, může být využití textového rozhraní rychlejší a pohodlnější než komplexní grafický program.

Ještě zmiňme, že i pokud dáváte přednost textovému rozhraní před grafickým, není nutné zů-

stávat u prostředí, které nabízí základní instalace R. Existuje program *rice*, který poskytuje moderní konzoli pro R, s mnoha funkcemi známými z grafických vývojových nástrojů (barevné zvýrazňování syntaxe, automatické dokončování výrazů apod.). Přitom se stále jedná o běžnou terminálovou aplikaci. Tento program je možné stáhnout z `github.com/randy3k/rice` či nainstalovat pomocí správce balíčků programovacího jazyka Python *pip*.

1.3.2. Integrovaná vývojová prostředí

K práci s R existuje řada rozšiřujících programů, které nabízejí grafické rozhraní a funkcionalitu tak zvaných integrovaných vývojových prostředí – to znamená, že v jediném programu je obsažena široká řada funkcí, od textového editoru, přes nejrůznější vývojové a debugovací nástroje, vykreslování grafických výstupů až například po systémy pro správu verzí, jako je Git či Subversion.

Nejrozšířenějším takovým programem pro R je *RStudio IDE*. Je k dispozici pro operační systémy Windows, Mac OS X a Linuxové distribuce Ubuntu, RHEL, Fedora a OpenSUSE. K dispozici je také verze, kterou je možné nainstalovat na server a přistupovat k ní jako webové aplikaci, což činí *RStudio* dostupné pro prakticky jakoukoli existující platformu. Instalace je jednoduchá, stačí z oficiálních webových stránek stáhnout instalační soubor pro požadovaný operační systém a postupovat způsobem běžným pro danou platformu. [7]

Uživatelské rozhraní *RStudia* je rozděleno do čtyř podoken. Výchozí rozmístění oken je takové, že v levé horní části se nachází textový editor, pod textovým editorem najdeme okno se dvěma funkcemi, nazvanými *Console* a *Terminal*, přičemž *Console* dává přístup k běžnému textovému rozhraní R, jaké jsme popsali v předchozí kapitole, a záložka *Terminal* dává k dispozici standardní textové rozhraní daného operačního systému. Na pravé straně se nachází dvojice oken, která dohromady zobrazují řadu užitečných informací – souborový adresář, grafy, vestavěnou nápovědu a mnoho dalších. Jednotlivé funkce v těchto dvou oknech je možné do určité míry přesouvat, takže uživatel není omezen výchozím rozvržením, stejně tak je možné zaměnit pozici libovolných dvou oken. Možné úpravy ovšem nejsou neomezené, není například možné změnit celkový počet oken. Velikost oken je libovolně nastavitelná, změny lze dosáhnout buď chycením a tažením okraje okna, nebo použitím tlačítek na minimalizaci a maximalizaci okna. Ne zrovna šťastnou vlastností *RStudia* je skutečnost, že nastavení uživatelského rozhraní ovlivňuje vykonávání kódu v R: například má-li okno pro vykreslování grafů příliš malou velikost nebo je rovnou minimalizované a náš skript vykresluje graf, dojde k zastavení vykonávání skriptu chybou v místě volání funkce `plot()`.

Kromě aplikací vytvořených speciálně pro R jsou k dispozici také pluginy přidávající podporu R do oblíbených vývojových prostředí, jako je *Visual Studio* (*R Tools for Visual Studio*) [8], *IntelliJ IDEA* (*R Language Support*) [9] či *Eclipse* (*StatET*) [10]. Pokud se plnohodnotné IDE jeví jako příliš těžkopádné řešení, další možností je využít pokročilé textové editory, jež s pomocí pluginů dokáží nabídnout funkcionalitu velmi blízkou IDE, ale s menšími nároky na zdroje – z této oblasti stojí za zmínku *Sublime Text* s pluginem *R-Box*. [11]

Žádný z těchto nástrojů ovšem není nezbytností. R nabízí veškerou svou funkcionalitu prostřednictvím textového rozhraní a skripty lze psát v libovolně jednoduchém textovém editoru. Záleží tak čistě na preferencích a možnostech každého uživatele, jaké nástroje se rozhodne pro svou práci používat.

2. R, Shiny a knitr

Tématem této práce jsou nástroje *knitr* a *Shiny*, jež budou popsány v této kapitole. V obou případech se jedná o knihovny jazyka *R*. Pro jejich použití je nezbytná alespoň základní znalost tohoto jazyka, proto zde naleznete i stručný úvod do syntaxe *R*.

2.1. Syntaxe jazyka R

R je interpretovaný, dynamicky typovaný jazyk s automatickou správou paměti. Ačkoliv umožňuje programování v různých paradigmatech, jeho návrh byl značně ovlivněn Lispem a dalšími funkcionálními jazyky, což se projevuje v mnoha rysech R. [12] Funkcionální, výrazově orientovaný způsob programování také dobře ladí s interaktivní prací přes příkazový řádek, a je tedy vhodným způsobem psaní programů v R. V neposlední řadě, využití funkcionálních idiomů může programy v R urychlit.

Vliv funkcionálních jazyků se v R projevuje i silnou orientací na výrazy. Většina konstrukcí v jazyku se vyhodnocuje a vrací nějakou hodnotu. Jako jeden z důsledků tohoto návrhu není funkce nutné zakončovat příkazem `return()`, automaticky vrátí poslední vypočtenou hodnotu (a tedy `return()` je nutné použít pouze v případě, že chceme vrátit nějakou jinou hodnotu). Podobně konstanty jsou výrazy, které při vyhodnocení vrátí samy sebe. Jednotlivé výrazy mohou být odděleny středníkem nebo novým řádkem. Středník výraz vždy ukončuje, nový řádek výraz může ukončit, pokud je výraz syntakticky kompletní. Interpreter dokáže poznat, že výraz není kompletní, a očekává jeho dokončení, přičemž v případě práce v příkazovém řádku toto indikuje změnou promptu.

2.1.1. Datové struktury

R je poněkud specifický v tom, že neobsahuje atomární proměnné. Nejjednodušší datovou strukturou je tak zvaný *vektor*, což je uspořádaná množina libovolně mnoha prvků stejného datového typu. Vektor vytvoříme funkcí `vector()`, která nám umožní nastavit datový typ a délku vektoru, či funkcí `c()` (od slova *combine*), která přijímá rovnou jednotlivé prvky vektoru – v takovém případě je datový typ vektoru zvolen automaticky na základě nejnižšího společného jmenovatele všech prvků (např. vytvoříme-li vektor obsahující jen logické hodnoty, bude datový typ bool, ovšem přidáme-li číslo, všechny logické hodnoty se automaticky konvertují na čísla, přidáme-li string, všechny logické či číselné elementy se konvertují na textové řetězce).

Další běžné datové struktury, se kterými se můžeme v R setkat, jsou seznam, matice, pole a dataframe. Seznam můžeme chápat jako vektor vektorů, přičemž každý prvek (vektor) může mít odlišný datový typ. Seznam je specifický tím, že má charakteristiky dvou běžných

datových struktur známých z jiných programovacích jazyků: seznamu a slovníku. Prvky v seznamu v R jsou uspořádané a lze k nim přistupovat podle jejich indexu (běžná charakteristika seznamu), současně ovšem mohou mít klíč, přes který se k nim dá také přistupovat, a je možné vložit prvek na libovolnou pozici seznamu, aniž by musely být obsazeny všechny předchozí pozice (toto je chování běžné spíše pro slovníky). Pole jsou rozšířením vektoru do více dimenzí a matice jsou speciálním, dvoudimenzionálním případem polí. Dataframe je další dvourozměrnou datovou strukturou, lišící se od matic tím, že jednotlivé sloupce mohou mít odlišný typ.

2.1.2. Typový systém

Vzhledem k tomu, že se jedná o dynamicky typovaný jazyk, nepřichází uživatel při běžné práci přímo do styku s typovým systémem R. Přesto je dobré mít přehled alespoň o základních typech (v prostředí R také někdy nazývané *módy*). Jsou jimi *logical*, *numeric* a *character*. Logical je vektor obsahující pouze logické hodnoty `TRUE` či `FALSE`. Numeric je číselný vektor. Tento mód se ještě dále dělí na *double*, *integer* a *complex*. Double je výchozí datový typ pro číselné vektory v R a reprezentuje reálné číslo s plovoucí desetinnou čárkou. Integer je celočíselný vektor, a k jeho vytvoření na konec čísla napíšeme velké `L`. Complex reprezentuje komplexní číslo, a abychom ho vytvořili, zapíšeme toto číslo jako součet reálné a imaginární složky zakončené malým `i` (nebo jen imaginární složku zakončenou malým `i`). Nakonec character je textový řetězec. Datový typ objektu v R můžeme zjistit pomocí vestavěné funkce `typeof()`.

```
# Vytvoříme-li číselný vektor, bude jeho datový typ double:
A = c(1, 2, 3)
typeof(A)

## [1] "double"

B = c(1.1, 2.0, 3.5)
typeof(A)

## [1] "double"

# Celočíselný vektor vytvoříme sufixem L:
C = c(1L, 2L, 3L)
typeof(C)

## [1] "integer"

# Vektor komplexních čísel:
D = c(5i, 0+1i, 8-3i)
typeof(D)

## [1] "complex"
```

```r
# Nakonec vektor logických hodnot a vektor textových řetězců:
E = c(TRUE, FALSE, F, T)
typeof(E)

## [1] "character"

F = c("Ahoj", "Světe")
typeof(F)

## [1] "character"
```

2.1.3. Operátory

V R je nám k dispozici běžná suita aritmetických a logických operátorů. Těmi číselnými jsou běžné operátory pro základní operace (sčítání, odčítání, násobení a dělení), dále pak operátor pro umocňování (můžeme si vybrat mezi `^` a `**`), `%%` pro modulo a `%/%` pro celočíselné dělení. Mezi logickými operátory nalezneme všechny možné kombinace rovná se a nerovnosti pro porovnávání, test na rovnost se provádí operátorem `==`, nerovnost `!=`, negace `!`. Pro logický součin a součet máme opět známé operátory `&&` a `||`. V tomto ohledu nás R ničím nepřekvapí.

V čem nás R překvapit může, je množství operátorů určených ke zdánlivě triviální operaci přiřazení hodnoty k názvu proměnné. Těmito operátory jsou `<-`, `<<-`, `->`, `->>` a `=`. `<-`, `->` a `=` provádějí jednoduché přiřazení. `<<-` a `->>` přiřazují proměnné v oblasti platnosti o jednu úroveň vyšší, než kde se nachází operátor, můžeme tak například vytvořit proměnnou v globálním prostředí zevnitř funkce. Operátory `->` a `->>` se od svých obrácených protějšků liší pouze tím, na kterou stranu výrazu se píše název proměnné a na kterou hodnota. Zbývá jen pochopit rozdíl mezi `<-` a `=`. `<-` je v R obsažena z historických důvodů (tento přiřazovací operátor byl používán v S) a ve svém chování se neliší od `=` až na jediný případ, a tím je přiřazení uvnitř volání funkce. Pokud při volání funkce předáme parametry pomocí operátoru `<-`, je nejprve vytvořena pojmenovaná proměnná a tato proměnná je předána funkci jako poziční parametr. Naopak, použijeme-li na stejném místě operátor `=`, dojde k předání hodnoty funkci jakožto pojmenovaný parametr.

```r
# Funkce vector(mode, length) vytváří vektor dané délky a datového typu.
# Předáme-li funkci pojmenované parametry, nezáleží na jejich pořadí:
w = vector(length = 2, mode = "logical")
w

## [1] FALSE FALSE

# Použijeme-li ovšem k přiřazení hodnot ve špatném pořadí operátor <-, dojde k chybě.
w = vector(length <- 3, mode <- "character")

## Warning in vector(length <- 3, mode <- "character"): NAs introduced by coercion

## Error in vector(length <- 3, mode <- "character"): vector size cannot be NA/NaN
```

```
# Můžeme si ovšem všimnout, že došlo ke vzniku proměnných nazvaných length a mode:
length

## [1] 3

mode

## [1] "character"
```

Podobně komplikovaná situace existuje v oblasti indexačních operátorů, které máme k dispozici celkem tři. Jsou jimi `[]`, `[[]]` a `$`. Jejich chování je poměrně komplikované, v závislosti na tom, nad jakým datovým typem je použijeme a zda se nachází na levé či pravé straně výrazu. Podstatné je, že pro jednodimenzionální struktury není mezi `[]` a `[[]]` rozdíl, ovšem u vícedimenzionálních objektů `[]` vrací vektor nalezených prvků, zatímco `[[]]` vždy vrátí pouze jeden prvek (vektor velikosti jedna). Nakonec operátor `$` slouží k indexaci prvků seznamu či jednoho objektu (třídy) pomocí názvu tohoto prvku. Indexační operátory v R mohou sloužit jak k získání hodnoty prvku, tak i k přiřazení hodnoty či přímo části objektu.

Nakonec je ještě potřeba zmínit operátor `:`. Ten slouží k vytváření posloupností, výraz `a:b` tedy vrací vektor celých čísel mezi a a b včetně, v krocích po 1 nebo -1. Tento operátor je značně užitečný ve `for` cyklech.

2.1.4. Řídící struktury

V R se setkáme s řadou klíčových slov, která slouží k řízení toku dat v programu, a pokud máme zkušenost s jakýmkoliv jiným programovacím jazykem, nic nás zde nepřekvapí. Těmito řídícími strukturami jsou `if`, `else`, `next` (skok do dalšího kroku cyklu, obdoba rozšířenějšího `continue`), `break`, `for`, `while` a `repeat` (nekonečný cyklus). `if` a `while` přijímají jako parametr atomický logický vektor, pokud je jim předán vektor o větší délce, použijí pouze první prvek. `for` přijímá vektor a následně opakuje deklarovaný blok kódu pro každý prvek tohoto vektoru. `repeat` nepřijímá žádný parametr a skončí jedině, pokud narazí na příkaz `break`, funkci `return()` či dojde k ukončení programu.

```
if (1 <= 2) {
        print("Tato větev se vykoná.")
} else {
        print("Tato větev se nevykoná.")
}

## [1] "Tato větev se vykoná."


for (i in c(1, 2, 3)) print(2 * i)
```

```
## [1] 2
## [1] 4
## [1] 6
```

```
repeat {
        print("Nekonečný cyklus.")
        break
}
```

```
## [1] "Nekonečný cyklus."
```

Protože, jak jsme řekli, téměř vše v R jsou výrazy, jež se vyhodnocují, lze if-else konstrukt použít jako ternární operátor:

```
a = if (5 > 6) c(1, 2, 3) else c(2, 4, 8)
a
```

```
## [1] 2 4 8
```

Alternativou k těmto běžně známým řídícím strukturám jsou tak zvané funkce vyšších řádů. Jedná se o další konstrukci, kterou R přijalo od svých funkcionálních předků. Funkce vyššího řádu je taková funkce, která jako jeden ze svých parametrů přijímá jinou funkci, a tuto pak zpravidla aplikuje na vektor či seznam. [13] R obsahuje šest základních vestavěných funkcí vyšších řádů. Těmi jsou:

- `Reduce`
- `Filter`
- `Map`
- `Find`
- `Position`
- `Negate`

Nejprve si ukážeme, jak vlastně funkce vyšších řádů fungují, na příkladu použití funkce `Filter`. Ta přijímá dva parametry, funkci vracející logickou hodnotu a vektor. Vrací podmnožinu vstupního vektoru tvořenou těmi prvky, pro které vstupní funkce vrátila pravdu.

```
# Mějme vektor kladných a záporných čísel:
v = c(-1, -5, 5, -3, 0, 1, 1, -5)
# Předáme tento vektor funkci Filter spolu s funkcí,
#  která vrací TRUE pro nezáporná čísla:
w = Filter(function (x) x >= 0, v)
# Výsledkem tohoto volání je vektor nezáporných čísel:
w
```

```
## [1] 5 0 1 1
```

Funkce `Find` a `Position` fungují podobně: přijímají predikát (funkce vracející logickou hodnotu) a vektor. `Find` vrátí první prvek, který splní predikát, funkce `Position` vrátí index tohoto prvku v poli.

`Negate` přijímá pouze samotný predikát a obrací jeho hodnotu. Zkombinováním této funkce s předchozí ukázkou můžeme získat všechna záporná čísla:

```
v = c(-1, -5, 5, -3, 0, 1, 1, -5)
w = Filter(Negate(function (x) x >= 0), v)
w


## [1] -1 -5 -3 -5
```

Funkce `Reduce` je trochu komplikovanější než předchozí příklady. Přijímá binární funkci, tedy funkci se dvěma vstupními parametry, a tuto funkci rekurzivně aplikuje na jednotlivé prvky vstupního vektoru a nepovinnou počáteční hodnotu. Tím na výstupu získáme jedinou hodnotu, jež je výsledkem aplikování binární funkce mezi všemi prvky vektoru. Triviálním příkladem je implementace matematických operací sumy či produktu:

```
sum = Reduce(`+`, c(1, 2, 3, 4, 5))
sum


## [1] 15


prod = Reduce(`*`, c(2, 10, 5))
prod


## [1] 100
```

Nakonec, `Map` pouze aplikuje funkci na jednotlivé prvky vektoru. Stejnou funkcionalitu nabízí i rodina funkcí *apply, jedná se o funkce `lapply`, `sapply` a `vapply`, lišící se především v typu návratové datové struktury.

```
# Map vrací seznam, použijeme funkci unlist na konverzi do vektoru:
a = unlist(Map(function (x) x ** 2, c(1, 2, 3, 4)))
a


## [1]  1  4  9 16


# Sapply defaultně vrací vektor, oproti Map má prohozené argumenty:
b = sapply(c(1, 2, 3, 4), function (x) x ** 2)
b


## [1]  1  4  9 16
```

V R je preferováno použití funkcí vyšších řádů místo klasických, imperativních řídících struktur. Je ovšem potřeba dbát na čitelnost kódu a vždy volit vhodný nástroj podle specifického problému, který řešíme.

2.1.5. Vestavěné a uživatelské funkce

Jak je u programovacích jazyků zvykem, R nabízí nejen primitivní struktury a jednoduché řídící mechanismy, ale také bohatou škálu vestavěných funkcí. K dispozici máme samozřejmě množství funkcí pro manipulaci s objekty a proměnnými v dané relaci. Například funkce pro kontrolu typů (`class()`, `mode()` a `typeof()`) a konverzi mezi nimi (`as.vector()`, `as.matrix()`...) či zjišťování a manipulaci s vlastnostmi objektů (`length()`, `ncol`...). Velmi užitečnou funkcí je (`help()` (či ve zkrácené formě (`?`), která slouží k zobrazení nápovědy přímo v prostředí R.

R je běžně prezentováno jako prostředí pro statistické výpočty, nepřekvapí tedy, že krom běžných jazykových konstrukcí obsahuje v základu také mnoho funkcí implementujících statistické algoritmy. Dále je k dispozici velký počet rozšiřujících knihoven, které lze stáhnout a nainstalovat funkcí `install.packages()`. K funkcím obsaženým v knihovně lze přistupovat přes operátor čtyřtečky, spojující název knihovny a funkce (např. `jsonlite::fromJson()`). Alternativně lze veškerý obsah knihovny načíst do prostředí prostřednictvím funkce `library()`, která přijímá jako vstupní parametr název knihovny.

Použití funkcí již bylo předvedeno v předchozích sekcích. Za názvem funkce se nacházejí jednoduché závorky, ve kterých předáváme jednotlivé parametry. Parametry lze předávat pozičně či podle názvu, navíc lze oba přístupy kombinovat. K předání parametru slouží operátor `=`, operátor `<-` vytvoří novou proměnnou a tu předá pozičně.

```
# Toto jsou validní způsoby volání funkce vector:
v = vector("integer", 5)
v = vector(length = 5, mode = "integer")
v = vector(5, mode = "integer")
v = vector("integer", length = 5)
v = vector(l = 5, m = "integer")
```

Samozřejmě je možné také vytvářet vlastní funkce. Jak bylo zmíněno výše, R je silně inspirováno funkcionálním pohledem na programování. V důsledku toho se nijak neliší deklarování funkce, uložení funkce do proměnné či anonymní funkce. K jejich vytváření slouží klíčové slovo `function`, za kterým v závorkách následují parametry. Chceme-li některému parametru nastavit výchozí hodnotu, použijeme opět `=`, jako bychom předávali pojmenovaný parametr při volání funkce. Vyhodnocení takového výrazu nám vrátí funkci (v R nazývaný closure neboli uzávěr, ačkoli ne všechny funkce jsou uzávěry). Tu můžeme přiřadit proměnné či použít jako anonymní funkci (například jako parametr funkci vyššího řádu). Ačkoli v R existuje klíčové slovo `return()` k vrácení výsledku z funkce, není nutné jej vždy použít, standardně je hodnota funkce rovna poslednímu vyhodnocenému výrazu v těle funkce.

```
# Vytvoření pojmenované funkce:
f = function(x) x+1
f(5)
```

```
## [1] 6

# Ekvivalentní funkce:
f = function(x) return(x+1)
f(5)

## [1] 6

# Funkce s výchozí hodnotou:
f = function(x = 0) return(x+1)
f()

## [1] 1

# Použití anonymní funkce:
v = sapply(c(1,2,3), function(x) 2*x)
v

## [1] 2 4 6
```

2.2. knitr

Knitr je knihovna pro dynamické generování dokumentů. K tomuto účelu kombinuje skriptovací a značkovací jazyky. Ačkoli nejpopulárnější volbou je kombinace R a *markdownu*, *knitr* sám podporuje více jazyků. Nepřekvapí podpora pro python, perl či ruby, ale je možné použít například také C, C++ či Fortran. Výstupem mohou být dokumenty ve značkovacích jazycích, jako jsou markdown, LaTeX, HTML, AsciiDoc či RST, které pak lze externími nástroji (např. pandoc) převést do prakticky libovolného jazyka či formátu.[14]

Práce s touto knihovnou funguje tak, že na vstupu máme textový dokument v některém ze dvou výše zmíněných formátů - markdownu či TeXu – obsahující tak zvané *chunky*, neboli úryvky kódu, v některém z mnoha podporovaných jazyků. Při kompilaci zdrojového souboru *knitrem* jsou tyto úryvky kódu vyhodnoceny stejně, jako by se jednalo o běžný, samostatný program v tomto jazyce. Přitom si pro každý samostatný *chunk* můžeme zvolit, zda má být zdrojový kód, výstup, nebo obojí vidět ve výsledném dokumentu. *Knitr* se postará o formátování kódu a výstupu podle pravidel výsledného jazyka (odsazení, barevné zvýraznění, grafy a tabulky...) a vytvoří dokument, který je již pouze čistý TeXči markdown (který pak můžeme dále upravovat).

Knihovnu *knitr* vyvinul čínský statistik Yihui Xie. Je částečně založena na starším balíčku *Sweave*. Nejzásadnějším zlepšením oproti *Sweave* je modulární návrh pro snažší údržbu, vývoj a rozšiřování. *Sweave* se dá vnímat jako podmnožina *knitru*.

Jak bylo zmíněno výše, *knitr* podporuje velké množství jazyků. Ovšem tím primárním je

R, což s sebou přináší jistá nepatrná omezení, pokud chceme použít některý z ostatních – například nemožnost přenášet objekty mezi jednotlivými úryvky kódu (*chunky*), což je dáno tím, že (pokud se nejedná o R) jsou jednotlivé úryvky vyhodnocovány samostatně. Z tohoto důvodu se dá doporučit použití *knitru* primárně v kombinaci s R, pokud takovou možnost máme. Stejně tak můžeme generovat dokumenty ve velkém množství značkovacích jazyků či formátů. Po zbytek této práci budu popisovat použití *knitru* v kombinaci R + TEX, ale použití jakékoliv jiné kombinace je stejně přímočaré (byť pochopitelně s odlišnou syntaxí). Velkou výhodou této knihovny je, že pokud máme zkušenosti s jazyky, které se rozhodneme použít, *knitr* sám pouze přidává malé množství syntaxe pro jejich propojení a nepřináší žádné výrazné komplikace. Ačkoli disponuje bohatou škálou voleb, jejichž prostřednictvím můžeme modifikovat jeho chování, není nezbytně nutné je znát, dokud je nepotřebujeme použít.

Nejdůležitější syntaktickou konstrukcí *knitru* je tak zvaný *chunk*, umožňující vložit část zdrojového kódu do textového dokumentu. Pokud je textový dokument psán v TEXu (jako je tomu v případě této práce), vypadá *chunk* takto:

```
<<knitr-example-01, eval = TRUE, echo = FALSE, include = TRUE>>=

@
```

Chunk začíná dvojicí špičatých závorek, uvnitř kterých se nachází nejprve jedinečný název konkrétního úryvku kódu (v ukázce výše *knitr-example-01*), dále pak parametry ovlivňující jakým způsobem *knitr* daný úryvek zpracuje. V příkladu výše jsou těmito parametry *eval = TRUE*, *echo = FALSE* a *include = TRUE*. První z těchto parametrů říká, že kód má být vyhodnocen, druhý parametr zadává, že zdrojový kód nemá být vložen do výstupního dokumentu. Třetí parametr pak zajišťuje, že *výstupy* vyhodnocení tohoto kódu, to znamená například grafy, *budou* zahrnuty do výsledného dokumentu. Po parametrech následuje dvojice uzavírajících špičatých závorek a rovnítko. Vše, co následuje za tímto řádkem, je chápáno a vyhodnoceno jako zdrojový kód psaný v R (pokud parametr *engine* neuvádí jinak), dokud se nedojde ke znaku @, který ukončuje daný úryvek.

2.3. Shiny

Shiny je framework pro tvorbu webových aplikací postavený na R. Je určený primárně k prezentaci dat, přičemž umožňuje jejich zkoumání pomocí bohatého interaktivního prostředí. Typickým příkladem použití může být webová aplikace zobrazující různé vizualizace dat (především grafy, tabulky, obrázky a textové výstupy), kde si uživatel může snadno upravovat jednotlivé vstupní parametry. Veškeré výstupy přitom v reálném čase reagují na změny, takže je možné pozorovat, jak jakákoliv změna na vstupu ovlivňuje výstup.

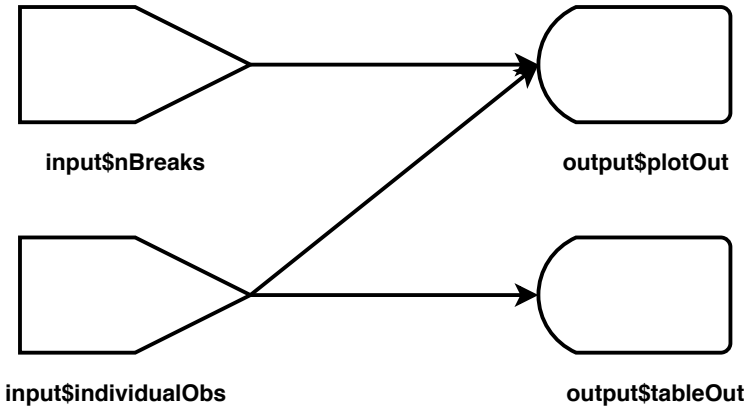
Díky Shiny se dá vytvořit plnohodnotná a komplexní webová aplikace čistě v R, bez potřeby znalosti HTML, CSS, JavaScriptu či dalších tradičních webových technologií. V R je možné implementovat klientskou i serverovou logiku, uživatelské rozhraní – defaultně založené na CSS knihovně Bootstrap, s možností alternativních témat v podobě rozšiřujících balíčků – přičemž máme k dispozici veškerou funkcionalitu a knihovny základního R. Navíc je k dispozici bohaté množství knihoven určených přímo pro integraci se Shiny, přidávající nové formy vizualizace (například balíčky pro tvorbu grafů *ggPlot* a *plotly* nebo widget přidávající interaktivní mapy *leaflet*), celé formy uživatelského rozhraní (*shinydashboard*) nebo rozšiřu-

jící základní funkcionalitu Shiny o funkce z jiných jazyků (knihovny *shinythemes*, *shinyjs* či *htmlwidgets*). Pouze popsat všechna rozšíření, která jsou pro tento framework k dispozici, je úkol nad rámec této práce. Ačkoliv je Shiny primárně určeno k prezentaci statistických dat a jedná se o oblast, pro kterou lze najít nejvíce návodů a nástrojů, není v žádném případě omezeno pouze pro tento účel. Stejně jako je R v principu univerzální programovací jazyk, který nemusí být použit pouze pro statistické výpočty, lze v Shiny tvořit libovolné webové aplikace.

Ačkoli nám Shiny dává možnost vytvořit celou aplikaci pouze v R, ani v tomto směru nám neklade žádné omezení. Shiny aplikace se dají snadno propojit s uživatelským rozhraním psaným v HTML, mohou využívat CSS stylů (ať už embedovaných přímo v R kódu, čtených ze souborů na lokálním disku nebo z webových zdrojů) a JavaScriptových funkcí (a to bez použití pomocných knihoven, které jen usnadňují volání těch nejpoužívanějších).

Reaktivita

Programátorské rozhraní Shiny využívá paradigmatu reaktivního programování. Celý program se skládá z tak zvaných reaktivních výrazů. Přitom každý reaktivní výraz si „pamatuje“ výrazy, které používá pro výpočet své hodnoty, a pokud v některém z těchto výrazů (v nomenklatuře Shiny nazývaných *zdroje*) dojde ke změně, všechny závislé (nazývané *endpointy*) výrazy automaticky – pokud jim neřekneme jinak – znovu vypočtou svou hodnotu. [15] Při tvorbě Shiny aplikace nám tak odpadá psaní velkého množství balastního kódu, který má za úkol pouze obsluhovat události a propagovat změny skrze složitý systém navzájem závislých jednotek. To vše obstará samotný framework, a my se můžeme soustředit především na kód, který je jedinečný pro naši aplikaci.



Obrázek 2.1.: Diagram vztahů mezi reaktivními výrazy v jednoduché Shiny aplikaci – reprodukováno podle [15]

Zdroji změn jsou zpravidla widgety uživatelského rozhraní. Jako endpointy figurují výstupy, například grafy, texty či tabulky. Nejedná se o neporušitelné pravidlo, pouze o běžný přístup, jakým je Shiny aplikace strukturována. Navíc zdroje a endpointy nemusejí být propojeny přímo, mezi libovolným zdrojem a endpointem se může nacházet neomezené množství dalších reaktivních výrazů (nazývaných konduktory), které mohou sloužit ku příkladu jako cache či mezikroky při výpočtu dat. Za určitých okolností ovšem není reaktivita žádoucí, třeba

pokud máme dlouhotrvající výpočet, můžeme chtít, aby se spustil pouze po kliknutí na tlačítko a nikoli při jakékoli změně jakéhokoli vstupního parametru. Případně potřebujeme, aby výraz reagoval na změny na jednom vstupu, ale využíval hodnoty z více. I pro tyto případy nám Shiny nabízí potřebné nástroje.

```
# Ukázka jednoduché Shiny aplikace:
library(shiny)

ui = basicPage(
  sliderInput("obs", "Observations", min = 0, max = 1000, value = 100),
  plotOutput("plot")
)

server = function(input, output) {
  output$plot = renderPlot({
    hist(rnorm(input$obs))
  })
}

runApp(appDir = shinyApp(ui, server))
```

Jednoduchá Shiny aplikace se skládá ze dvou částí, objektu nazvaného *ui*, který obsahuje uživatelské rozhraní, a funkce *server*, která zajišťuje veškerou logiku. Pro vytváření uživatelského rozhraní máme v Shiny k dispozici širokou škálu funkcí, které je možné libovolně skládat a vrstvit, přičemž tyto funkce obstarávají generování stránek, layoutů a jednotlivých widgetů. Funkce *server* přijímá parametry *input* a *output*, což jsou seznamy obsahující hodnoty vstupních a výstupních výrazů, volitelně také parametr *session* obsahující informace a funkce pro manipulaci s aktuální relací.

3. Zadání úloh a popis použitých algoritmů

Hlavním cílem této práce je popsat možnosti využití nástrojů *Shiny* a *knitr*, společně se samotným jazykem R, pro vizualizaci dat, algoritmů a výsledků experimentů či měření. Bylo vybráno několik úloh z disciplín jako analýza dat, numerická matematika, výpočetní geometrie či fyzikální simulace. Tyto úlohy slouží jako zdroj dat, která následně vizualizujeme pomocí popisovaných nástrojů.

Tato sekce obsahuje teoretický popis jednotlivých úloh. Důraz je kladen především na jejich algoritmizaci, nezbytnou pro možnost jejich realizace. Ačkoli v některých případech můžeme zcela (k-means) nebo alespoň částečně (Delaunayova triangulace) použít funkce přímo vestavěné v R či rozšiřující knihovny, v některých případech je nejsnazším řešením problému vlastní implementace. V obou případech bychom ovšem měli mít znalost o tom, jak tyto algoritmy fungují.

Přinejmenším nastíněno bude i využití těchto metod k řešení praktických problémů, neboť ačkoli vizualizace je hlavním tématem této práce, tyto algoritmy neexistují primárně za účelem vytváření pěkných grafů.

3.1. K-means

K-means (k-průměry) je metoda datové analýzy, jejímž cílem je rozdělení bodů v prostoru do k skupin, kde každý bod je přiřazen do skupiny, jejíž střed se nachází (Euklidovsky) nejblíže. Obecněji řečeno, k-means je nástrojem shlukové analýzy, jejímž účelem je účelem nalezení struktury v datech, pokud danou strukturu předem neznáme. Výstupem algoritmu je rozdělení bodů na k disjunktních množin, které tvoří Voronoiův diagram. [16]

Jedná se o nedeterministicky polynomiální problém, avšak existuje řada heuristických algoritmů, jejichž složitost je v praktických případech lineární. Tyto algoritmy sice nezaručují nalezení skutečného optima a jejich výstup může být odlišný při opakování výpočtu. Běžným postupem je tedy vícenásobné spuštění algoritmu nad stejným datasetem a porovnání výsledků.

Algoritmy

Nejpoužívanějším algoritmem pro řešení k-means je Lloydův algoritmus. Pro výběr počátečních středů je možné použít dva různé postupy. Můžeme zvolit náhodně k bodů z datasetu a označit je za počáteční středy (Forgy). Tento postup běžně vede k rozmístění středu daleko

od sebe. Alternativně můžeme každému bodu náhodně přiřadit některý z k klastrů, čímž získáme středy, které se nacházejí blízko sebe. Následně aplikujeme iterativní postup, během kterého dochází ke střídání dvou kroků:

1. Prvním krokem je přiřazení jednotlivých bodů do odpovídajících klastrů. Každý klastr je definován geometrickým středem. Klastr, do kterého má být bod přiřazen, je určen druhou mocninou Euklidovské vzdálenosti, tedy bod je přiřazen do skupiny, jejíž geometrický střed se nachází nejblíže. Každý bod náleží vždy právě do jedné skupiny.
2. Druhým krokem je aktualizace pozic center. V tomto kroku je pro každou skupinu vypočten nový geometrický střed z polohy všech bodů, které byly klastru přiřazeny v předchozím kroku.

Tento cyklus končí v momentě, kdy druhý krok nezpůsobí změnu pozice žádného středu. Jedná se o poměrně rychlý algoritmus pro data, která jsou skutečně organizována do oddělených množin. Lloydův algoritmus zaručuje terminaci, tedy ukončení cyklu v nějakém optimálním řešení, ovšem může se jednat o lokální optimum.

R obsahuje funkci `kmeans`, jež implementuje několik různých algoritmů pro řešení tohoto problému (a přijímá parametr určující, který algoritmus má být použit). Jedná se o algoritmy nazvané po svých autorech Lloyd, Forgy, MacQueen a Hartigan-Wong. Dokumentace nás upozorňuje, že Lloyd a Forgy jsou dva názvy pro totožný algoritmus. MacQueen opakovaně posouvá středy klastrů na střed jejich odpovídajících Voronoiových buněk. [18]

Jako výchozí algoritmus pro výpočet k-means je v R použit Hartigan-Wong, neboť ve většině případů funguje lépe než dříve zmíněné postupy. Tento algoritmus na rozdíl od předchozích sám neurčuje výchozí pozice středů klastrů (ačkoli autoři uvádějí možný postup, jak tyto pozice získat), ale přijímá je jako jeden ze vstupních parametrů (zatímco ostatní algoritmy pouze přijímají parametr k). [19]

Jeden iterační krok Hartiganova-Wongova algoritmu vypadá takto: [17]

1. Je zvolen náhodný bod x z náhodného klastru C.
2. x je považován za samostatný (jednoprvkový) klastr s geometrickým středem v_x.
3. Je nalezen nový střed klastru C na základě pozic všech bodů x', kde $x' \neq x$.
4. Pro bod x je nalezen nový klastr C^* tak, aby bylo minimalizováno D. (D je obecně libovolná cenová funkce, zpravidla Euklidovská vzdálenost jednotlivých bodů od středů jejich klastrů.)

Algoritmus končí v okamžiku, kdy pro všechna x platí $C = C^*$.

Jednou z hlavních nevýhod k-means je, že algoritmus přijímá počet k klastrů jako vstupní parametr. To je problém, pokud chceme tuto metodu použít na odhalení struktur ve zkoumaných datech, přičemž neznáme nejen tvar těchto struktur, ale ani jejich počet, což je při použití k-means běžný případ. Metrikou, jež je často používána pro získání vhodného k, je střední vzdálenost bodů od geometrického středu jejich klastru. Je zde ovšem problém, neboť se zvyšujícím se k tato veličina vždy klesá, až dosáhne nuly v případě, kdy je k rovno počtu bodů. Jednou z používaných technik je vykreslení grafu funkce této hodnoty v závislosti na k a následné nalezení zlomu, ve kterém dochází k prudkému poklesu vzdáleností. [20]

3.2. Word cloud

Word clouds, také nazývané *tag clouds*, jsou nástrojem pro vizualizaci frekvence výrazů v textu. Základní princip je jednoduchý: čím častěji se výraz (zpravidla jedno slovo) vyskytuje ve zdrojovém textu, tím větší bude ve výstupní vizualizaci. Navíc mohou být ovlivněny další parametry, jako font či barva. *Tag cloudy* jsou spojovány s Webem 2.0. Často se vyskytují jako nástroj pro třídění obsahu a navigační pomůcka. Odtud se následně rozšířily jako univerzální nástroj pro vizualizaci libovolného textu. [21]

Ačkoli se jedná o poměrně populární a snadnou techniku, existují rizika, která mohou zkreslit výsledek. Jednou z hlavních nevýhod je zdůraznění frekvence slov, nikoli jejich skutečné důležitosti. Jinými slovy je zde implicitní předpoklad o vztahu mezi četností výskytu výrazu a jeho významu v textu, tento předpoklad ovšem nemusí být platný.

Další rizika souvisejí s nutností úpravy vstupních dat. Je vhodné upravit kapitalizaci slov (převést všechny výrazy do tvaru tvořeného jen velkými či malými písmeny), transformovat slova do základního tvaru a odstranit výrazy, které nechceme, aby se vyskytovaly ve výstupu (zpravidla spojky, členy a podobně). Pokud tyto úpravy neprovedeme, může se stát, že výstup bude obsahovat slova nenesoucí žádný speciální význam na úkor těch zajímavých. Případně může dojít k tomu, že se slovo, které se ve vstupním textu vyskytuje často, ve výstupu neobjeví, nebo bude označeno jako mnohem méně populární, v důsledku různých tvarů a různé kapitalizace. Podobně pokud je pro nějaký výraz použito mnoho synonym, ovlivní to přesnost výstupu, to je ovšem problém, který se při automatizaci obtížně řeší.

3.3. Schulzeho metoda

Schulzeho metoda (také *beatpath method*) je volební systém, který v roce 1997 popsal Markus Schulze. Jedná se o tak zvanou Condorcetovu metodu, to znamená, že vítězem je takový kandidát, který by vyhrál oproti každému jinému kandidátovi, pokud by se rozhodovalo jen mezi těmito dvěma.[22, 23]

V současné době se jedná o nejrozšířenější z Condorcetových metod. Ve svých volbách ji používá množství organizací, mimo jiné Švédská Pirátská Strana, Wikimedia Foundation, Debian, Ubuntu, Gentoo či KDE. Adopce na národní úrovni se ovšem jeví nepravděpodobná.

Algoritmy

V tomto volebním systému voliči seřadí jednotlivé kandidáty podle pořadí, v jakém je preferují. Přitom volič smí jednak přiřadit více kandidátům stejnou hodnotu (pokud je preferuje stejně) a také některým kandidátům nepřiřadit žádné pořadí – takový kandidáti jsou považováni za rovnocenné a umístěni za všechny ohodnocené kandidáty.

Vstupem pro výpočet je pak seznam všech možných kombinací kandidátů, každá z těchto kombinací ohodnocena počtem voličů, kteří ji zvolili. Následně je pro každou dvojici kandidátů a a b vypočtena hodnota vítězných hlasů, což je počet voličů, kteří preferují a před b. Toto číslo získáme sečtením všech hlasů pro všechna uspořádání, ve kterých platí, že je a preferován před b.

Tím je vytvořen orientovaný graf, kde jsou kandidáti reprezentováni vrcholy a párová porovnání tvoří orientované hrany mezi vrcholy. Každá orientovaná hrana (spojující každý pár kandidátů) je ohodnocena počtem vítězných hlasů (hodnoty vypočtená výše).

V tomto grafu je pojmem *beatpath* označená libovolná cesta po směru hran spojující libovolné kandidáty a a b. Hodnota *beatpath* odpovídá nejnižší hodnotě hrany vyskytující se v této cestě. Pokud je hodnota cesty od a k b vyšší nebo alespoň rovna hodnotě cesty od b k a, je a preferovaným kandidátem před b.

Vítězem v této metodě je takový kandidát a, pro kterého předchozí tvrzení platí pro všechny ostatní kandidáty b. Schulze dokázal, že takový vítěz vždy existuje, ačkoli může dojít k remíze. Tato metoda může sloužit nejen k určení jednoznačného vítěze, ale také k seřazení všech kandidátů podle preferencí.[24]

Existuje alternativní postup řešení, vhodný pro grafickou demonstraci algoritmu. V tomto postupu vytvoříme tabulku párových preferencí tak jako předtím, převedeme ji do tvaru rozdílů, to znamená, že buňka pro kandidáty a a b bude číselně rovna počtu hlasů, o kolik kandidát a vyhrál či prohrál (v takovém případě bude hodnota záporná) oproti kandidátovi b. Toho dosáhneme tak, že od tabulky párových preferencí odečteme její transponovanou formu. Abychom nalezli vítěze, opakujeme dva kroky:

1. Pokud se v některém řádku nenachází žádné záporné číslo, je kandidát v tomto řádku vítězem.
2. Hodnotu v buňce s nejvyšším záporným číslem nahradíme nulou či jinak označíme jako neplatnou.

Pokud chceme tímto postupem uspořádat všechny kandidáty, po nalezení prvního vítěze odstraníme sloupek a řádek tabulky odpovídající tomuto kandidátovi a postup opakujeme.

3.4. Numerické metody řešení nelineárních rovnic

Jedná se o klasickou matematickou úlohu. Úkolem je nalézt takovou hodnotu x (či více hodnot), ve kterých nabývá spojitá funkce $f(x)$ hodnoty 0. Jelikož platí, že je řešení rovnice $f(x) = g(x)$ je ekvivalentní hledání kořenu funkce $h(x) = f(x) - g(x)$, umožňují nám tyto algoritmy řešit libovolnou rovnici, kterou lze vyjádřit spojitou funkcí. Přitom je ovšem nutno brát v úvahu omezení existujících numerických algoritmů, jelikož nezaručují konvergenci a nalezení kořene a vracejí pouze aproximaci skutečné hodnoty x.

Existující algoritmy se liší především rychlostí konvergence. Navíc mohou klást odlišné požadavky na funkci a interval, ve kterém se snažíme kořen nalézt, například požadavek na existenci právě jednoho kořene v intervalu či na spojitost první derivace dané funkce. Všechny níže popsané algoritmy využívají iterace, produkují tedy posloupnost hodnot, jež by měla konvergovat ke skutečné hodnotě kořene. Iteraci zpravidla ukončíme, je-li dosaženo dostatečně přesného výsledku (rozdíl mezi hodnotami x_i a x_{i+1} je menší než zvolená hodnota p) nebo pokud je překročen maximální počet iterací. [25]

3.4.1. Metoda bisekce

Nejznámějším numerickým postupem pro hledání kořene funkce je metoda bisekce (také nazývána metoda půlení intervalu). Jedná se o jednoduchý algoritmus, jehož nevýhodou je ovšem poměrně pomalá konvergence. Navíc na výstupu vyžaduje definici intervalu, ve kterém hledání kořene probíhá, a je nutné, aby se na tomto intervalu kořen nacházel právě jednou. Postup je následující:

1. Mějme funkci $f(x)$, pro kterou chceme nalézt kořen $f(x) = 0$ na intervalu $[a, b]$.
2. Nalezneme prostřední bod intervalu $c = (a + b)/2$.
3. Spočteme funkční hodnotu $f(c)$.
4. Nahradíme některý z okrajů intervalu a či b za c tak, aby se kořen stále nacházel uvnitř nového intervalu (musí být splněna podmínka $f(a)\, f(b) < 0$).

3.4.2. Regula falsi

Nevýhodou metody půlení intervalu je pomalá konvergence. Funkce $f(x)$ neslouží k určení polohy nového bodu c – ten se vždy nachází přesně v polovině intervalu – pouze ke zvolení, kterou polovinu vybrat. Metoda regula falsi volí jako polohu bodu c průsečík spojnice bodů a a b s osou x. Předpis pro nalezení nové hodnoty c je následující: [26]

$$c = b - f(b)\frac{b - a}{f(b) - f(a)}$$

Po vypočtení c je určen nový interval stejně jako u metody bisekce.

3.4.3. Newtonova metoda

Newtonova metoda, také nazývána metodou tečen, zpravidla konverguje k řešení rychleji, než výše uvedené postupy, ovšem vyžaduje existenci spojité derivace zkoumané funkce. Vstupem pro tento algoritmus je zkoumaná funkce a počáteční odhad pro x, přičemž může nastat situace, že Newtonova metoda nebude konvergovat k řešení, zpravidla pokud je počáteční odhad příliš vzdálen od skutečného řešení. Metodu tečen lze vyjádřit vztahem:

$$x_i = x_{i-1} - \frac{f(x_{i-1})}{f'(x_{i-1})}$$

3.4.4. Metoda sečen

Metodu sečen obdržíme, pokud derivaci v Newtonově metodě nahradíme konečným diferenciálem. Nevyžaduje existenci spojité derivace a zpravidla konverguje rychleji, než metoda půlení intervalu, ovšem stejně jako u metody tečen může nastat případ, kdy ke konvergenci nedochází. Vstupem je funkce a dvě počáteční hodnoty x_i a x_{i+1} definující interval, na kterém hledáme řešení. Metodu sečen popisuje následující rovnice:

$$x_i = x_{i-1} - f(x_{i-1})\frac{x_{i-1} - x_{i-2}}{f(x_{i-1}) - f(x_{i-2})}$$

3.4.5. Zjištění počátečního odhadu

Všechny výše popsané metody vyžadují nějaký počáteční odhad x, či interval $[a, b]$, na kterém má být exaktní řešení hledáno. Určení této počáteční hodnoty je netriviální problém, přičemž špatný odhad může vést k pomalé konvergenci či zcela konvergenci zcela zabránit.

Jednou možností pro získání počátečního odhadu je nakreslení grafu funkce $f(x)$ a vyhledání průsečíků s osou x. Alternativně můžeme sestavit tabulku funkčních hodnot $[x_i, f(x_i)]$ pro nějaké dělení intervalu $[a, b]$. Pokud ve dvou sousedních bodech tabulky nabývá funkční hodnota $f(x)$ hodnot s opačným znaménkem, leží v tomto intervalu reálný kořen $f(x) = 0$.

3.5. Detekce rozhraní pomocí Delaunayovy triangulace

Cílem této úlohy je identifikovat rozhraní (např. mezi kapalinou a plynem, či dvěma neslučitelnými kapalinami) v množině bodů. Vzhledem k všudypřítomnosti takovýchto rozhraní v přírodě existuje vysoký zájem o techniky pro jejich zkoumání, současně se ovšem jedná o obtížně řešitelný problém. Bylo vyvinuto několik postupů, ačkoli se stále jedná o otevřenou oblast výzkumu. Jedním z řešení je detekce rozhraní založená na Delaunayově triangulaci.

Triangulací rozumíme rozdělení bodů v rovině do trojúhelníků různých velikostí, přičemž zpravidla vyžadujeme, aby sousední trojúhelníky sdílely hrany a vrcholy. Vzhledem k této volné definici existuje široká škála různých metod pro triangulaci. Specificky Delaunayova triangulace dělí body takovým způsobem, aby žádný neležel uvnitř opsané kružnice jakéhokoliv z vytvořených trojúhelníků. To vede k maximalizaci minimálních úhlů, vytvořené trojúhelníky se tak co možná nejvíce blíží rovnostranným. Delaunayova triangulace je duálním grafem Voronoiova diagramu.

Pro detekci rozhraní si stanovíme si parametr T. Označme jako c poloměr opsané kružnice trojúhelníku vytvořeného Delaunayovou triangulací. Pak za součást rozhraní budeme považovat každou hranu sdílenou dvěma trojúhelníky, pokud pro právě jeden z těchto trojúhelníků platí $c \leq T$ a současně pro druhý $c > T$.

Vystává otázka, jakým způsobem volit hodnotu T. Jednou možností je postupné zvyšování T, dokud nezískáme dobrý výsledek. Metodičtějším řešením je vzestupné seřazení poloměrů všech opsaných kružnic a jejich vykreslení do grafu – pokud v takovém grafu nalezneme jeden nebo více zlomů, jsou hodnoty v jejich okolí vhodnými kandidáty na T. [27]

3.6. Isingův model

Isingův model je standardní model feromagnetismu v oblasti statistické mechaniky. Feromagnetismus nastává, když se velké množství atomů otáčí se stejným spinem. Vyvstává tedy otázka, jak mohou vzdálené atomy znát svůj spin, když reagují jen se sousedními částicemi

skrze lokální interakce. Isingův model byl vytvořen za účelem studování těchto interakcí fyzikem Wilhelmem Lenzem, který jej zadal svému studentovi Ernestu Isingovi. Ising nalezl analytické řešení pro jednorozměrný případ modelu v roce 1924. Analytické řešení existuje také pro dvourozměrný Isingův model bez vnějšího magnetického pole. Pokud ovšem chceme modelovat účinky vnějšího pole, či zkoumat Isingův model ve vyšších dimenzích, je třeba použít numerické řešení.[28]

Isingův model je tvořen magnetickými spiny uspořádanými v mřížce s periodickými okrajovými podmínkami. Každý spin reprezentuje místní magnetický moment, přičemž uvažujeme pouze dva možné stavy, ve kterých se spin může nacházet: +1 (spin nahoru) a -1 (spin dolů). Celková energie systému je vyjádřena vztahem [29]

$$E = -J \sum_{\langle i,j \rangle} \sigma_i \sigma_j - H \sum_i \sigma_i$$

Zápis $\langle i, j \rangle$ značí sumu přes každý pár sousedících spinů $\sigma_i \sigma_j$. Energie systému se skládá ze dvou složek: interakce mezi sousedícími spiny a efektu vnějšího magnetického pole na jednotlivé spiny. Interakce mezi sousedícími spiny způsobuje homogenní uspořádání, tudíž je vyhodnocena záporně, pokud se oba spiny nacházejí ve stavu +1 či -1, a naopak kladně pokud spin +1 sousedí se spinem -1. J je pozitivní koeficient síly interakce mezi sousedícími spiny. H je síla vnějšího magnetického pole, pokud je $H > 0$, spiny mají tendenci uspořádávat se ve stavu +1, pokud platí $H < 0$, spiny gravitují ke stavu -1. Často je uvažován Isingův model bez vnějšího magnetického pole, pak je energie systému vypočtena jako

$$E = -J \sum_{\langle i,j \rangle} \sigma_i \sigma_j$$

Přestože byl Isingův model původně vytvořen pro zkoumání magnetismu, nachází dnes uplatnění v mnoha oborech, od fyzikálních simulací po ekonomické modely. Nejznámějším alternativním využitím Isingova modelu je simulace fázových přechodů mezi kapalinami a plyny. V takovémto modelu hodnota spinu reprezentuje přítomnost či nepřítomnost částice v dané poloze. Dvourozměrný Isingův model je nejjednodušším statistickým modelem, který umožňuje vyjádřit fázový přechod.

Algoritmy

Jak bylo uvedeno výše, pro případy jednorozměrného modelu a dvourozměrného modelu bez vnějšího pole existují analytická řešení. Ostatní případy lze řešit pouze numerickými metodami. Nejpoužívanějším postupem pro numerické řešení Isingova modelu je Metropolisův-Hastingsův algoritmus. Jeden krok této simulace vypadá takto:

1. Zvolíme náhodný spin σ.
2. Spočteme energii E podle vzorce uvedeného výše, přičemž bereme v úvahu externí magnetické pole a interakce se čtyřmi sousedními spiny.
3. Pokud $E > 0$, převrátíme hodnotu spinu.
4. Pokud $E < 0$, převrátíme hodnotu spinu s pravděpodobností $1 : e^{\frac{2E}{T}}$, kde T je teplota.

Tento postup je opakován pro zvolený počet kroků či dokud není dosaženo termální rovnováhy. Nevýhodou tohoto algoritmu je, že pokud zvětšíme velikost mřížky za účelem redukce vlivu diskrétního charakteru modelu, celá simulace se výrazně zpomalí, neboť je potřeba většího počtu kroků pro dosažení rovnovážného stavu.

Tento problém řeší algoritmy Swendsenův-Wangův a Wolffův. Tyto postupy jsou nelokální, to znamená, že v jednom kroku není převrácena hodnota pouze jednoho spinu, ale nějakým způsobem definované skupiny provázaných spinů, např. celý klastr spinů se stejnou hodnotou.

3.7. Molekulární dynamika systému tuhých disků

Pojmem *molekulární dynamika* označujeme metodu počítačové simulace pohybu atomů a molekul. Jedná se o deterministickou metodu, což znamená, že počáteční stav systému – počet, poloha a rychlosti jednotlivých částic – jednoznačně určuje veškerý následný vývoj systému. Tato simulace zpravidla využívá zákonů klasické mechaniky, především Newtonových pohybových rovnic. Systém je tvořen n disky reprezentujícími jednotlivé částice uvnitř uzavřené nádoby, přičemž každý disk se pohybuje rovnoměrným pohybem rychlostí v. Rychlost disku může být změněna pouze kolizí (neboli srážkou) s jiným diskem či stěnou nádoby. Kolize dvou částic je aproximována jako dokonale pružný ráz a nové rychlosti jsou vypočteny podle odpovídajících rovnic. Částice na sebe vzájemně působí pouze kolizemi.

Ačkoli se jedná o jednoduchý model, má molekulární dynamika značné využití. Především se jedná o úlohy z oblasti statistické mechaniky, kde je cílem popsat vztahy mezi makroskopickými proměnnými (teplota, tlak) a mikroskopickými ději. Stejné techniky nalézají své využití také v dalších oblastech, kde dochází k využití počítačů k simulaci částicových systémů, jako jsou počítačová grafika a hry.

Komplexnější modely pak nalézají své uplatnění mimo jiné v materiálové vědě, kde jsou použity ke zkoumání defektů (bodové, lineární a plošné defekty), lomů či proměn povrchů (povrchové tavení, fazetový růst, zdrsňování). Své uplatnění nalézají i při popisu biomolekul (rozsáhlé makromolekuly, proteiny, DNA, RNA) či fázových přechodů a jiných teplotně závislých jevů. [30]

Algoritmy

Částice jsou aproximovány jako disky pohybující se ve dvourozměrném prostoru a interagující mezi sebou navzájem a stěnami ohraničujícím prostor skrze elastické kolize. Problém lze řešit dvěma přístupy: [31]

1. *Časem řízená simulace:* v tomto přístupu je čas kvantifikován na uniformní díly velikosti dt. Stav systému je aktualizován v každém kroku velikosti dt v závislosti na vektorech (směru a velikosti) polohy a rychlosti každé částice. Jedná se o jednoduchý postup, má ale dvě nevýhody. Za prvé, v každém kroku simulace musíme provést N^2 výpočtů, což činí simulaci výpočetně náročnou. Zásadnějším problémem ovšem je, že pokud je krok dt příliš velký, částice se mohou namísto kolize „přeskočit“. Chceme-li dosáhnout vyšší přesnosti, musíme mít dostatečně malé dt, což opět zvyšuje výpočetní nároky na simulaci.

2. *Událostmi řízená simulace:* vzhledem k tomu, že se v tomto jednoduchém modelu veškeré částice mezi kolizemi pohybují rovnoměrnou rychlostí po přímce, můžeme se soustředit pouze na takové časové okamžiky, při kterých dochází k zajímavým událostem – tedy kolizím. Řešení problému je realizováno vytvořením seznamu kolizí, ke kterým by došlo, pokud by se všechny částice pohybovaly navždy přímo. Tento seznam následně procházíme, odstraníme kolize, které jsou nevalidní (kolizi $c1$ označíme za nevalidní, pokud se jí účastní částice p, která se od okamžiku vypočtení $c1$ dostala do kolize $c2$, jež změnila vektor rychlosti částice p), alternativně pokud kolize v seznamu odpovídá validní kolizi mezi částicemi i a j, aktualizujeme vektory rychlostí těchto částic a vypočteme a vložíme do seznamu nové kolize týkající se těchto částic.

Druhý postup dává robustnější, přesnější a méně výpočetně náročnou simulaci, ovšem za cenu komplexnějšího kódu. Pokud si zvolíme událostmi řízenou simulaci, budeme potřebovat rovnici pro vypočtení času do kolize dvou částic. Pokud uvažujeme uzavřený prostor, budeme také potřebovat zjistit čas do kolize částice a stěny. V obou přístupech budeme potřebovat také rovnici pro výpočet nových rychlostí částic po kolizi.

Pro zjednodušení uvažujeme, že se částice pohybují v uzavřeném prostoru na intervalu 0 až 1 a všechny mají stejnou hmotnost. Mějme částici tvaru kruhu, jejíž střed se nachází na souřadnicích $[x, y]$, pohybuje se rychlostí $[v_x, v_y]$ a má poloměr σ. Je zřejmé, že částice přichází do kontaktu s vodorovnou stěnou v čase $t + \Delta t$, pokud platí, že $y + \Delta t v_y$ je rovno buď σ nebo $1 - \sigma$. Můžeme tedy vyjádřit Δt:

$$\Delta t = \begin{cases} (1 - \sigma - y)/v_y & \text{pokud } v_y > 0 \\ (\sigma - y)/v_y & \text{pokud } v_y < 0 \\ \infty & \text{pokud } v_y = 0 \end{cases}$$

Analogickým postupem získáme rovnici pro výpočet času do kolize se svislou stěnou. Pro výpočet času do kolize dvou částic i a j máme vztah:

$$\begin{aligned} \Delta r &= (\Delta x, \Delta y) = (x_j - x_i, y_j - y_i) \\ \Delta v &= (\Delta v_x, \Delta v_y) = (v_{x,j} - v_{x,i}, v_{y,j} - v_{y,i}) \\ \Delta r \Delta r &= (\Delta x)^2 + (\Delta y)^2 \\ \Delta v \Delta v &= (\Delta v_x)^2 + (\Delta v_y)^2 \\ \Delta v \Delta r &= (\Delta v_x)(\Delta x) + (\Delta v_y)(\Delta y) \\ d &= (\Delta v \Delta r)^2 - (\Delta v \Delta v)(\Delta r \Delta r - \sigma^2) \\ \Delta t &= \begin{cases} \infty & \text{pokud } \Delta v \Delta r \geq 0 \\ \infty & \text{pokud } d < 0 \\ -\frac{\Delta v \Delta r + \sqrt{d}}{\Delta v \Delta v} & \text{jinak} \end{cases} \end{aligned}$$

Pro výpočet nové rychlosti částice při kolizi se stěnou stačí převrátit hodnotu odpovídající složky rychlosti – tedy máme-li částici s rychlostí (v_x, v_y), bude mít po nárazu do svislé stěny rychlost $(v_x, -v_y)$, po kolizi s vodorovnou stěnou bude její rychlost $(-v_x, v_y)$.

Po kolizi dvou částic budou jejich nové rychlosti dány vztahem: [32]

$$v_1' = v_1 - \frac{\langle v_1 - v_2, x_1 - x_2 \rangle}{\|x_2 - x_1\|^2}(x_1 - x_2)$$
$$v_2' = v_2 - \frac{\langle v_2 - v_1, x_2 - x_1 \rangle}{\|x_1 - x_2\|^2}(x_2 - x_1)$$

Část II.

Praktická část

Tato část práce obsahuje vypracované úlohy. Zdrojové kódy těchto programů, stejně jako zdrojový kód této práce, jsou k dispozici v příloze. Jednotlivé programy jsou realizovány vždy podle stejného vzoru – skládají se ze dvou souborů, nazvaných *<úloha>.R* a *<úloha>-shiny.R*.

Soubor *<úloha>.R* obsahuje tak zvaný backend programu, skupinu funkcí, které realizují úlohu jako takovou (výpočty, algoritmy, simulace...), plus pomocné práce, jako je importování knihoven, inicializace, vykreslování výstupů či parsování vstupů.

Tento backend je využíván dvěma typy frontendu: jednak se jedná o soubor *<úloha>-shiny.R*, která implementuje webovou aplikaci ve frameworku *Shiny*. Za druhé jde přímo o tuto práci – tedy kód je volán přímo z tohoto textového dokumentu – pro účely demonstrace knihovny *knitr* (a současně pro popis samotné implementace dané úlohy).

Popis výše není zcela přesný: jelikož je v tomto dokumentu popisován i *Shiny* frontend, je i kód obsažený v souborech *<úloha>-shiny.R* importován do tohoto textového souboru – ovšem pouze pro účely jeho popisu, tento kód jako takový není vyhodnocován.

4. K-means

Jak bylo zmíněno v teoretické části, R poskytuje vlastní implementaci pro výpočet k-means. Jedná se o funkci `kmeans` z balíčku `stats`. Tato funkce implementuje několik algoritmů (konkrétně se jedná o trojici Lloyd-Forgy, MacQueen a Hartigan-Wong), které byly popsány v teoretické části práce. Všechny tyto algoritmy jsou iterativní, to znamená, že vycházejí z určitého počátečního stavu (jež může být určen náhodně) a opakováním řady kroků konvergují k optimu.

Cílem naší vizualizace je znázornění této konvergence. K tomu využijeme jednoho z parametrů funkce `kmeans`, nazvaný `iter.max`. Tento parametr určuje maximální povolený počet iterací pro dané volání funkce. Pokud máme konstantní dataset a počáteční pozice středů klastrů, můžeme tuto funkci opakovaně volat se zvyšující se hodnotou `iter.max`, abychom získali stav výpočtu v jednotlivých krocích algoritmu.

Dále budeme chtít vědět, zda algoritmus v daném počtu kroků konvergoval. Bohužel, funkce `kmeans` tuto informaci poskytuje poněkud nešťastným způsobem, totiž vypsáním varování v případě, že algoritmus nekonvergoval. Pro naše potřeby toto varování místo vypsání na stdout uložíme do seznamu. Následně vrátíme logickou hodnotu určenou na základě toho, zda tento seznam obsahuje alespoň jeden prvek.

Za tímto účelem si vytvoříme wrapper:

```
kmeans.wrapper = function(points, centers, step, algorithm = "Lloyd") {
  warning = NULL
  handler = function(w) {
    warning <<- c(warning, list(w))
    invokeRestart("muffleWarning")
  }
  means = withCallingHandlers(
    kmeans(points, centers, iter.max = step, algorithm = algorithm),
    warning = handler)
  list(means = means, converged = is.null(warning))
}
```

Další vlastní funkce, kterou budeme potřebovat, bude mít za úkol grafické znázornění objektu třídy `kmeans`, což je objekt vracený funkcí implementovanou v R. Vykreslení grafu bude probíhat tak, že každému klastru přiřadíme odlišnou barvu, každý bod obarvíme podle toho, k jakému klastru náleží, a pro každý bod vykreslíme tenkou úsečkou spojující jej s geometrickým středem daného klastru. Barvy pro jednotlivé klastry budeme náhodně vybírat z JSON souboru *colors.json*, který byl vygenerován z webu `tools.medialab.sciences-po.fr/iwanthue/`. Pro čtení JSON formátu použijeme knihovnu `jsonlite`.

```
library(jsonlite)
fileName = "./colors.json"
json = readChar(fileName, file.info(fileName)$size)
colors = sample(fromJSON(json))
```

Funkce `display` přijímá čtyři parametry: `mns` je objekt vracený funkcí `kmeans`, uložený pod názvem `means` ve výstupu našeho wrapperu; `pts` jsou souřadnice všech bodů; `cts` jsou souřadnice počátečních geometrických středů jednotlivých klastrů; nakonec `size` je velikost plochy. Nejprve jsou nastaveny grafické parametry a paleta barev. Je vytvořen prázdný graf o daných rozměrech (0 až `size`). Pokud nad body nebylo provedeno klastrování, jsou vykresleny jen samotné body a centra (pokud existují). Jinak jsou vykresleny nejprve jednotlivé body, každý barvou podle klastru, ke kterému náleží. Následně pro každý geometrický střed je vykreslen opět odpovídající barvou bod a pro každý pár středu a bodu, který k němu náleží, je vykreslena tenká čára.

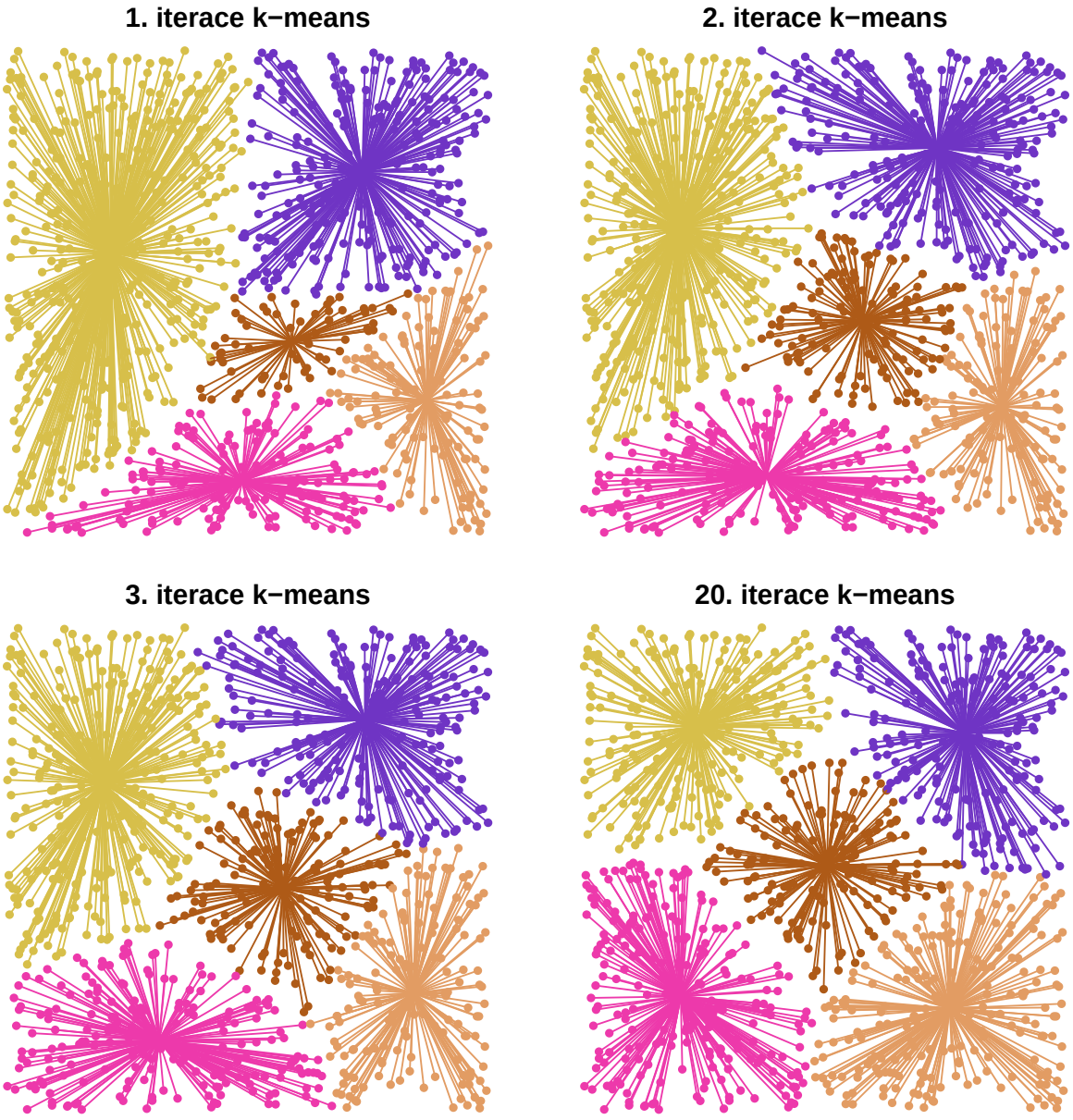
```
display = function(mns, pts, cts, size) {
  par(
    fg = "transparent",
    col.axis = "transparent",
    col.lab = "transparent",
    mar = c(1, 1, 1, 1)
  )
  palette(colors)
  plot(0, 0, type="n", xlab="", ylab="", xlim = c(0, size), ylim = c(0, size))
  if (is.null(mns)) {
    if (!is.null(pts)) {
      points(pts, pch = 20, col = "black")
    }
    if (!is.null(cts)) {
      points(cts, pch = 4, cex = 4, lwd = 4, col = "green")
    }
  } else if (!is.null(pts)) {
    points(pts, pch = 20, col = mns$cluster)
    for (i in 1:nrow(pts)) {
      x0 = c(pts[i,1], mns$centers[mns$cluster[i],1])
      y0 = c(pts[i,2], mns$centers[mns$cluster[i],2])
      lines(x0, y0, col = mns$cluster[i])
    }
  }
}
```

4.1. knitr

Vytvoříme čtyři grafy, reprezentující k-means clustering nad tisíci náhodnými body a pěti klastry. Zobrazíme stav po první, druhé a třetí iteraci, stejně jako stav v konvergovaném

stavu. K výpočtu použijeme Lloydův algoritmus.

```
size = 1 # max velikost grafu
pn = 1000 # počet bodů
ps = matrix(runif(2*pn, 0, size), ncol = 2, nrow = pn)
cn = 5 # počet klastrů
cs = matrix(runif(2*cn, 0, size), ncol = 2, nrow = cn)

par(mfrow=c(2,2))

# první krok iterace
m = kmeans.wrapper(ps, cs, 1)
display(m$means, ps, cs, size)
title("1. iterace k-means")
m$converged

## [1] FALSE



# druhý krok
m = kmeans.wrapper(ps, cs, 2)
display(m$means, ps, cs, size)
title("2. iterace k-means")
m$converged

## [1] FALSE



# třetí krok
m = kmeans.wrapper(ps, cs, 3)
display(m$means, ps, cs, size)
title("3. iterace k-means")
m$converged

## [1] FALSE



# graf v konvergovaném stavu
i = 1
while (!m$converged) {
  m <<- kmeans.wrapper(ps, cs, i)
  i = i+1
}
display(m$means, ps, cs, size)
title(paste(i, ". iterace k-means", sep = ""))
```




```
m$converged
```

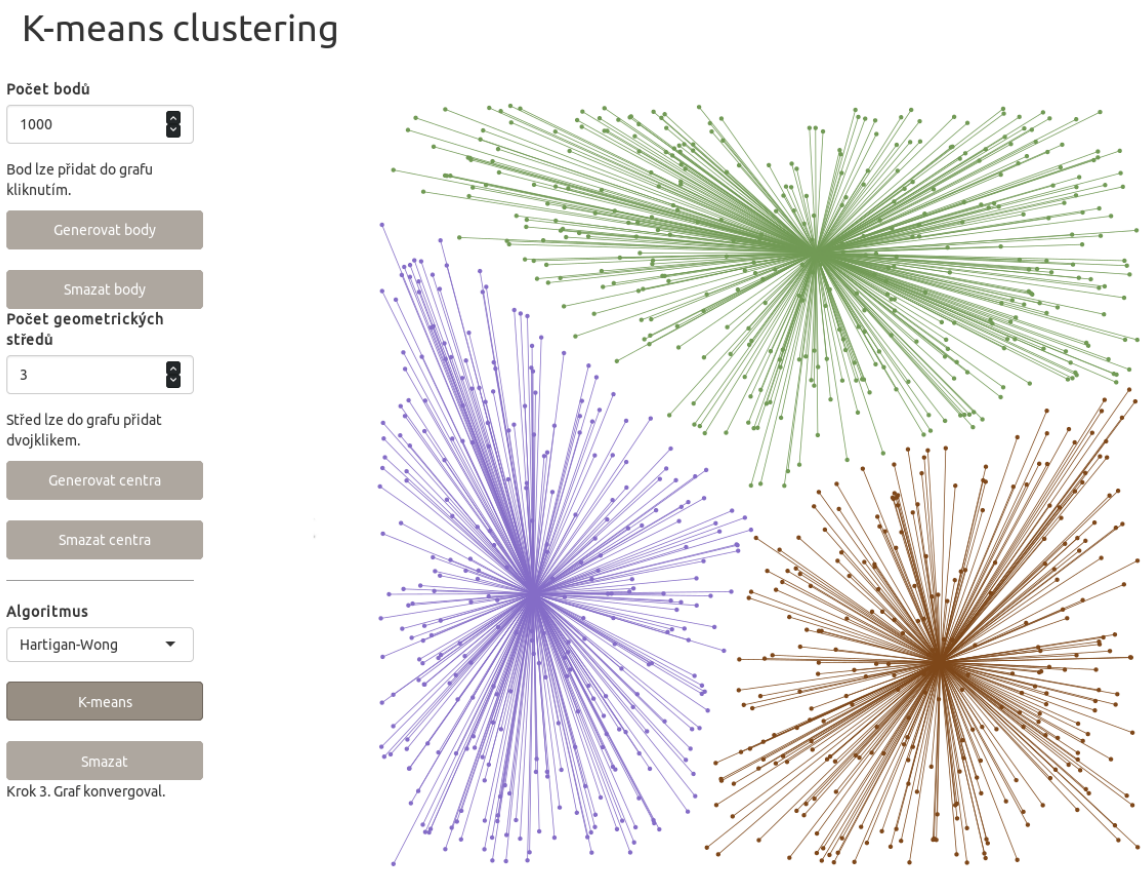
```
## [1] TRUE
```

Porovnání rychlosti konvergence jednotlivých algoritmů pro výpočet k-means:

```
size = 1 # max velikost grafu
pn = 10000 # počet bodů
ps = matrix(runif(2*pn, 0, size), ncol = 2, nrow = pn)
cn = 12 # počet klastrů
cs = matrix(runif(2*cn, 0, size), ncol = 2, nrow = cn)

converged.in = function(algorithm) {
```

```
  i = 1
  m = kmeans.wrapper(ps, cs, i, algorithm)
  while (!m$converged) {
    m = kmeans.wrapper(ps, cs, i, algorithm)
    i = i+1
  }
  i
}

converged.in("Lloyd")

## [1] 30

converged.in("Forgy")

## [1] 30

converged.in("MacQueen")

## [1] 18

converged.in("Hartigan-Wong")

## [1] 6
```

4.2. Shiny



Obrázek 4.1.: K-means – Shiny aplikace

Kromě běžných knihoven `shiny` a `shinythemes` importujeme ještě knihovnu `glue`, která slouží k interpolaci řetězců (vkládání proměnných přímo do textu). Importujeme také soubor realizující backend.

```
library(shiny)
library(glue)
library(shinythemes)
source("kmeans.R")
```

Nastavení globálních proměnných: velikost plochy, rozsah grafu, výchozí počet bodů a geometrických středů, a délku trvání jednoho kroku (v milisekundách). Na rozdíl od jiných aplikací zde není cílem dosáhnout co nejrychlejšího běhu, ale ukázat iterativní, konvergující povahu algoritmů pro výpočet k-means, chceme proto, aby každý krok na nějakou dobu zůstal na obrazovce.

```
size = 1
lim = c(0, size)
point.count.default = 1000
centers.count.default = 3
step.time = 1
```

V deklaraci serverové funkce je vytvořena pomocná funkce pro vygenerování náhodných bodů. Reaktivní proměnné obsahují souřadnice bodů a geometrických středů, objekt `means` vracený naším wrapperem nad nativní metodou `kmeans`, příznak konvergence (překreslování grafu zastavíme, pokud již došlo ke konvergenci), čítač kroků a příznak běhu výpočtu.

```
server = function(input, output, session) {
  gen.points = function(point.count) {
    matrix(runif(2*point.count, 0, 1), ncol = 2)
  }

  values = reactiveValues()
  values$points = gen.points(point.count.default)
  values$centers = gen.points(centers.count.default)
  values$means = NULL
  values$converged = NULL
  values$step = 0
  values$run = FALSE
```

Následující skupina observerů obsluhuje tlačítka pro generování a smazání bodů a geometrických středů, a smazání výsledku výpočtu. Dá se říci, že jsou triviální. Tlačítka pro generování využívají výše definované funkce, které předávají odpovídající hodnotu ze vstupu – buď počet bodů nebo geometrických středů. Tlačítka pro mazání bodů a středů nastavují odpovídajícím reaktivním proměnným hodnotu `NULL`. Handler pro smazání výsledku výpočtu nastaví hodnotu `NULL` pro reaktivní proměnnou `means` a příznak konvergence, nastaví `FALSE` pro vlajku indikující běh výpočtu a resetuje čítač kroků.

```
  observeEvent(input$gen.points, {
    values$means = NULL
    values$converged = NULL
    values$step = 0
    values$run = FALSE
    values$points = gen.points(input$point.count)
  })

  observeEvent(input$clear.points, {
    values$means = NULL
    values$converged = NULL
    values$step = 0
    values$run = FALSE
    values$points = NULL
```

```
})

observeEvent(input$gen.centers, {
  values$centers = gen.points(input$centers.count)
})

observeEvent(input$clear.centers, {
  values$centers = NULL
})

observeEvent(input$clear.kmeans, {
  values$run = FALSE
  values$means = NULL
  values$converged = NULL
  values$step = 0
})
```

Zde máme další ukázku možností interaktivních grafů v *Shiny* aplikaci. Definujeme dvojici event handlerů, které reagují na jednoduché kliknutí a dvojklik na graf. Při jednoduchém kliknutí se daný bod (souřadnice kliku) přidají do seznamu bodů, při dvojkliku se přidají do seznamu geometrických středů.

```
observe({
  x = input$plot.click$x
  y = input$plot.click$y
  isolate({
    values$points = rbind(values$points, c(x, y))
  })
})

observe({
  x = input$plot.dblclick$x
  y = input$plot.dblclick$y
  isolate({
    values$centers = rbind(values$centers, c(x, y))
  })
})
```

Následující observer realizuje jeden krok výpočtu. Výpočet je podmíněn jednak pravdivou hodnotou vlajky `run`, dále pak tím, že jsou nadefinovány body a středy. Je zvýšena hodnota čítače kroků. Je zavolán v backendu definovaný wrapper pro výpočet k-means a jeho výstup je uložen do odpovídajících reaktivních proměnných. Nakonec, podle toho, zda již došlo ke konvergenci, je tato metoda zařazena do fronty pro znovuvyhodnocení, pokud již výpočet konvergoval, je výpočet zastaven.

```
observe({
  if (
    values$run &&
    (!is.null(values$centers)) &&
    (nrow(values$centers) > 0) &&
    (!is.null(values$points)) &&
    (nrow(values$points) > 0)
  ) {
    isolate({
      values$step = values$step + 1
    })

    means = kmeans.wrapper(
      values$points, values$centers, values$step, input$algorithm)
    values$means = means$means
    values$converged = means$converged

    if (!values$converged) {
      invalidateLater(step.time)
    } else {
      values$run = FALSE
    }
  }
})
```

Obsluha tlačítka pro spuštění výpočtu sestává z nastavení pravdivé hodnoty pro odpovídající vlajku a resetování výsledku předchozího výpočtu. Podobně jednoduchá je funkce pro vykreslení grafu, která sestává pouze z volání metody backendu. Následuje handler pro text output, který bude zobrazovat čítač kroků a informaci, zda již došlo ke konvergování grafu.

```
observeEvent(input$kmeans, {
  values$means = NULL
  values$converged = NULL
  values$step = 0
  values$run = FALSE
  values$run = TRUE
})

output$plot = renderPlot({
  display(values$means, values$points, values$centers, size)
})

output$label = renderText({
  if (!is.null(values$means) &&
      !is.null(values$points) &&
      !is.null(values$converged)) {
```

```
      convergence = if (values$converged) "Graf konvergoval."
        else "Graf konverguje."
      glue("Krok {values$step}.\n{convergence}")
    } else {""}
  })
}
```

Uživatelské rozhraní má běžné uspořádání. Stránka je rozdělena na dva sloupce oddělené mezerou. V užším levém sloupci se nacházejí ovládací prvky: dvojice číselných vstupů pro nastavení počtu bodů a geometrických středů, tlačítka pro jejich vygenerování a smazání, stejně jako tlačítka pro spuštění a smazání výpočtu. Dále se zde nachází `selectInput` umožňující uživateli zvolit jeden ze čtyř algoritmů pro výpočet k-means.

V širším sloupci je umístěn pouze výstup pro graf. Zde je jedinou zajímavostí nastavení názvů proměnných, jež budou obsahovat informace o kliknutí a dvojkliku.

```
ui = fluidPage(
  theme = shinytheme("united"),
  tags$style(type = "text/css",
    ".recalculating {opacity: 1.0;}"
  ),
  headerPanel("K-means clustering"),
  fluidRow(
    column(2,
      verticalLayout(
        br(),
        numericInput("point.count", "Počet bodů",
          min = 100, max = 1000000, value = point.count.default, width = "15vw"),
        p("Bod lze přidat do grafu kliknutím."),
        actionButton("gen.points", label = "Generovat body", width = "15vw"),
        br(),
        actionButton("clear.points", label = "Smazat body", width = "15vw"),
        numericInput("centers.count", "Počet geometrických středů",
          min = 1, max = 100, value = centers.count.default, width = "15vw"),
        p("Střed lze do grafu přidat dvojklikem."),
        actionButton("gen.centers", label = "Generovat centra", width = "15vw"),
        br(),
        actionButton("clear.centers", label = "Smazat centra", width = "15vw"),
        tags$hr(style="border-color: #999999;"),
        selectInput("algorithm", "Algoritmus", algorithms,
          selected = NULL, multiple = FALSE, width = "15vw"),
        actionButton("kmeans", label = "K-means", width = "15vw"),
        br(),
        actionButton("clear.kmeans", label = "Smazat", width = "15vw"),
        textOutput('label')
      )
    ),
```

```
    column(1),
    column(9,
      plotOutput("plot", height = "92vh", width = "92vh",
        click = "plot.click",
        dblclick = dblclickOpts(
          id = "plot.dblclick"
      ))
    )
  )
)
```

Aplikaci spustíme zavoláním funkce `runApp`.

```
runApp(appDir = shinyApp(ui, server), port = 8005, host = "127.0.0.1")
```

5. Word cloud

Pro generování word cloudu použijeme knihovny `tm` (text mining – příprava vstupního textu) a `wordcloud`, dále pak knihovnu `RColorBrewer` pro vytvoření barevné palety.

```
library(tm)
library(wordcloud)
library(RColorBrewer)
```

Vytvoříme seznam všech palet obsažených v balíčku `RColorBrewer` zavoláním metody `as.list` na seznam jmen řádků tabulky `brewer.pal.info`. Podobným postupem získáme seznam všech barev vestavěných v R. Tyto seznamy poslouží jako zdroj možností pro `selectInput` v *Shiny* aplikaci. Vytvoříme také seznam, který bude sloužit k výběru způsobu řazení slov ve výstupním grafu.

```
fg_colors = as.list(rownames(brewer.pal.info))
bg_colors = as.list(colors(distinct = TRUE))
ordering = list("Náhodně" = TRUE, "Seřazeno" = FALSE)
```

Samotné generování word cloudu sestává pouze ze dvou metod. Funkce `term.matrix` slouží k vytvoření matice výrazů (*term matrix*) ze vstupního textu. Nejprve vytvoříme tak zvaný *korpus*, k tomu použijeme metodu `Corpus` knihovny `tm`. Tato knihovna nám umožňuje na korpus aplikovat řadu užitečných filtrů, např. převední slov do základního tvaru, do malého písma nebo odstranění nežádoucích výrazů (tzv. stopwords, často aplikováno na spojky, členy a podobně). Filtry využité v naší funkci `term.matrix` jsou převedení do malého písma, odstranění čísel a interpunkce, volitelně odstranění stopwords. Poté již můžeme korpus převést na matici výrazů, a to funkcí `TermDocumentMatrix`.

```
term.matrix = function(text) {
  corpus = Corpus(VectorSource(text))
  corpus = tm_map(corpus, content_transformer(tolower))
  corpus = tm_map(corpus, removePunctuation)
  corpus = tm_map(corpus, removeNumbers)
  doc.matrix = TermDocumentMatrix(corpus)
  mat = as.matrix(doc.matrix)
  sort(rowSums(mat), decreasing = TRUE)
}
```

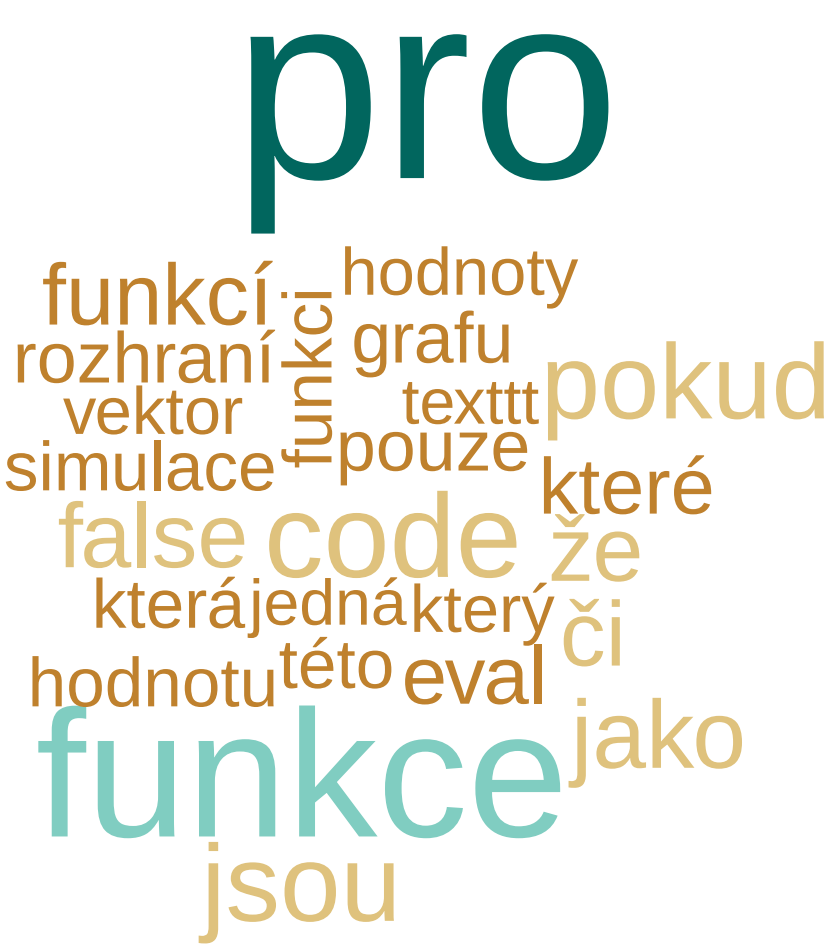
Druhou důležitou funkcí backendu je samotné vykreslení word cloudu, k čemuž použijeme funkci `wordcloud` ze stejnojmenného balíku. Předávanými parametry je barva pozadí, barevná paleta pro vykreslení jednotlivých slov word cloudu, minimální frekvence výskytu slova v textu, maximální počet slov ve word cloudu a velikost nejmenšího a největšího slova.

```
display = function(terms,
  color = fg_colors[1],
  background = "transparent",
  min.freq = 1,
  max.words = 25,
  min.scale = 1,
  max.scale = 5,
  random.order = TRUE)
{
  par(bg = background)
  wordcloud(names(terms), terms,
    random.order = random.order,
    colors = brewer.pal(8, color),
    min.freq = min.freq,
    max.words = max.words,
    scale = c(max.scale, min.scale))
}
```

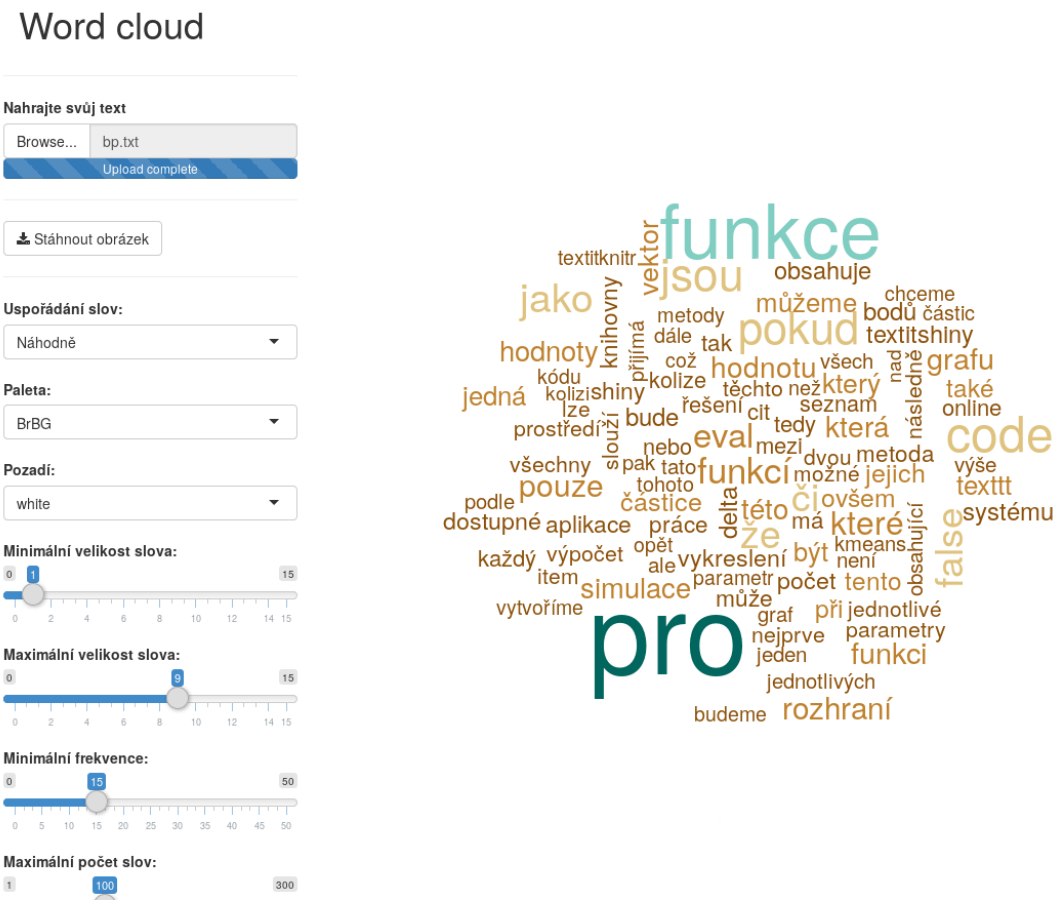
5.1. knitr

S výše nadefinovanými funkcemi je vytvoření word cloudu snadné. Jako vstupní text použijeme tento dokument.

```
terms = term.matrix(readLines("./bp.Rnw", encoding = "UTF-8"))
display(terms, max.scale = 9, color = "BrBG")
```


pro
funkcí
hodnoty
rozhraní
funkci
grafu
pokud
vektor
texttt
simulace
pouze
které
false
code
že
kterájednákterý
či
hodnotu
této
eval
funkce
jako
jsou

5.2. Shiny



Obrázek 5.1.: Word cloud – Shiny aplikace

Import knihovny `shiny` a backendu.

```
library(shiny)
source("wordcloud.R")
```

Uživatelské rozhraní je rozděleno do dvou sloupců. V levém sloupci se nachází widget `fileInput` pro nahrání souboru ve formátu .txt. Další ovládací prvek slouží ke stažení vygenerovaného obrázku. Zbytek widgetů umožňuje nastavení parametrů pro generování word cloudu. Jedná se o stejné hodnoty, které jsou předávány funkci `display`: barva pozadí a barevná paleta word cloudu, velikost nejmenšího a největšího slova, minimální frekvenci a maximální počet slov. Pravý sloupec pak obsahuje grafický výstup pro vykreslení word cloudu.

```
ui = fluidPage(
  headerPanel("Word cloud"),
  fluidRow(
    column(3,
```

```
      hr(),
      fileInput("file", "Nahrajte svůj text", multiple = FALSE,
        accept = c("text/plain", ".txt")),
      hr(),
      downloadButton("download", "Stáhnout obrázek", width = "100%"),
      hr(),
      selectInput("order", "Uspořádání slov:", choices = ordering),
      selectInput("color", "Paleta:", choices = fg_colors),
      selectInput("background", "Pozadí:", choices = bg_colors),
      sliderInput("min.scale", "Minimální velikost slova:", min = 0,
        max = 15,  value = 1),
      sliderInput("max.scale", "Maximální velikost slova:", min = 0,
        max = 15, value = 5),
      sliderInput("min.freq", "Minimální frekvence:", min = 0,
        max = 50, value = 15),
      sliderInput("max.words", "Maximální počet slov:", min = 1,
        max = 300,  value = 100)
    ),
    column(9,
      plotOutput("plot", width = "90vh", height = "90vh")
    )
  )
)
```

Serverová funkce obsahuje reaktivní proměnnou `terms`, která obsahuje matici výrazů. Následuje event handler pro nahrání souboru. Pokud uživatel žádný soubor nenahraje, je jako zdroj textu opět použit tento dokument. Daný soubor je přečten a jeho obsah předán backendové funkci `term.matrix`. Její výstup je uložen do reaktivní proměnné `terms`.

Nadefinujeme funkci `plot.wordcloud` pro vykreslení word cloudu. Bude použita na dvou místech: jednak pro zobrazení grafu na stránce, současně také při stažení obrázku. Tato funkce volá metodu `display` s parametry zadanými vstupními prvky, pokud existuej matice výrazů (tedy není `NULL`).

Vykreslení grafu ve stránce je ekvivalentní výše popsané funkci `plot.wordcloud`. V event handleru pro stažení obrázku je potřebu udělat něco navíc: definujeme grafické zařízení `png`, čímž dojde k vytvoření bitmapy ve formátu .png se zadaným názvem. Do této bitmapy vykreslíme word cloud. Bitmapu uzavřeme funkcí `dev.off`. Takto vygenerovaný soubor je předán k downloadu.

```
server = function(input, output, session) {
  values = reactiveValues()
  values$terms = NULL

  observe({
    inputPath = input$file$datapath
    if (!is.null(inputPath)) {
```

```
      text = readLines(inputPath, encoding = "UTF-8")
      values$text = text
      values$terms = term.matrix(text)
    }
  })

  plot.wordcloud = function() {
    if (!is.null(values$terms)) {
      display(values$terms,
        color = input$color,
        background = input$background,
        min.freq = input$min.freq,
        max.words = input$max.words,
        min.scale = input$min.scale,
        max.scale = input$max.scale,
        random.order = input$order)
    }
  }

  output$plot = renderPlot(plot.wordcloud())

  output$download = downloadHandler(
    filename = "wordcloud.png",
    content = function(file) {
      png(file)
      plot.wordcloud()
      dev.off()
    }
  )
}
```

Spuštění aplikace běžným způsobem:

```
runApp(appDir = shinyApp(ui, server), port = 8007, host = "127.0.0.1")
```

6. Schulzeho metoda

Pro práci se sítěmi a vykreslování síťových grafů budeme potřebovat knihovny `network`, `ggplot2` a `GGally`.

```
library(network)
library(GGally)
library(ggplot2)
```

Nadefinujeme si funkci `preferred`. Slouží k rozpoznání, který ze dvou kandidátů x a y je preferován v nějakém uspořádání. Příklad: mějme vektor reprezentující preference určitého počtu voličů ve tvaru $C > A > B$, a chceme zjistit, zda je v tomto uspořádání kandidát A preferován před B.

```
preferred = function(X, Y, prefs) {
  ixX = match(X, prefs)
  ixY = match(Y, prefs)
  ixX < ixY
}
```

```
preferred("A", "B", c("C","A","B"))


## [1] TRUE
```

Metoda `pairwise` slouží k vytvoření matice párových preferencí. Přijímá dva parametry. `a` je jednoduchý textový vektor obsahující jména všech kandidátů. Objekt `A` je seznam seznamů v následujícím tvaru: každý vnitřní seznam má právě dva prvky, tím prvním je opět jednoduchý textový vektor jmen kandidátů, uspořádaných nějakým způsobem; druhým prvkem je číslo vyjadřující počet voličů, kteří preferují toto uspořádání.

```
pairwise = function(A, a) {
  C = length(a)
  D = matrix(0, nrow = C, ncol = C, dimnames = list(a, a))
  for (i in 1:C) {
    for (j in 1:C) {
      if (i != j) {
        for (pref in A) {
          if (preferred(a[i], a[j], pref[[1]])) {
            D[i, j] = D[i, j] + pref[[2]]
```

```
          }
        }
      }
    }
  }
  return(D)
}
```

Funkce `strongest.paths` přijímá matici párových preferencí a vektor jmen kandidátů a vypočítává matici nejsilnějších cest.

```
strongest.paths = function(D, a) {
  C = dim(D)[1]
  P = matrix(0, nrow = C, ncol = C, dimnames = list(a, a))
  for (i in 1:C) {
    for (j in 1:C) {
      if (i != j) {
        if (D[i, j] > D[j, i]) {
          P[i, j] = D[i, j]
        } else {
          P[i, j] = 0
        }
      }
    }
  }
  for (i in 1:C) {
    for (j in 1:C) {
      if (i != j) {
        for (k in 1:C) {
          if (i != k && j != k) {
            P[j, k] = max(P[j, k], min(P[j, i], P[i, k]))
          }
        }
      }
    }
  }
  return(P)
}
```

Nakonec už zbývá jen z tabulky nejsilnějších cest získat vítězné uspořádání kandidátů. K tomu slouží funkce `result`.

```
result = function(P, a) {
  C = length(a)
  b = vector()
  for (i in 1:C) {
```

```
    ix = 1
    for (j in 1:C) {
      if (i != j) {
        if (P[i, j] < P[j, i]) {
          ix = ix + 1
        }
      }
    }
    b[ix] = a[i]
  }
  return(b)
}
```

Funkce `display` vykresluje síťový graf reprezentující párové preference. Prochází tabulku párových preferencí a pro každou buňku, která se nenachází na hlavní diagonále, rozhodne, kterým směrem má být daná hrana orientovaná a do vektoru popisků hran vloží hodnotu odpovídající vítězné buňce. Výstupní graf je uspořádán do tvaru kruhu.

```
display = function(D) {
  a = sort(rownames(D))
  C = length(a)
  edge.labels = vector()
  heads = vector()
  tails = vector()
  for (i in 1:C) {
    for (j in i:C) {
      if (j != i) {
        v = if (D[i,j] > D[j,i]) c(i,j) else c(j,i)
        heads = c(heads, v[1])
        tails = c(tails, v[2])
        edge.labels = c(edge.labels, D[v[1], v[2]])
      }
    }
  }
  ggnet2(network(data.frame(heads, tails)),
    size = 0,
    mode = "circle",
    label = a,
    edge.size = 1,
    edge.color = "#555555",
    edge.label = edge.labels,
    edge.label.size = 6,
    arrow.size = 6,
    arrow.gap = 0.05) +
  geom_point(aes(color = color), size = 12, color = "black") +
  geom_point(aes(color = color), size = 11, color = "white") +
```

Tabulka 6.1.: Tabulka párových preferencí 1

	a	b	c	d	e
a	0	20	26	30	22
b	25	0	16	33	18
c	19	29	0	17	24
d	15	12	28	0	14
e	23	27	21	31	0

```
  geom_text(aes(label = label), color = "black", size = 6)
}
```

6.1. knitr

Vytvoříme si seznam kandidátů a jednotlivých hlasů:

```
a = c("a","b","c","d","e")
A = list(
  list(c("a","c","b","e","d"), 5),
  list(c("a","d","e","c","b"), 5),
  list(c("b","e","d","a","c"), 8),
  list(c("c","a","b","e","d"), 3),
  list(c("c","a","e","b","d"), 7),
  list(c("c","b","a","d","e"), 2),
  list(c("d","c","e","b","a"), 7),
  list(c("e","b","a","d","c"), 8)
)
D = pairwise(A, a)
```

Vypočteme tabulku párových preferencí. Pro vykreslení tabulky v *knitru* použijeme balík `kable`.

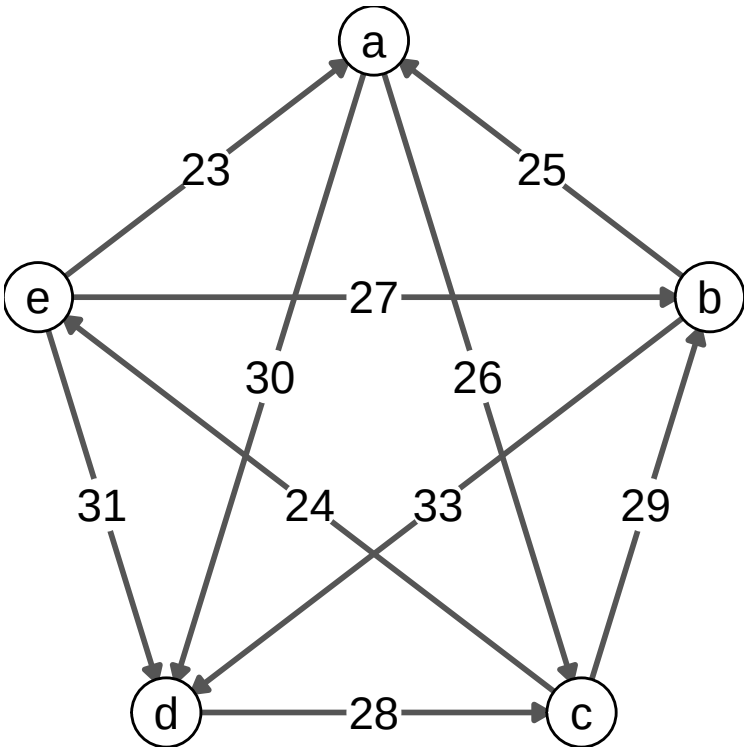
```
kable(D, booktabs = TRUE, caption = "Tabulka párových preferencí 1") %>%
  kable_styling(position = "center")
```

A graf:

```
display(D)
```

Tabulka 6.2.: Tabulka párových preferencí 2

	a	b	c	d	e
a	0	-5	7	15	-1
b	5	0	-13	21	-9
c	-7	13	0	-11	3
d	-15	-21	11	0	-17
e	1	9	-3	17	0



Tutéž tabulku ve tvaru rozdílů snadno získáme odečtením transponované podoby matice:

```
margins = D - t(D)
kable(margins, booktabs = TRUE, caption = "Tabulka párových preferencí 2") %>%
  kable_styling(position = "center", full_width = TRUE, font_size = 7)
```

Následně vypočteme matici nejsilnějších cest:

```
P = strongest.paths(D, a)
kable(P, booktabs = TRUE, caption = "Tabulka nejsilnějších cest") %>%
  kable_styling(position = "center")
```

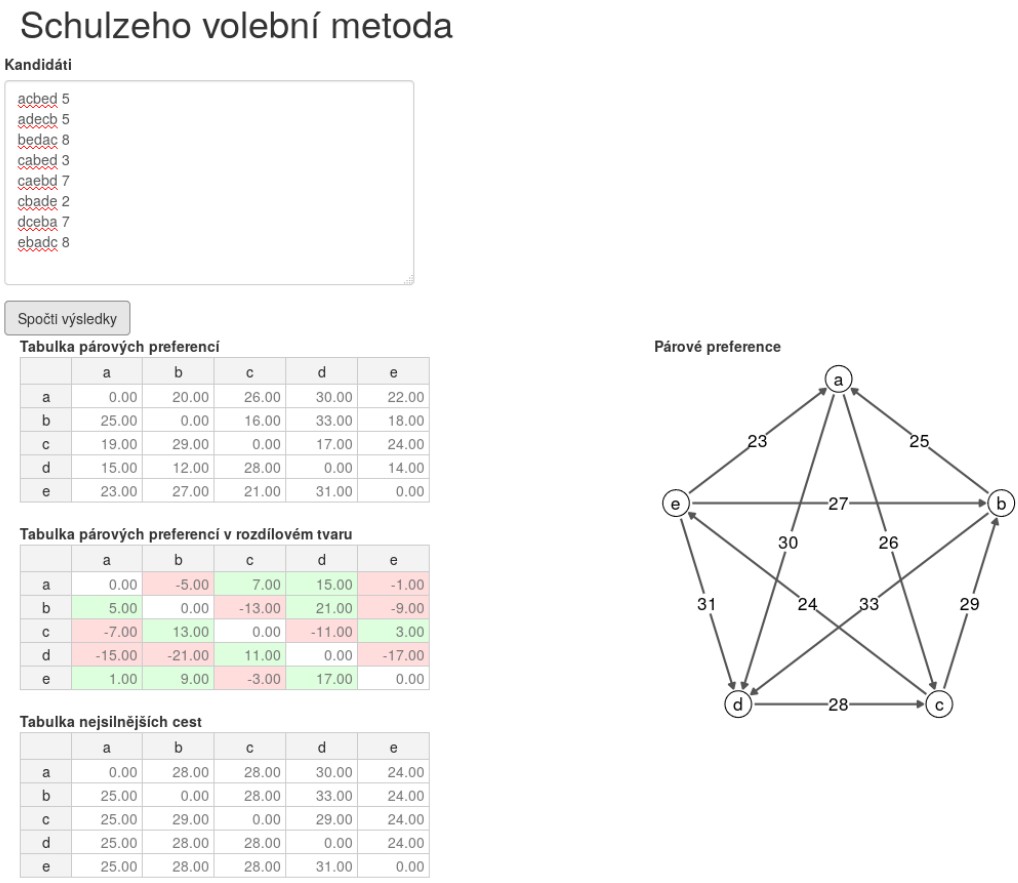
Nyní můžeme zjistit vítězné uspořádání kandidátů:

```
result(P, a)


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

Tabulka 6.3.: Tabulka nejsilnějších cest

	a	b	c	d	e
a	0	28	28	30	24
b	25	0	28	33	24
c	25	29	0	29	24
d	25	28	28	0	24
e	25	28	28	31	0

6.2. Shiny



Obrázek 6.1.: Schulzeho volební metoda – Shiny aplikace

Importujeme knihovny `shiny`, `shinyjs` a `rhandsontable`, a backend. Knihovnu `shinyjs` využijeme k dynamickému zobrazení elementů s výsledky až po jejich vypočtení. `Rhandsontable` je R wrapper pro knihovnu `handsontable.js`, který využijeme pro vykreslování interaktivních tabulek s výsledky.

```
library(shiny)
library(shinyjs)
library(rhandsontable)
source("./schulze.R")
```

Nadefinujeme dvě pomocné funkce, `get.ballot` a `get.candidates`. Chceme, aby uživatelé mohli zadávat vstupy ve tvaru:

abc 12
cba 5
bac 8

Tedy každý řádek bude obsahovat dva řetězce oddělené mezerou, první řetězec obsahující uspořádání kandidátů, druhým bude číslo reprezentující počet voličů, kteří preferují toto uspořádání. Jména kandidátů mohou být pouze jednopísmenná. Funkce `get.ballot` vstup v tomto tvaru transformuje do podoby, který může být předán funkcím backendu. Nejprve je textový vstup rozdělen podle řádků, následně každý řádek podle mezery. Seznam takto rozdělených vstupů je následně procházen, pokud v nějakém řádku toto rozdělení nevytvoří právě dva prvky, je vrácen `NULL`, stejně tak, pokud se druhý prvek nepodaří konvertovat na číslo. Pokud tyto kroky uspějí, je první prvek rozdělen na vektor jednotlivých kandidátů, metodami `strsplit`, jež vrací seznam, a `unlist`, která seznam transformuje na vektor. Následně je seznam těchto dvou objektů vložen do předpřipraveného seznamu `ballot`, který je v případě úspěchu výstupní hodnotou této funkce.

```
get.ballot = function(cands) {
  cands = unlist(strsplit(cands, "\n"))
  cands = strsplit(cands, " ")
  ballot = list()
  for (i in 1:length(cands)) {
    cand = cands[[i]]
    if (length(cand) != 2) return(NULL)
    voters = as.numeric(cand[2])
    if (is.na(voters)) return(NULL)
    order = unlist(strsplit(cand[1], ""))
    ballot[[i]] = list(order, voters)
  }
  ballot
}
```

Metoda `get.candidates` z výstupu funkce `get.ballot` extrahuje jména jednotlivých kandidátů, aby je uživatel nemusel zadávat zvlášť. Postup je přímočarý, nejprve se vytvoří vektor všech výskytů jmen v každém prvním prvku objektu `ballot`, na ten je následně aplikována funkce `unique`, jež odstraní duplikátní hodnoty, je setříděn a vrácen.

```
get.candidates = function(ballot) {
  v = vector()
  for (i in 1:(length(ballot))) {
    v = c(v, unlist(ballot[[i]][1]))
  }
  sort(unique(v))
}
```

Uživatelské rozhraní je rozděleno na dvě části, část obsahující vstupy, která je zobrazena neustále, a část s výsledky, jež se zobrazí až po jejich vypočtení. Stránka je uspořádána funkcí `verticalLayout`, která jednotlivé prvky řadí vertikálně. Část se vstupy je jednoduchá, obsahuje pouze jedno textové pole, kam uživatel zadá jednotlivá uspořádání kandidátů a počty hlasů, v tvaru uvedeném výše. Druhá část je horizontálně rozdělena do dvou stejně velkých sloupců. Levý sloupec obsahuje trojici tabulek, realizované funkcí `rHandsontableOutput` z balíčku `rhandsontable`. Jedná se o tabulky párových preferencí, párových preferencí ve tvaru rozdílů, a tabulku nejsilnějších cest. Pravý sloupec obsahuje síťový graf párových preferencí.

```
ui = fluidPage(
  useShinyjs(),
  headerPanel("Schulzeho volební metoda"),
  verticalLayout(
    textAreaInput("ballot", "Kandidáti", width = 400, height = 200,
      resize = "vertical", rows = 10),
    actionButton("run", "Spočti výsledky"),

    hidden(
      tags$div(id = "results",
        column(6,
          tags$b("Tabulka párových preferencí"),
          rHandsontableOutput("pairwise", width = 400), br(),
          tags$b("Tabulka párových preferencí v rozdílovém tvaru"),
          rHandsontableOutput("margins", width = 400), br(),
          tags$b("Tabulka nejsilnějších cest"),
          rHandsontableOutput("strongest", width = 400), br(),
          tags$b("Vítězné uspořádání:"),
          textOutput("winners")),
        column(6,
          tags$b("Párové preference"),
          plotOutput("plot", width = "40vh", height = "40vh"))
      )
    )
  )
)
```

Serverová část obsahuje obsluhu tlačítka `run`, které spouští výpočet, a funkce pro vykreslení jednotlivých výstupů. Nadefinujeme si reaktivní proměnnou `calculated`, která bude obsahovat vlajku značící, zda byl vypočten výsledek. Podle tohoto příznaku se budou vykreslovat

výstupy. Následuje funkce `observeEvent`, jež obsluhuje tlačítko `run`. Po stisku tlačítka je nejprve zavolána funkce `get.ballot`. K dalším krokům se přistoupí, pouze pokud tato funkce proběhla úspěšně. Metodou `get.candidates` jsou získána jména kandidátů. Jsou zavolány funkce backendu pro výpočet a jejich výstupy uloženy do reaktivních proměnných. Nakonec je zavolána funkce knihovny `shinyjs` pro zobrazení části uživatelského rozhraní s výsledky.

```
server = function(input, output, session) {
  values = reactiveValues()
  values$calculated = FALSE
  observeEvent(input$run, {
    ballot = get.ballot(input$ballot)
    if (is.null(ballot)) {
      print("error")
    } else {
      values$cands = get.candidates(ballot)
      values$pair.prefs = pairwise(ballot, values$cands)
      values$margins = values$pair.prefs - t(values$pair.prefs)
      values$strongest.paths = strongest.paths(values$pair.prefs, values$cands)
      values$calculated = TRUE
      values$winners = result(values$strongest.paths, values$cands)
      shinyjs::show("results")
    }
  })
```

Funkce pro vykreslení tabulek a grafu. Pro zobrazení tabulek je použita funkce `rhandsontable` ze stejnojmenného balíčku, za povšimnutí stojí druhá z nich, jež vykresluje tabulku párových preferencí ve tvaru rozdílů. Jež zde nadefinována custom javascriptová funkce, předaná jako parametr `renderer`, sloužící k logickému obarvení jednotlivých buněk tabulky. Samotná JS funkce je triviální, buňky na hlavní diagonále jsou bílé, buňky obsahující kladné číslo světle zelené a buňky se zápornou hodnotou jsou světle červené. Vykreslení grafu je realizováno výše popsanou funkcí backendu `display`.

```
  output$pairwise = renderRHandsontable({
    if (values$calculated) {
      rhandsontable(values$pair.prefs, readOnly = TRUE, stretchH = "all")
    }
  })

  output$margins = renderRHandsontable({
    if (values$calculated) {
      rhandsontable(values$margins, readOnly = TRUE, stretchH = "all") %>%
      hot_cols(renderer = "
        function (instance, td, row, col, prop, value, cellProperties) {
         Handsontable.renderers.NumericRenderer.apply(this, arguments);
         if (row == col) {
          td.style.background = '#ffffff';
         } else if (value < 0) {
```

```
          td.style.background = '#ffdddd';
        } else {
          td.style.background = '#ddffdd';
        }}"
    )
  }
})

output$strongest = renderRHandsontable({
  if (values$calculated) {
    rhandsontable(values$strongest.paths, readOnly = TRUE, stretchH = "all")
  }
})

output$plot = renderPlot({
  if (values$calculated) {
    display(values$pair.prefs)
  }
})

output$winners = renderText({
  values$winners
})
}
```

Zbývá jen spustit aplikaci.

```
runApp(appDir = shinyApp(ui, server), port = 8004, host = "127.0.0.1")
```

7. Numerické metody řešení nelineárních rovnic

Backend pro tuto úlohu není nijak komplikovaný. Obsahuje pouze čtyři funkce, které implementují v teoretické části popsané numerické metody pro hledání odhadu kořenu funkce, plus klasickou metodu pro vykreslení grafu. Je využita knihovna `Deriv`, která slouží pro získání derivace funkce.

```
library(Deriv)
```

Jednotlivé implementace numerických metod jsou přímočaré přepisy algoritmů, uvedených v teoretické části. Jediná zvláštnost stojící za pozornost je, že nevracejí atomickou hodnotu (nejlepší možný odhad), ale vektor všech odhadů. To je z důvodu, že v *Shiny* frontendu budeme chtít možnost měnit počet iterací a pozorovat zpřesňování odhadu a porovnávat rychlost konvergence různých metod. Takto nebude nutné provádět přepočet při každé změně počtu iterací – běžný případ výměny paměti za rychlost.

Implementace metody bisekce:

```
bisection = function(f, a, b, steps = 100) {
  result = vector(length = steps)
  for (i in 1:steps) {
    x = (a + b) / 2
    if (f(a) * f(x) < 0) {
      b = x
    }
    else if (f(x) * f(b) < 0) {
      a = x
    }
    result[i] = x
  }
  result
}
```

Implementace Newtonovy metody:

```
newton = function(f, x, steps = 100) {
  df = Deriv(f)
  result = vector(length = steps)
  for (i in 1:steps) {
```

```
    y = f(x)
    yd1 = df(x)
    x2 = x - y / yd1
    result[i] = x2
    x = x2
  }
  return(result)
}
```

Metoda sečen:

```
secant = function(f, a, b, steps = 100){
  result = vector(length = steps)
  for (i in 1:steps) {
    x = b - f(b) * (b - a) / (f(b) - f(a))
    a = b
    b = x
    result[i] = if (!is.na(x)) x else result[i-1]
  }
  result
}
```

A metoda regula falsi:

```
regula.falsi = function(f, a, b, steps = 100) {
  result = vector(length = steps)
  for (i in 1:steps) {
    x = b - f(b) * ((b - a) / (f(b) - f(a)))
    if (f(a) * f(x) < 0) {
      b = x
    } else if (f(x) * f(b) < 0) {
      a = x
    }
    result[i] = x
  }
  result
}
```

Funkce pro vykreslení grafu přijímá čtveřici parametrů. Prvním, nazvaným `f` je samotná funkce, na které hledáme kořen. Hodnoty `xlim` a `ylim` definují na jakém rozmezí má být graf vykreslen (v obou případech jde o právě dvouprvkové číselné vektory). Parametr `legend` je booleanovský příznak říkající, zda má být vykreslena legenda. Posledním parametrem `l` je seznam obsahující prvky `b`, `n`, `s` a `r`, ve kterých jsou uloženy x hodnoty aproximující kořen funkce, nalezené odpovídajícími numerickými metodami.

Nejprve je vytvořen prázdný graf (skrze parametr `type = "n"` pro vestavěnou funkci `plot`) o odpovídajících rozměrech. Do grafu je přidána mřížka a v závislosti na hodnotě `legend` také

legenda. Funkcí `abline` je vykreslena nulová osa. Voláním metody `curve` je vykreslena křivka funkce `f`. Jelikož tato funkce je jedním ze vstupů zadaných uživatelem, nemusí jít o platný výraz a pokus o její vykreslení může skončit chybou. Volání metody `curve` je tedy obaleno konstruktem `tryCatch`, který zajišťuje, že případná chyba nezpůsobí pád celého programu. Nakonec je funkcí `abline` vykreslena čtveřice svislých čar indikující polohu odhadů kořenu funkce jednotlivými numerickými metodami.

```
display = function(f, xlim, ylim, legend, l) {
  par(col.lab = "transparent", mar = c(3, 3, 1, 1))
  plot(0, 0, type = "n", xlim = xlim, ylim = ylim)
  grid()
  if (legend) {
    legend(x = "topleft",
      c("Bisekce", "Regula falsi", "Newtonova metoda", "Metoda sečen"),
      col = c("green", "orange", "blue", "red"),
      lwd = 4, bg = "white")
  }
  abline(h = 0, col = "grey")
  tryCatch({
    curve(f, from = xlim[1], to = xlim[2], add = TRUE, col = "black")
  }, error = function(cond) print("error"))
  abline(v = l$b, col = "green")
  abline(v = l$n, col = "blue")
  abline(v = l$s, col = "red")
  abline(v = l$r, col = "orange")
}
```

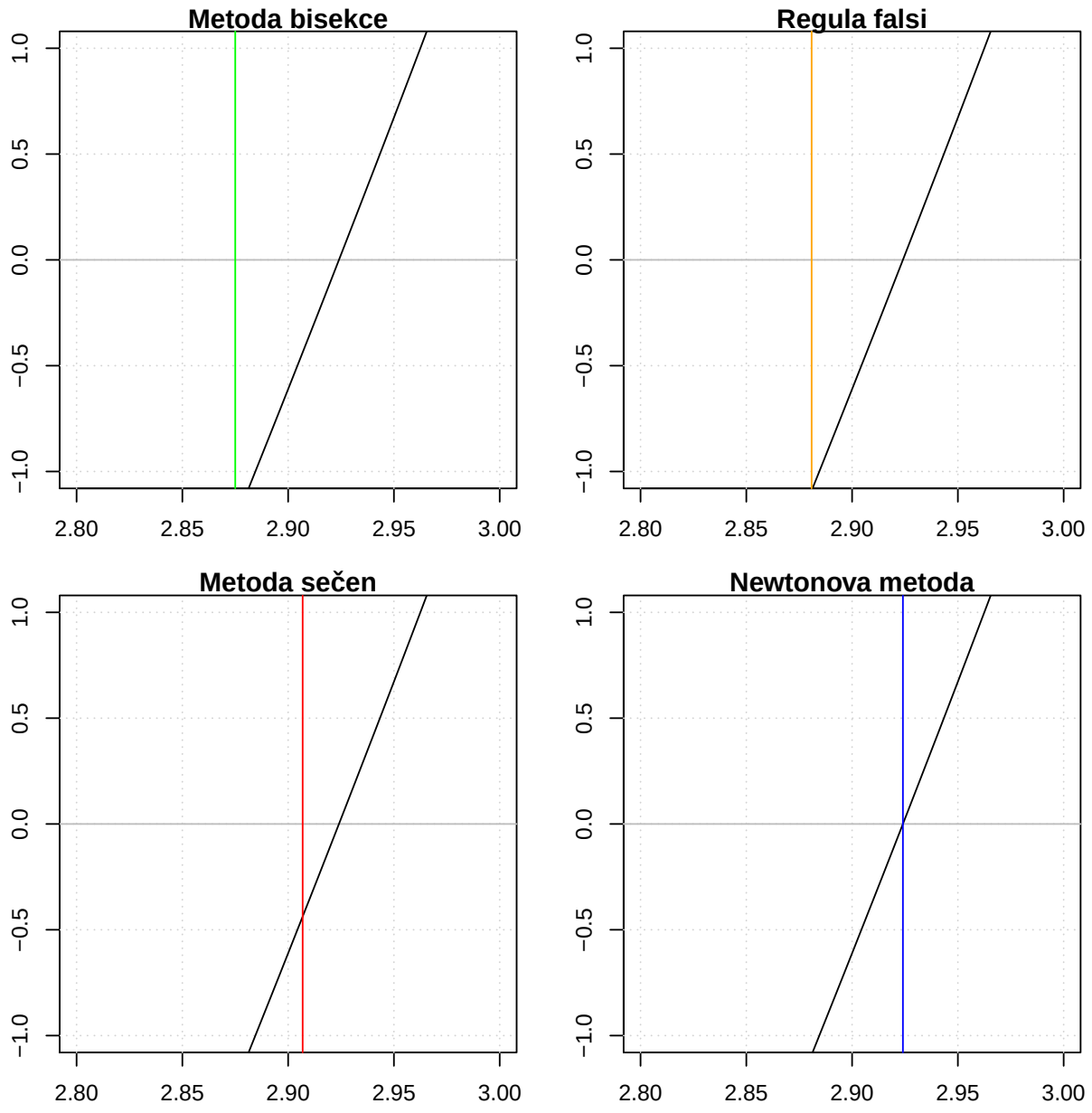
7.1. knitr

V rámci ukázky *knitr* frontendu vykreslíme čtveřici grafů, každý se stejnou funkcí, obsahující odhady získané jednotlivými metodami. Maximální počet iterací je záměrně malý, aby bylo možno porovnat přesnost, respektive rychlost konvergence různých postupů.

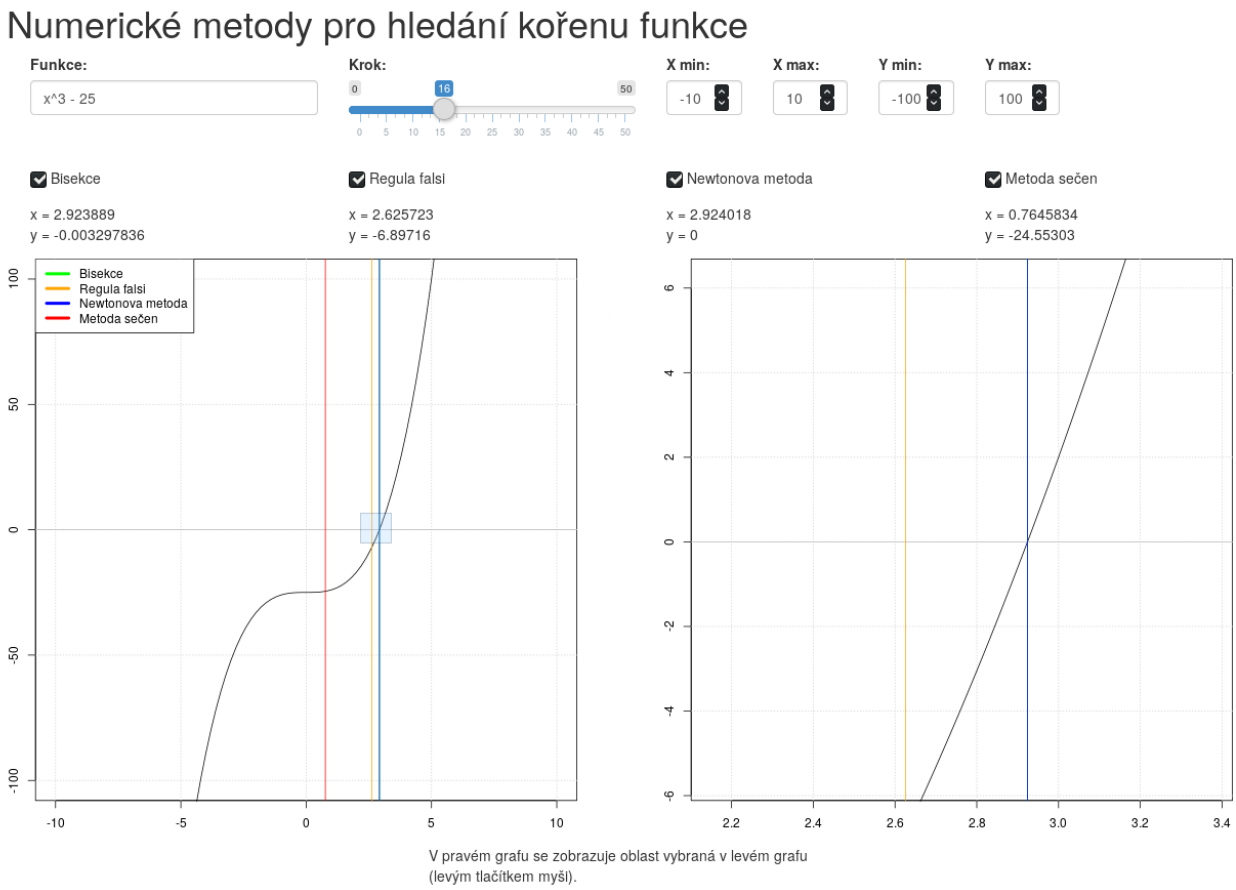
Nejprve si nadefinujeme funkci, na které budeme hledat kořen. Vytvoříme si proměnné, které budou definovat rozsah vykreslené oblasti. Poté už jen vypočteme odhady jednotlivými metodami a v každém grafu vykreslíme poslední prvek vektoru, vráceného každou z metod.

```
f = function(x) x^3 - 25
xmin = 0.5
xmax = 4.5
ymin = -1
ymax = 1
step = 5
xlim = c(2.8, 3)
ylim = c(ymin, ymax)
```

```
b = bisection(f, xmin, xmax, step)
n = newton(f, xmax/2, step)
s = secant(f, xmin, xmax, step)
r = regula.falsi(f, xmin, xmax, step)
par(mfrow=c(2,2))
display(f, xlim, ylim, FALSE, list(b = b[step]))
title("Metoda bisekce")
display(f, xlim, ylim, FALSE, list(r = r[step]))
title("Regula falsi")
display(f, xlim, ylim, FALSE, list(s = s[step]))
title("Metoda sečen")
display(f, xlim, ylim, FALSE, list(n = n[step]))
title("Newtonova metoda")
```



7.2. Shiny



Obrázek 7.1.: Hledání kořenu funkce – Shiny aplikace

Provedeme klasický import knihovny `shiny` a backendu. Následně v globálním prostředí nadeklarujeme konstanty, které využijeme v grafickém rozhraní – jedná se o velikost jednotlivých grafů a odsazení `div` elementů s widgety pro uživatelské vstupy.

```
library(shiny)
source("./root.R")

size = "65vh"
style = "margin-left: 3vw"
```

Uživatelské rozhraní této aplikace bude poněkud odlišné od předchozích. Jak lze vidět na přiloženém obrázku, stránka je rozdělena do čtyř řádků (nepočítaje nadpis). První dva řádky obsahují ovládací prvky, následují textové výstupy, ukazující aktuální odhad hodnoty kořene funkce jednotlivými metodami, a v posledním řádku je dvojice grafů.

Pro vytvoření stránky opět použijeme funkci `fluidPage`. Na vytvoření jednotlivých řádků využijeme `fluidRow` a prostor uvnitř řádků rozdělíme funkcí `column`. Na stránce se nachází

nadpis, realizovaný funkcí `headerPanel`. Pod nadpisem je první řádek, obsahující `textInput` pro zadání předpisu funkce, kterou chceme zkoumat. Následující vstup, `sliderInput` pojmenovaný step udává počet iterací – jeho zvyšováním můžeme zpřesňovat odhad kořene a porovnávat rychlost konvergence jednotlivých numerických metod. Následuje čtveřice číselných vstupů pro nastavení rozsahu, který má být zobrazován hlavním grafem.

```
ui = fluidPage(
  headerPanel("Numerické metody pro hledání kořenu funkce"),
  fluidRow(
    div(style = style,
      column(3,
        textInput("func", "Funkce:", value = "x^3 - 25")),
      column(3,
        sliderInput("step", "Krok:", min = 0, max = 50, value = 1)),
      column(1,
        numericInput("xmin", "X min:", value = -10, min = -100, max = 100, step = 1)),
      column(1,
        numericInput("xmax", "X max:", value = 10, min = -100, max = 100, step = 1)),
      column(1,
        numericInput("ymin", "Y min:", value = -100, min = -100, max = 100, step = 1)),
      column(1,
        numericInput("ymax", "Y max:", value = 100, min = -100, max = 100, step = 1))
  )),
```

Druhý řádek je jednoduchý. Je zde pouze čtveřice checkboxů, skrze které lze zvolit metody, jejichž výstupy mají být vykresleny do grafu. Defaultně jsou zaškrtnuty všechny čtyři.

```
  fluidRow(
    div(style = style,
      column(3,
        checkboxInput("bisec", "Bisekce", value = TRUE)),
      column(3,
        checkboxInput("rfalsi", "Regula falsi", value = TRUE)),
      column(3,
        checkboxInput("newton", "Newtonova metoda", value = TRUE)),
      column(3,
        checkboxInput("secant", "Metoda sečen", value = TRUE))
    )
  ),
```

Dále je zde řádek s textovými výstupy. Řádek je rozdělen na čtyři stejně velké části, každá obsahující informace o jedné ze čtyř numerických metod. Každý tento díl je vytvořen podle stejného vzoru: funkcí `b` ze třídy `tags` je vytvořen popisek. Třída `tags` nám dává přístup k řadě funkcí, které implementují valnou část HTML tagů v podobě funkcí, které lze snadno volat z R. Funkce `HTML` přijímá textový řetězec, který vyhodnotí jako HTML kód. Následují elementy `textOutput`, jejichž obsah je dynamicky generován a měněn funkcemi `renderText` ze serverové části *Shiny* aplikace.

```
fluidRow(
  div(style = style,
    column(3,
      HTML("x = "),
      textOutput("bisec.x", inline = TRUE),
      br(), HTML("y = "),
      textOutput("bisec.y", inline = TRUE)),
    column(3,
      HTML("x = "),
      textOutput("rfalsi.x", inline = TRUE),
      br(), HTML("y = "),
      textOutput("rfalsi.y", inline = TRUE)),
    column(3,
      HTML("x = "),
      textOutput("newton.x", inline = TRUE),
      br(), HTML("y = "),
      textOutput("newton.y", inline = TRUE)),
    column(3,
      HTML("x = "),
      textOutput("secant.x", inline = TRUE),
      br(), HTML("y = "),
      textOutput("secant.y", inline = TRUE))
  )
),
```

Poslední řádek je krátký, ale asi nejzajímavější. Obsahuje dva grafické výstupy, `main.plot` a `side.plot`. Chceme, aby uživatel mohl v hlavním grafu ohraničit oblast, která se mu ve vedlejším grafu zobrazí přiblížená. Za tím účelem hlavnímu grafu nastavíme parametr `brush`, což je jeden ze čtyř parametrů umožňujících v *Shiny* vytvářet interaktivní grafy. Konkrétně `brush` umožňuje uživateli kliknout a táhnout skrze graf a vytvořit tak oblast výběru, k jejímž definujícím souřadnicím můžeme přistupovat v serverové části. Hlavním, ale zdaleka ne jediným, využitím tohoto konstruktu je přibližování grafu. Další možnosti interaktivních grafů uvidíme v ostatních aplikacích.

```
fluidRow(
  column(6, plotOutput("main.plot", width = size, height = size,
    brush = brushOpts(id = "brush", resetOnNew = TRUE))
  ),
  column(6,
    plotOutput("side.plot", width = size, height = size)
  )
),
fluidRow(
  column(4),
  column(4,
    tags$p("V pravém grafu se zobrazuje oblast vybraná v levém grafu "),
```

```
      tags$p("(levým tlačítkem myši)."),
    column(4)
  )
)
```

Opět si běžným způsobem vytvoříme funkci, která bude realizovat serverovou část aplikace. Zde si nadefinujeme dvě nereaktivní proměnné, které budou držet aktuální hodnotu výběru v grafu, neboť hodnota obsažená ve vstupu se smaže, pokud graf překreslíme (např. při změně počtu iterací), ale my budeme chtít, aby přiblížení zůstalo zachováno. Vytvoříme si také kontejner pro reaktivní hodnoty.

```
server = function(input, output) {
  brush.cache.x = NULL
  brush.cache.y = NULL
  rv = reactiveValues()
```

První reaktivní výraz zpracovává textový vstup. Zde uživatel zadává předpis funkce (s pouze jednou proměnnou nazvanou x). Tento textový vstup od uživatele spojíme s dalším textem, který nám vytvoří validní zápis funkce v R, a funkcemi `eval` a `parse` z tohoto textového řetězce vytvoříme skutečnou funkci, se kterou můžeme v kódu pracovat. Jelikož uživatel může do textového pole zadat libovolný vstup, nemusela by být funkce úspěšně vyhodnocena, proto musí být její vytváření obaleno v `tryCatch` bloku. Následně funkcí `exists` otestujeme, zda byla uživatelská funkce úspěšně vytvořena. Povšimněte si, že `exists` přijímá textový řetězec. Pokud funkce existuje, uložíme ji jako reaktivní hodnotu.

```
  observe({
    tryCatch({
      f = eval(parse(text = paste("function(x) { ", input$func, " }", sep="")))
      x = input$xmin:input$xmax
      f(x)
    }, error = function(cond) {
      rm(f)
      print("Error parsing user submitted function.")
    })

    if (exists("f")) {
      rv$f = f
    }
  })
```

Na vytvoření funkce `f` zareaguje následující reaktivní výraz, který vypočte a uloží jako reaktivní výrazy odhady kořenu funkce `f` vytvořené jednotlivými metodami. Naše implementace těchto metod nevracejí pouze atomickou hodnotu, ale celý vektor obsahující každý odhad provedený v iterativním procesu. Následující observer pak z tohoto grafu vezme hodnotu, odpovídající aktuálně zvolenému kroku (čili počtu iterací). Není tedy nutné provádět celý výpočet znovu při každé změně kroku, pouze při změně funkce.

```
observe({
  tryCatch({
    guess = input$xmax / 2
    rv$b = bisection(rv$f, input$xmin, input$xmax)
    rv$n = newton(rv$f, guess)
    rv$s = secant(rv$f, input$xmin, input$xmax)
    rv$r = regula.falsi(rv$f, input$xmin, input$xmax)
  }, error = function(cond) print("error"))
})

observe({
  rv$b.val <<- rv$b[input$step]
  rv$n.val <<- rv$n[input$step]
  rv$s.val <<- rv$s[input$step]
  rv$r.val <<- rv$r[input$step]
})
```

Tento výraz pouze spojuje minimální a maximální hodnoty výběru v grafu do jednoho vektoru a ten ukládá jako reaktivní proměnnou.

```
observe({
  rv$range.x = if (is.null(input$brush)) NULL
    else c(input$brush$xmin, input$brush$xmax)

  rv$range.y = if (is.null(input$brush)) NULL
    else c(input$brush$ymin, input$brush$ymax)
})
```

Funkce `vals` zabaluje aktuální hodnoty odhadu kořene do seznamu, který je předán vykreslovací funkci. Následuje dvojice reaktivních výrazů, které slouží k vykreslení hlavního a vedlejšího grafu. Vykreslení hlavního grafu je jednoduché: z uživatelských vstupů se získá rozsah, který má být vykreslen, příznak pro vykreslení legendy se nastaví na `TRUE` a funkcí `vals` se získají hodnoty odhadů pro vykreslení. Následně se pouze zavolá backendová metoda `display`.

Vykreslení vedlejšího grafu je komplikováno požadavkem na zoom a tím, že hodnoty pro rozsah oblasti pro vykreslení mohou přijít z několika zdrojů. Nejprve je přečten obsah reaktivních hodnot `range.x` a `range.y`, které pouze zprostředkovávají aktuální výběr. Následný `if-else` blok přečte obsah cache, pokud je aktuální výběr `NULL`, v opačném případě do cache uloží aktuální výběr. Pokud je aktuální výběr i cache prázdná (zpravidla hned po spuštění aplikace), jsou přečteny hodnoty, které udávají rozsah pro hlavní graf. Následně je opět zavolána funkce `display`, tentokrát ovšem s příznakem legendy nastaveným na `FALSE`.

```
vals = function() {
  b = if (input$bisec)  rv$b.val else NULL
  n = if (input$newton) rv$n.val else NULL
```

```
    s = if (input$secant) rv$s.val else NULL
    r = if (input$rfalsi) rv$r.val else NULL
    list(b = b, n = n, s = s, r = r)
  }

  output$main.plot = renderPlot({
    xlim = c(input$xmin, input$xmax)
    ylim = c(input$ymin, input$ymax)
    display(rv$f, xlim, ylim, TRUE, vals())
  })

  output$side.plot = renderPlot({
    xlim = rv$range.x
    ylim = rv$range.y

    if (!is.null(xlim) && !is.null(ylim)) {
      brush.cache.x <<- xlim
      brush.cache.y <<- ylim
    } else {
      xlim = brush.cache.x
      ylim = brush.cache.y
    }

    xlim = if (!is.null(xlim)) xlim else c(input$xmin, input$xmax)
    ylim = if (!is.null(ylim)) ylim else c(input$ymin, input$ymax)

    display(rv$f, xlim, ylim, FALSE, vals())
  })
```

Zbývá vypsat hodnoty x a y pro jednotlivé metody. Tento kód je přímočarý:

```
  output$bisec.x = renderText(rv$b.val)
  output$bisec.y = renderText(rv$f(rv$b.val))
  output$rfalsi.x = renderText(rv$r.val)
  output$rfalsi.y = renderText(rv$f(rv$r.val))
  output$newton.x = renderText(rv$n.val)
  output$newton.y = renderText(rv$f(rv$n.val))
  output$secant.x = renderText(rv$s.val)
  output$secant.y = renderText(rv$f(rv$s.val))
}
```

Aplikaci opět spustíme běžným způsobem:

```
runApp(appDir = shinyApp(ui, server), port = 8003, host = "127.0.0.1")
```

8. Detekce rozhraní pomocí Delaunayovy triangulace

Výpočet `Delaunayovy triangulace` je implementován v souboru `delaunay.R`. Využívá knihovny `tripack` pro provedení triangulace nad sítí bodů. Knihovnu importujeme funkcí `library`:

```
library(tripack)
```

Ve funkci `delaunay` je realizována triangulace, výpočet poloměrů opsané kružnice pro každý trojúhelník, a převedení dat do formátu, se kterým se dá lépe pracovat. Tato funkce přijímá na vstupu dvojici numerických vektorů `x` a `y`. Ty vektory přirozeně popisují polohu jednotlivých bodů, nad kterými budeme provádět triangulaci. Nejprve je provedena triangulace, a to voláním funkcí knihovny `tripack`: `tri.mesh` (provádí samotnou triangulaci) a `triangles` (slouží k extrahování dat o jednotlivých trojúhelnících). Následně je potřeba výstup funkce `triangles` převést do tvaru, s kterým budou pracovat další funkce. Výstupem naší custom funkce `delaunay` má být dataframe, kde každý řádek popisuje jeden trojúhelník triangulace, a deseti sloupci: x a y souřadnice každého ze tří vrcholů trojúhelníku, indexy sousedících trojúhelníků (sdílejících vždy jednu hranu) a poloměr opsané kružnice daného trojúhelníku. Přitom výstupem funkce `triangles` je dataframe s podobnou strukturou, jen místo samotných souřadnic vrcholů trojúhelníků obsahuje indexy do vstupních vektorů `x` a `y`. Převedení prvních devíti sloupců do požadovaného tvaru je přímočaré. Následně je potřeba vypočíst poloměr opsané kružnice – k tomu poslouží opět funkce knihovny `tripack`, nazvaná `circumcircle`. Nejprve vytvoříme vektor o délce odpovídající počtu trojúhelníků, poté v cyklu procházíme již vytvořený dataframe obsahující jednotlivé trojúhelníky a pro každý vypočteme poloměr opsané kružnice. Vypočtenou hodnotu uložíme do připraveného vektoru a po skončení cyklu tento vektor připojíme jako desátý sloupec do již vytvořeného dataframeu. Celá funkce vypadá takto:

```
delaunay = function(x, y) {
  t = triangles(tri.mesh(x, y))
  x0 = x[t[,1]]
  x1 = x[t[,2]]
  x2 = x[t[,3]]
  y0 = y[t[,1]]
  y1 = y[t[,2]]
  y2 = y[t[,3]]
  n0 = t[,4]
  n1 = t[,5]
  n2 = t[,6]
  trins = data.frame(x0, x1, x2, y0, y1, y2, n0, n1, n2)
```

```
  l = nrow(trins)
  c = vector(length = l)
  for (i in 1:l) {
    c[i] = circumcircle(
      c(x0[i], x1[i], x2[i]),
      c(y0[i], y1[i], y2[i])
    )$radius
  }
  trins$c = c
  trins
}
```

Druhou zásadní funkcí v této úloze je nalezení rozhraní v triangulaci `t` podle hodnoty `threshold`. `Threshold` nám udává hranici, která dělí trojúhelníky triangulace do dvou skupin: trojúhelníky, u kterých poloměr opsané kružnice je menší než `threshold`, a trojúhelníky, u kterých je větší. Pro naše potřeby (vizualizace) se budeme zajímat především o jednotlivé hrany, které toto rozhraní tvoří, tedy takové hrany, které jsou sdíleny vždy jedním trojúhelníkem z každé skupiny. Každá hrana bude definována právě čtyřmi hodnotami, `x0`, `y0`, `x1` a `y1`, uložených v odpovídajících vektorech. Definujeme si pomocnou funkci, `find.interface`, která pracuje vždy se dvěma trojúhelníky, porovná poloměry jejich opsaných kružnic, a pokud tvoří rozhraní, nalezne jejich sdílenou hranu a definující body uloží do odpovídajících vektorů. Po definování této funkce stačí procházet jednotlivé trojúhelníky a pro každou dvojici (aktuální trojúhelník, jeden ze sousedů) zavoláme metodu `find.interface`. Samotná metoda `find.interface` obsahuje podmínku, která zajistí, že nedojde k opakování nad tou samou dvojicí trojúhelníků. Výstupem funkce jsou vektory `x0`, `y0`, `x1` a `y1` vrácené v seznamu.

```
interface = function(t, threshold) {
  x0 = vector()
  x1 = vector()
  y0 = vector()
  y1 = vector()
  find.interface = function(i, n) {
    if (n > i) {
      c2 = t[n,]$c
      if (((c1 - threshold) * (c2 - threshold)) < 0) {
        x = intersect(t[i,1:3], t[n,1:3])
        y = intersect(t[i,4:6], t[n,4:6])
        x0 <<- c(x0, x[[1]])
        x1 <<- c(x1, x[[2]])
        y0 <<- c(y0, y[[1]])
        y1 <<- c(y1, y[[2]])
      }
    }
  }
  for (i in 1:nrow(t)) {
    c1 = t[i,]$c
```

```
    n = t[i,]$n0
    find.interface(i, n)
    n = t[i,]$n1
    find.interface(i, n)
    n = t[i,]$n2
    find.interface(i, n)
  }
  list(x0 = x0, x1 = x1, y0 = y0, y1 = y1)
}
```

Funkce `display` slouží k vytvoření grafu. Na vstupu přijímá triangulaci `t`, seznam hran tvořících rozhraní `interfaces` a hodnotu hranice `threshold`. Nejprve jsou nastaveny obecné grafické parametry funkcí `par`, konkrétně průhledná barva pro popředí grafu, osy a popisy, a jednotkové okraje. Funkcí `plot` je jednak vytvořen nový graf a současně vykreslený jednotlivé uzly. Opakované volání funkce `segments`, která slouží ke kreslení úseček do grafu, je použito k vykreslení všech hran tvořících trojúhelníky. Funkce `polygon` je použita k vybarvení trojúhelníků uvnitř rozhraní poloprůhlednou červenou barvou. Nakonec je opět použita funkce `segments`, tentokrát k vykreslení samotného rozhraní.

```
display = function(t, interfaces, threshold) {
  par(
    fg = "transparent",
    col.axis = "transparent",
    col.lab = "transparent",
    mar = c(1, 1, 1, 1))
  plot(t[,1], t[,4], pch = 20, col = "black")
  segments(t$x0, t$y0, t$x1, t$y1, col = "black")
  segments(t$x2, t$y2, t$x0, t$y0, col = "black")
  segments(t$x2, t$y2, t$x1, t$y1, col = "black")
  for (i in 1:nrow(t)) {
    if (t[i,]$c < threshold) {
      polygon(
        x = c(t[i,]$x0, t[i,]$x1, t[i,]$x2),
        y = c(t[i,]$y0, t[i,]$y1, t[i,]$y2),
        col = rgb(255, 0, 0, max = 255, alpha = 125)
      )
    }
  }
  segments(
    x0 = interfaces$x0,
    x1 = interfaces$x1,
    y0 = interfaces$y0,
    y1 = interfaces$y1,
    col = "red", lwd = 2)
}
```

Druhou grafickou funkcí je `plot.circums`, která slouží k vykreslení grafu poloměrů všech opsaných kružnic. Opsané kružnice jsou nejprve vzestupně seřazeny. Opět jsou nastaveny grafické parametry, poloměry jsou vykresleny do grafu, nakonec je ještě do toho samého grafu svislou červenou čarou zakreslena aktuální hodnota hranice `threshold`.

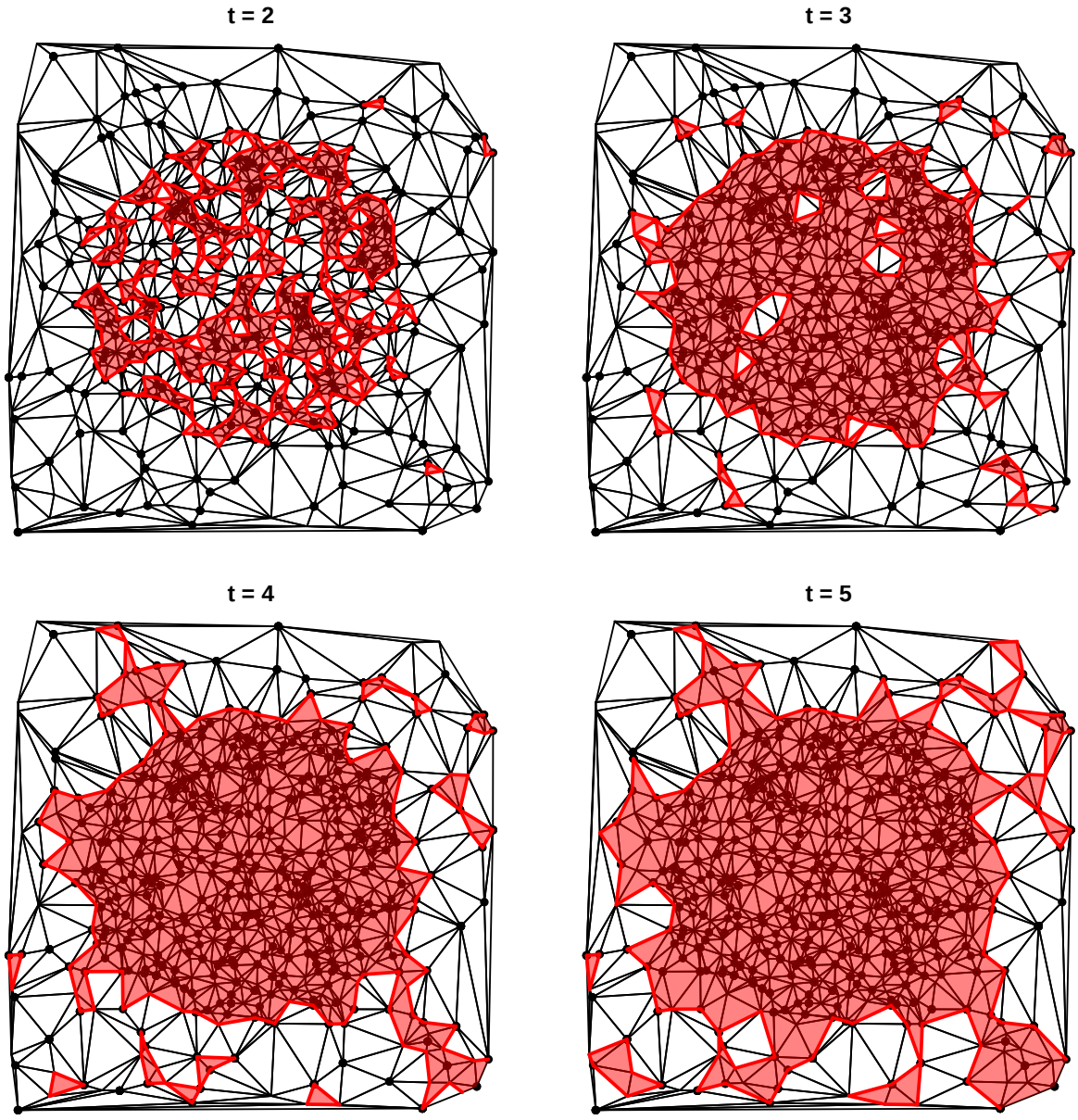
```
plot.circums = function(t, threshold) {
  c = sort(t$c)
  x = 1:length(c)
  par(
    fg = "black",
    col.axis = "transparent",
    col.lab = "transparent",
    mar = c(1, 1, 1, 1))
  plot(x, c, type = "l", xaxt = "n", , col = "black")
  for (i in 1:length(c)) {
    if (c[i] > threshold) {
      abline(v = i, col = "red")
      break
    }
  }
}
```

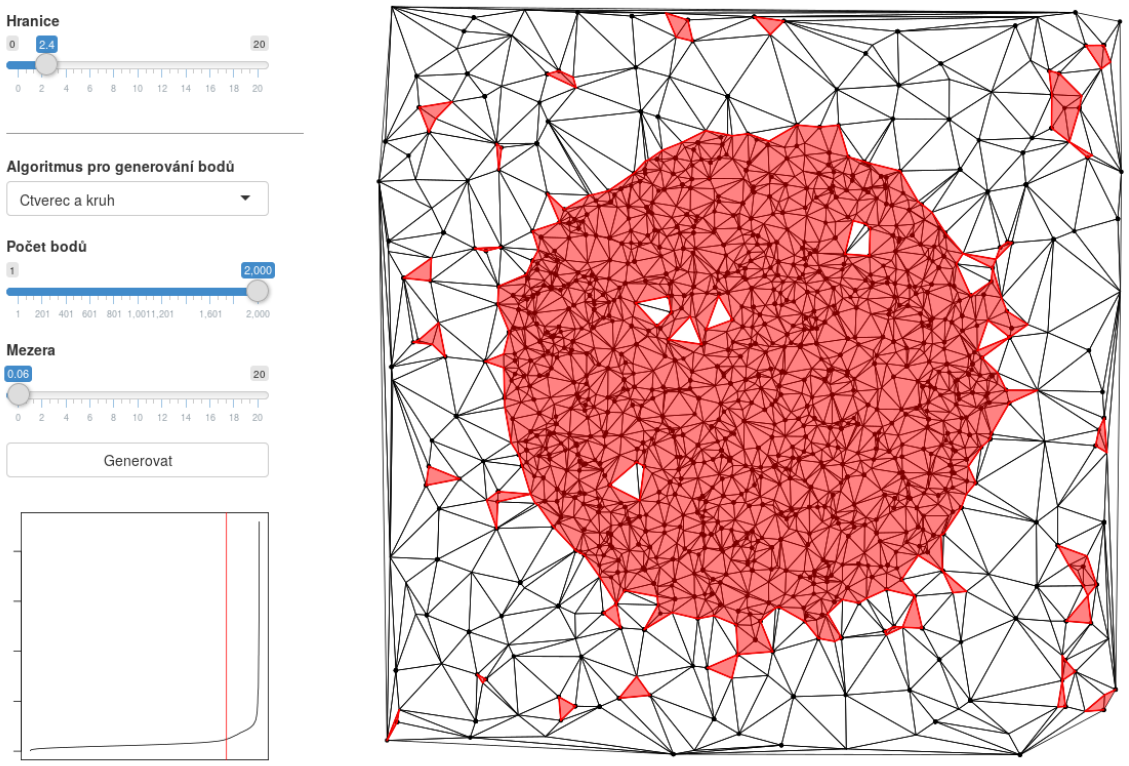
8.1. knitr

V této ukázce vytvoříme čtveřici grafů nad stejnou množinou bodů, s hodnotami hranice pro rozhraní 2, 3, 4 a 5. Pro vytvoření bodů, nad kterými budeme provádět triangulace, použijeme funkci definovanou v souboru *point-gen.R*, nazvanou `CtverecKruh`, která vygeneruje 1000 bodů v intervalu [0, 100] s minimálními mezerami 1. Dále nastavíme grafické parametry specifické pro tento graf: velikost popisků a to, že graf má mít dva řádky a dva sloupce. Zbytek kódu je očividný:

```
p = CtverecKruh(1000, 1, 100)
d = delaunay(p$x, p$y)

par(ps = 10, mfrow = c(2,2))

t = 2
i = interface(d, t)
display(d, i, t)
title(paste("t = ", t, sep = ""))

t = 3
i = interface(d, t)
display(d, i, t)
```

```
title(paste("t = ", t, sep = ""))

t = 4
i = interface(d, t)
display(d, i, t)
title(paste("t = ", t, sep = ""))

t = 5
i = interface(d, t)
display(d, i, t)
title(paste("t = ", t, sep = ""))
```





Obrázek 8.1.: Delaunayovo rozhraní – Shiny aplikace

8.2. Shiny

Každá *Shiny* aplikace musí importovat knihovnu `shiny`. Importujeme také knihovnu `shinythemes`, obsahující několik CSS stylů, které můžeme v naší aplikaci použít. Importujeme také soubor realizující backend této úlohy a soubor obsahující funkce pro generování bodů.

```
library(shiny)
library(shinythemes)
source("./delaunay.R")
source("./point-gen.R")
```

Proměnná `ui` popisuje grafické uživatelské rozhraní naší aplikace. K jeho vytvoření využijeme řady funkcí, poskytovaných knihovnou `shiny`. Rozhraní má stromovou strukturu (je přeloženo do HTML) a tedy je možné funkce k jeho vytváření libovolně skládat. Nejvrchnější prvek je vytvořen funkcí `fluidPage`, což je základní funkce pro definování uživatelského rozhraní v *Shiny* aplikaci. Tato stránka obsahuje právě jeden řádek (funkce `fluidRow`, zajišťující, že všechny uvnitř obsažené elementy budou na stejné horizontální úrovni), rozdělený do dvou sloupců. Jak bylo zmíněno, defaultní uživatelské rozhraní v *Shiny* vychází z CSS frameworku Bootstrap, který stránku dělí na dvanáct svislých dílů, proto je šířka jednotlivých sloupců definována čísly, jejichž součet musí vycházet roven dvanácti. První sloupec má tvořit čtvrtinu stránky, proto je mu dána šířka 3, šířka druhého sloupce je dána dopočtem

do 12. Prvky v prvním sloupci jsou uspořádány do vertikálního layoutu. Funkce `sliderInput` vytváří elementy sloužící k zadávání číselných vstupních proměnných skrze posuvné ovládací prvky. `actionButton` vytváří běžné tlačítko. Posledním prvkem v tomto sloupci je plocha, do které bude vykreslen jeden z grafických výstupů, graf jednotlivých poloměrů opsaných kružnic.

Druhý sloupec obsahuje pouze jeden prvek, plochu pro vykreslení hlavního grafu.

```
ui = fluidPage(
  fluidRow(
    column(3,
      verticalLayout(
        br(), br(),
        sliderInput("threshold", "Hranice", min = 0, max = 20,
          value = 10, step = 0.01, width = "20vw"),
        tags$hr(style="border-color: #999999;"),
        selectInput("sel.algorithm", "Algoritmus pro generování bodů",
          algorithms, selected = NULL, multiple = FALSE, width = "20vw"),
        sliderInput("points", "Počet bodů", min = 1, max = 2000,
          value = 500, step = 1, width = "20vw"),
        sliderInput("gap", "Mezera", min = 0, max = 20,
          value = 1, step = 0.01, width = "20vw"),
        actionButton("btn.generate", "Generovat", width = "20vw"),
        br(),
        plotOutput("plot.circums", width = "30vh", height = "30vh")
      )
    ),
    column(9,
      div(class = "plot",
        plotOutput("plot", width = "90vh", height = "90vh")
      )
    )
  )
)
```

Funkce `server` realizuje veškerou logiku aplikace. Přijímá parametry `input` a `output`. Obojí jsou seznamy obsahující aktuální hodnoty vstupních prvků (input) a proměnné, do kterých se ukládají výstupy (output).

Nadefinujeme si proměnnou `rv` typu `reactiveValues`. Jedná se o proměnné, které fungují v rámci reaktivního paradigmatu *Shiny* aplikace, to znamená, že změna hodnoty reaktivní proměnné spustí přehodnocení všech funkcí, které s touto proměnnou pracují. Zatím zůstane prázdná, ale později do ní uložíme vygenerovanou triangulaci – v důsledku toho se při každé změně automaticky překreslí grafy, které berou tuto triangulaci jako vstup (*source* v terminologii *Shiny*).

Dále funkce `server` obsahuje trojici reaktivních výrazů. První dva definují, jak mají vypadat grafy. Nejsou nijak složité, jednoduše pokud existuje objekt triangulace, zavolají odpovídající

funkci backendu a předají ji potřebné proměnné. *Shiny* automaticky zajišťuje, že grafy jsou vykresleny do odpovídajících výstupních elementů.

Třetím reaktivním výrazem je obsluha události stisknutí tlačítka. Nejprve je vytvořena funkce pro generování bodů na základě hodnoty vybrané uživatelem v `selectInputu` – funkce je z jejího názvu vytvořena skrze vestavěnou funkci `get`. Tato metoda je použita k vygenerování uživatelem zadaného počtu bodů, nad nimi je provedena triangulace a její výsledek je uložen do výše nadefinovaného seznamu reaktivních proměnných.

```
server = function(input, output) {
  rv = reactiveValues()

  output$plot = renderPlot({
    if (!is.null(rv$del)) {
      i = interface(rv$del, input$threshold)
      display(rv$del, i, input$threshold)
    }
  })

  output$plot.circums = renderPlot({
    if (!is.null(rv$del)) {
      plot.circums(rv$del, input$threshold)
    }
  })

  observeEvent(input$btn.generate, {
    gen = get(input$sel.algorithm);
    p = gen(input$points, input$gap, 100)
    rv$del = delaunay(p$x, p$y)
  })
}
```

Shiny aplikaci nakonec spustíme funkcí `runApp`:

```
runApp(appDir = shinyApp(ui, server), port = 8002, host = "127.0.0.1")
```

9. Isingův model

Celý Isingův model je implementován v souboru *ising.R*. Nejprve nadefinujeme funkci pro generování čtvercového pole o zadané velikosti, obsahující prvky s hodnotou 1 nebo -1. Distribuce těchto prvků je náhodná a rovnoměrná, reprezentují spiny magnetického pole v Isingově modelu. Celkový počet spinů ve čtvercovém poli velikosti n bude n^2. Použijeme vestavěnou funkci `runif`, která vygeneruje takový počet hodnot v rozsahu 0 až 1. Následně na celý vektor vytvořený touto funkcí aplikujeme anonymní funkci, která hodnoty menší než 0.5 převede na hodnotu -1, a hodnoty větší než 0.5 převede na 1. Funkcí `matrix` z tohoto vektoru vytvoříme matici velikosti $n * n$.

```
spin.arr = function(size) {
  cells.count = size ** 2
  cells = runif(cells.count, 0, 1)
  s = sapply(cells, function(x) {
    if (x < 0.5) -1 else 1
  })
  matrix(s, size, size)
}
```

Isingův model popisuje chování systému, které vede ke snižování celkové energie a eventuálně dosažení rovnovážného stavu. Jelikož celková energie systému je jednou z hodnot, které chceme měřit / zobrazovat uživateli, vytvoříme si funkci, která ji bude počítat. Jedná se o přímočarou implementaci vzorce pro výpočet celkové energie, uvedeného v teoretické části.

```
total.energy = function(S, J, H) {
  n = nrow(S)
  si = 0
  se = 0
  for (i in 1:(n-1)) {
    for (j in 1:(n-1)) {
      si = si + (S[i, j] * S[i+1, j])
      si = si + (S[i, j] * S[i, j+1])
      se = se + S[i, j]
    }
    si = si + (S[i, n] * S[i, 1])
    si = si + (S[n, i] * S[1, i])
  }
  E = -(J*si)-(H*se)
}
```

Hlavní jádro simulace, funkce, která provede jeden krok – to znamená, vybere náhodný prvek z předaného pole, provede výpočet daný Isingovým modelem a podle výsledku tohoto výpočtu rozhodne, zda má či nemá převrátit hodnotu vybraného prvku. Následně vrátí (ne)upravenou matici.

```
single.step = function(S, T, J = 1, H = 0) {
  size = nrow(S)
  i = sample(1:size, 1)
  j = sample(1:size, 1)
  n = 0
  n = if (i > 1) n + S[i-1, j] else n + S[size, j]
  n = if (i < size) n + S[i+1, j] else n + S[1, j]
  n = if (j > 1) n + S[i, j-1] else n + S[i, size]
  n = if (j < size) n + S[i, j+1] else n + S[i, 1]
  energy = -(J * S[i, j] * n) - (H * S[i, j])
  if (T != 0 && (energy > 0 || runif(1) < (exp(2 * energy / T)))) {
    S[i,j] = (-1 * S[i,j])
  }
  return(S)
}
```

Tato funkce jednoduše opakuje výše popsaný postup pro `n` kroků.

```
run.simulation = function(S, T, J = 1, H = 0, n) {
  for (i in 1:n) {
    S = single.step(S, T, J, H)
  }
  return(S)
}
```

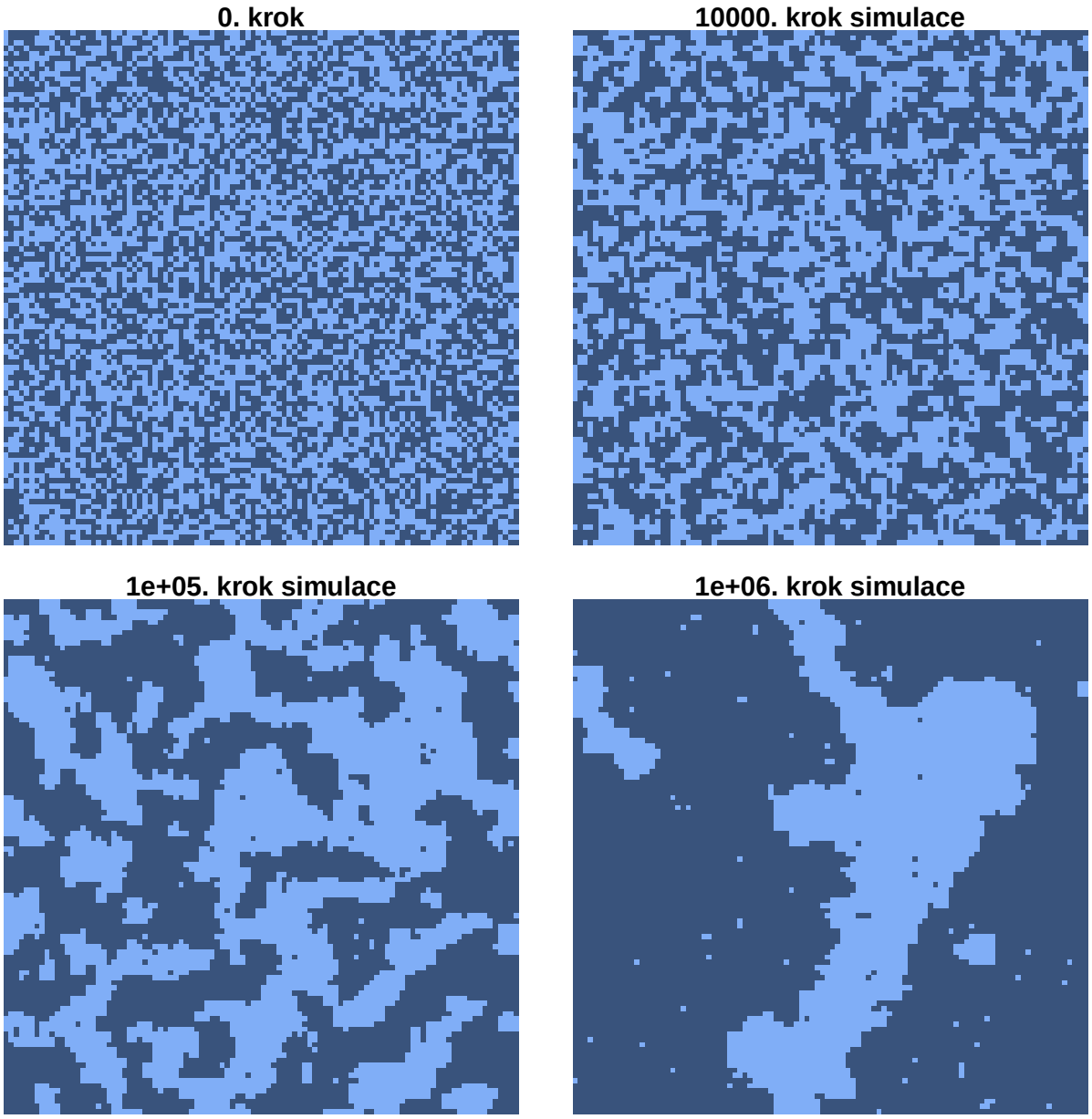
Vykreslovací funkce nastaví parametry grafického zařízení (absenci os apod.) a následně zavolá vestavěnou metodu `image`, která konvertuje numerickou matici na bitmapu.

```
display = function(S, col1 = "#80aef7", col2 = "#39537c") {
  par(
    fg = "transparent",
    col.axis = "transparent",
    col.lab = "transparent",
    mar = c(1, 1, 1, 1)
  )
  image(S, axes = TRUE, col = c(col1, col2))
}
```

9.1. knitr

Nyní můžeme použít výše nadefinované funkce pro spuštění simulace. Nejprve si vytvoříme pole náhodných spinů. Nad tímto polem budeme provádět simulaci a vykreslíme stav pole funkcí `display` po 10 000, 100 000 a 1 000 000 kroků.

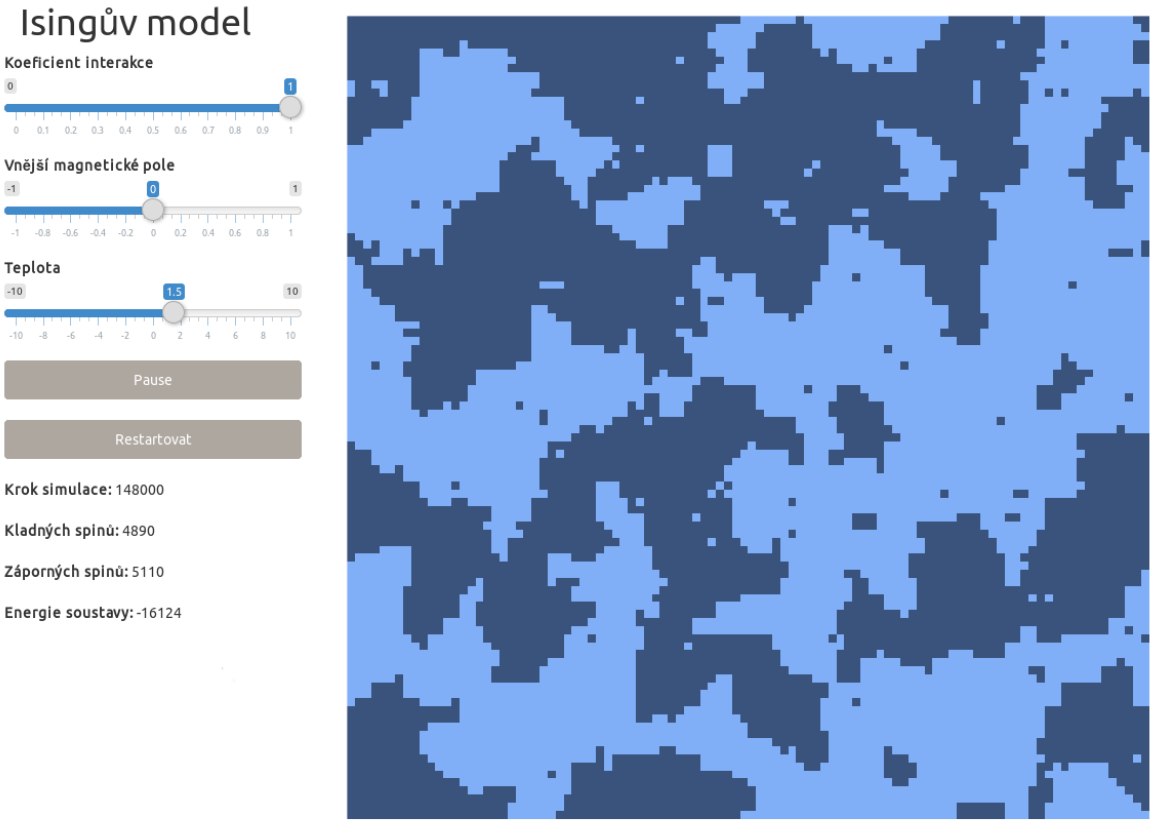
```
temperature = 1.5
size = 100
S = spin.arr(size)
par(mfrow=c(2,2))

display(S)
title("0. krok")

step = 10000
S = run.simulation(S, temperature, n = step)
display(S)
title(paste(step, ". krok simulace", sep = ""))

step = 100000
S = run.simulation(S, temperature, n = step)
display(S)
title(paste(step, ". krok simulace", sep = ""))

step = 1000000
S = run.simulation(S, temperature, n = step)
display(S)
title(paste(step, ". krok simulace", sep = ""))
```



9.2. Shiny

V *Shiny* aplikaci využijeme kromě backendu také knihovny `shiny`, `shinythemes` a `shinyjs`. Balíček `shinyjs` nám umožňuje volat některé běžné javascriptové funkce přímo z R, bez potřeby psát vlastní javascriptový kód. Jedná se například o funkce pro skrytí / zobrazení či aktivaci / deaktivaci různých prvků stránky.

```
library(shiny)
library(shinyjs)
library(shinythemes)
source("./ising.R")
```



Obrázek 9.1.: Isingův model – Shiny aplikace

Nadefinujeme si konstanty: interval, ve kterém bude docházet k prekreslování grafu a údajů o stavu systému (v milisekundách), a rozměr mřížky.

```
renderingInterval = 10
latticeSize = 100
```

Metoda `server` implementuje logiku *Shiny* aplikace. Nadefinujeme si reaktivní proměnnou `run`, vlajku indikující, zda právě běží simulace. Vygenerujeme pole spinů nazvané `S` a nastavíme hodnotu proměnné `step` na nulu.

```
server = function(input, output, session) {
  values = reactiveValues()
  values$run = FALSE

  S = spin.arr(latticeSize)
  step = 0
```

Následuje hlavní funkce v aplikaci. Tato funkce se vykonává stále dokola, pokud je nastaven příznak `run`. V každém zavolání této funkce je provedeno tisíc kroků simulace a stav systému je uložen zpět do globální proměnné `S`. Je také zvýšen `step` o odpovídající hodnotu. Příkaz `invalidateLater(0)`, volaný na konci této funkce, ji opět zařadí do fronty na vyhodnocení. Jeho parametrem je číslo indikující kolik milisekund se má čekat, než bude tato funkce znovu vyhodnocena.

```
  observe({
    if (values$run) {
      for (i in 1:1000) {
        S <<- single.step(S, input$T, input$J, input$H)
        step <<- step + 1
      }
      invalidateLater(0)
    }
  })
```

Následují obslužné rutiny pro vykreslení grafu a vypsaní informací o stavu systému uživateli. Všechny sledují stejný vzor: pokud je nastaven příznak spuštěné simulace, zařadí se do fronty pro znovuvyhodnocení, se zpožděním daným konstantou určující interval vykreslování. Dále funkce pro vykreslení grafu pouze volá odpovídající metodu backendu. Funkce pro vykreslení textu musejí vrátit vektor znaků, proto nejprve provedou zařazení do fronty pro znovuvyhodnocení, pak teprve vypočtou a vrátí požadovanou hodnotu – počet uplynulých kroků simulace, počet kladných/záporných spinů v systému a celkovou energii systému.

```
  output$plot = renderPlot({
    display(S)
    if (values$run) {
      invalidateLater(renderingInterval)
```

```
  }
})

output$step = renderText({
  if (values$run) {
    invalidateLater(renderingInterval)
  }
  return(step)
})

output$pos.spins = renderText({
  if (values$run) {
    invalidateLater(renderingInterval)
  }
  return(sum(S > 0))
})

output$neg.spins = renderText({
  if (values$run) {
    invalidateLater(renderingInterval)
  }
  return(sum(S < 0))
})

output$total.energy = renderText({
  if (values$run) {
    invalidateLater(renderingInterval)
  }
  return(total.energy(
    S, input$J, input$H))
})
```

Následující řada event handlerů obsluhuje trojici `actionButton` elementů. Jsou v nich využívány funkce implementované knihovnou `shinyjs` pro skrytí a zobrazení elementů, kromě toho první dvě funkce nedělají nic jiného, než je nastavení odpovídající hodnoty reaktivní proměnné `run`. Poslední obslužná rutina resetuje stav systému, tedy volá funkci backendu pro vygenerování nového pole spinů a nastaví hodnotu `step` na nulu.

```
observeEvent(input$run, {
  shinyjs::hide("run")
  shinyjs::show("stop")
  values$run = TRUE
})

observeEvent(input$stop, {
  shinyjs::hide("stop")
```

```
    shinyjs::show("run")
    values$run = FALSE
  })

  observeEvent(input$restart, {
    shinyjs::hide("stop")
    shinyjs::show("run")
    S <<- spin.arr(latticeSize)
    step <<- 0
    values$run = FALSE
  })
}
```

V objektu `ui` je opět uložena definice uživatelského rozhraní. Je zde nastaveno CSS za použití knihovny `shinythemes` a volání funkce `useShinyJs` nezbytné pro použití funkcí z této knihovny. Dále je zde použit HTML tag `style` (skrze objekt `tags` poskytovaný knihovnou `shiny`). Výchozím chováním *Shiny* frameworku je vyblednutí grafu pokud dochází k přepočítání hodnot, na kterých je závislý. Abychom potlačili toto chování, je nastavena hodnota `opacity` pro CSS třídu `recalculating` na 1.

Zbytek je standardní *Shiny* UI definované funkcemi poskytovanými touto knihovnou. Obrazovka je opět rozdělena do dvou sloupců v poměru 1:3. Levý sloupec obsahuje nadpis a `sliderInput` elementy pro nastavení koeficientů ovlivňujících běh simulace (tyto hodnoty lze měnit při spuštěné simulaci a na živo sledovat jejich vliv). Dále je zde trojice tlačítek pro spuštění / pozastavení / restartování simulace, a nakonec prostor pro vykreslení textových výstupů spolu s jejich popisky.

```
ui = fluidPage(
  theme = shinytheme("united"), useShinyjs(),
  tags$style(type = "text/css",
    ".recalculating {opacity: 1.0;}"
  ),

  fluidRow(
    column(3,
      headerPanel("Isingův model"),
      br(),
      sliderInput("J", label = "Koeficient interakce",
        value = 1, step = 0.1, min = 0, max = 1, width = "100%"),
      sliderInput("H", label = "Vnější magnetické pole",
        value = 0, step = 0.1, min = -1, max = 1, width = "100%"),
      sliderInput("T", label = "Teplota",
        value = 1.5, step = 0.1, min = -10, max = 10, width = "100%"),
      actionButton("run", label = "Start", width = "100%"),
      hidden(
        actionButton("stop", label = "Pause", width = "100%")
      ),
```

```
      br(), br(),
      actionButton("restart", label = "Restartovat", width = "100%"),
      br(), br(),
      tags$b("Krok simulace: "),
      textOutput("step", inline = TRUE),
      br(), br(),
      tags$b("Kladných spinů: "),
      textOutput("pos.spins", inline = TRUE),
      br(), br(),
      tags$b("Záporných spinů: "),
      textOutput("neg.spins", inline = TRUE),
      br(), br(),
      tags$b("Energie soustavy: "),
      textOutput("total.energy", inline = TRUE)
    ),
    column(9, br(), plotOutput("plot", width="90vh", height="90vh"))
  )
)
```

Aplikaci spustíme běžným způsobem:

```
runApp(appDir = shinyApp(ui, server), port = 8001, host = "127.0.0.1")
```

10. Molekulární dynamika systému tuhých disků

Pro vytvoření kružnice v grafu budeme potřebovat importovat balíček `plotrix`.

```
library(plotrix)
```

Naše simulace bude využívat událostmi řízený model. To znamená, že při spuštění simulace vypočteme nadcházející kolize pro všechny částice a po zbytek budeme dopočítávat pouze nové kolize pro právě kolidující částice – viz teoretická část práce. Stav systému v simulaci bude z velké části definován dvěma objekty – čtyřsloupcovou maticí `M`, obsahující na každém řádku informace o poloze a rychlosti každé částice, a dataframem `collisions`, obsahujícím záznamy o nadcházejících kolizích.

Funkce `init` slouží k inicializaci systému. Vytváří matici `M` pro zadaný počet `n` částic, přičemž každá částice má poloměr `size` a rychlost náhodně zvolenou z intervalu `min.speed` až `max.speed`. Částice jsou náhodně generovány na prostoru 0 až 1, ovšem vyvstává zde komplikace v tom, že mají nenulovou velikost a chceme, aby se nepřekrývali. Po vygenerování nové částice je spočtena její vzdálenost od všech ostatních, a pokud je vzdálenost od libovolné částice menší než `2.5*size`, není nově přidaná částice vložena do systému.

```
init = function(n, min.speed, max.speed, size) {
  x = runif(n, 0+size, 1-size)
  y = runif(n, 0+size, 1-size)
  for (i in 1:(n-1)) {
    if (is.na(x[i]) || is.na(y[i])) { next }
    for (j in (i+1):n) {
      if (i == j || is.na(x[j]) || is.na(y[j])) { next }
      a = dis(x[i], y[i], x[j], y[j])
      if (a < 2.5*size) {
        x[j] = y[j] = NA
      }
    }
  }
  x = x[!is.na(x)]
  y = y[!is.na(y)]
  n = length(x)
  vx = runif(n, min.speed, max.speed)
  vy = runif(n, min.speed, max.speed)
  m = matrix(ncol = 4, nrow = n)
```

```
  m[,1] = x
  m[,2] = y
  m[,3] = vx
  m[,4] = vy
  return(m)
}
```

Metoda `find.nearest.collision` přijímá matici `M` popisující stav systému, velikost částice `size` a index do matice `M` nazvaný `i`. Účelem této funkce je vypočíst čas do nejbližší kolize (ať už se stěnou či jinou částicí) pro částici definovanou indexem `i`. V případě, že se jedná o kolizi s jinou částicí, je zaznamenán také index do `M` definující tuto druhou částici. K nalezení kolizí slouží metody `wall` a `spheres`, popsané níže. Výstupem je seznam svým tvarem odpovídající jednomu řádku dataframeu `collisions`.

```
find.nearest.collision = function(M, size, i) {
  ttc = wall(M[i, 1], M[i, 3], size)
  col = NULL
  if (!is.na(ttc)) {
    col = list(time = ttc, primary = i, secondary = -1, valid = TRUE)
  }
  ttc = wall(M[i, 2], M[i, 4], size)
  if (is.null(col) || (!is.na(ttc) && ttc < col$time)) {
    col = list(time = ttc, primary = i, secondary = -1, valid = TRUE)
  }
  for (j in 1:nrow(M)) {
    if (j != i) {
      ttc = spheres(
        M[i, 1], M[i, 2], M[i, 3], M[i, 4], size,
        M[j, 1], M[j, 2], M[j, 3], M[j, 4], size)
      if (!is.na(ttc) && ttc < col$time) {
        col = list(time = ttc, primary = i, secondary = j, valid = TRUE)
      }
    }
  }
  return(col)
}
```

`get.collisions` je funkce odpovědná za počáteční vytvoření dataframeu `collisions`. Jak jsme viděli již v předchozím odstavci, jeden řádek této tabulky (popisující právě jednu kolizi se stěnou či jinou částicí) obsahuje čtyři sloupce: `time`, udávající čas (v jednotkách $\mathrm{d}t$) do kolize; `primary`, index částice, které se tato kolize týká (tento index má vždy platnou hodnotu), `secondary`, což je index druhé částice v kolizi, či hodnotu -1, pokud jde o kolizi se stěnou; nakonec příznak `valid`, nezbytný, neboť v průběhu času budou některé z původně vypočtených kolizí „invalidovány“.

```
get.collisions = function(M, size) {
  collisions = data.frame(time = NA,
    primary = NA, secondary = NA, valid = NA)

  for (i in 1:nrow(M)) {
    collisions[nrow(collisions) + 1,] =
      find.nearest.collision(M, size, i)
  }

  collisions = collisions[!is.na(collisions$time),]
  collisions[order(collisions$time),]
}
```

Tato funkce počítá čas do kolize částice se zdí, k čemuž do ní vstupují parametry `r`, značící pozici (jednu souřadnici, buď x nebo y) částice, `v` značící rychlost, a `o`, poloměr částice.

```
wall = function(r, v, o) {
  if (v > 0) return((1 - o - r) / v)
  if (v < 0) return((o - r) / v)
  if (v == 0) return(NA)
}
```

Metoda `spheres` počítá čas do kolize dvou těles. Parametrů je hodně, ale jejich význam je zřejmý: poloha (x a y) a rychlost každé částice, stejně jako jejich poloměr. Opět se jedná o přesný přepis algoritmů, uvedených v teoretické části.

```
spheres = function(rxi, ryi, vxi, vyi, oi, rxj, ryj, vxj, vyj, oj) {
  o = oi+oj
  dx = rxj - rxi
  dy = ryj - ryi
  dvx = vxj - vxi
  dvy = vyj - vyi
  drdr = dx^2 + dy^2
  dvdr = dvx*dx + dvy*dy
  dvdv = dvx^2 + dvy^2
  d = dvdr^2 - (dvdv * (drdr - o^2))
  dt = if (dvdr >= 0) {
    NA
  } else if (d < 0) {
    NA
  } else {
    -1 * (dvdr + sqrt(d)) / (dvdv)
  }
  return (dt)
}
```

`dis` slouží ke spočtení Eukleidovské vzdálenosti.

```
dis = function(x1, y1, x2, y2) {
  sqrt( ((x1-x2)**2) + ((y1-y2)**2) )
}
```

Metoda `impulse` počítá nové rychlosti dvojice částic po jejich kolizi. Vstupní argumenty jsou opět pozice a rychlost každé z částic. Výstupem je seznam obsahující čtyři prvky, x a y složku nové rychlosti pro každou z částic.

```
impulse = function(xi, yi, vxi, vyi, xj, yj, vxj, vyj) {
  x1 = c(xi, yi)
  v1 = c(vxi, vyi)
  x2 = c(xj, yj)
  v2 = c(vxj, vyj)
  nv1 = v1 - (x1 - x2) *
    (as.vector((v1 - v2) %*% (x1 - x2)) / norm(x1 - x2, t="2")^2)
  nv2 = v2 - (x2 - x1) *
    (as.vector((v2 - v1) %*% (x2 - x1)) / norm(x2 - x1, t="2")^2)
  return(list(vxi = nv1[1], vyi = nv1[2], vxj = nv2[1], vyj = nv2[2]))
}
```

Toto je nejkomplikovanější metoda simulace, implementující jeden krok. Realizuje čtveřici úkolů: nejprve sníží zbývající čas do kolize o jednotku dt; následuje rozřešení eminentních ($<$ dt) kolizí; posunutí všech částic o odpovídající vzdálenost danou jejich rychlostí a délkou kroku dt; nakonec je určena nová hodnota dt. Tři z těchto úkolů jsou relativně triviální, ovšem řešení aktuálních kolizí je relativně komplikovaný problém.

Tento postup budeme opakovat pro všechny kolize, do kterých zbývá méně času, než je délka jednoho kroku simulace. Nejprve si do proměnných `p` a `s` uložíme indexy (do matice `M`) částic, kterých se kolize týká. Pokud jde o kolizi částice `p` se stěnou (ať už horizontální či vertikální), hodnota `s` bude -1. Nyní nastavíme příznak neplatnosti pro tuto kolizi a všechny nadcházející kolize, kterých se účastní částice `p`. Následuje *if-else* konstrukt, který rozlišuje, zda se jedná o kolizi se stěnou (`s < 1`) nebo s druhou částicí. Pokud jde o kolizi se stěnou, zjistíme, zda jde o svislou či horizontální stěnu, a obrátíme odpovídající složku rychlosti částice `p`. Pokud `s` nemá hodnotu -1, šlo o kolizi s jinou částicí. V tomto případě musíme invalidovat všechny nadcházející kolize, zahrnující na primární či sekundární pozici částici `s`. Poté metodou `impulse` vypočteme nové hodnoty rychlostí pro obě částice a přiřadíme jim je. V této větvi také vypočteme nejbližší následující kolizi pro částici `s`. Výpočet nadcházející kolize pro `p` musíme provést v obou případech, proto se nachází až za *if-else* blokem. Ještě musíme celý seznam nadcházejících kolizí setřídit podle sloupce `time` (což je paradoxně rychlejší operace, než vložení nového záznamu na správnou pozici).

Zbývá nám změnit pozici všech částic, což provedeme přičtením dt-násobku třetího a čtvrtého sloupce matice `M` k prvnímu a druhému sloupci téže matice, respektive. Určení nové délky kroku simulace je dáno časem do nejbližší nadcházející kolize: pokud je menší, než je výchozí hodnota kroku, nastavíme dt rovno času do kolize, v opačném případě použijeme výchozí hodnotu. Zbývá vrátit matici `M`, seznam kolizí a novou hodnotu délky kroku.

```
single.step = function(M, collisions, size, default.dt) {

  dt = if ((collisions[collisions$valid,][1,]$time) < default.dt) {
    collisions[collisions$valid,][1,]$time
  } else {
    default.dt
  }

  collisions$time = collisions$time - dt

  for (i in 1:nrow(M)) {
    M[i, 1] = M[i, 1] + dt * M[i, 3]
    M[i, 2] = M[i, 2] + dt * M[i, 4]
  }

  if (collisions[collisions$valid,][1,]$time < dt) {
    p = collisions[collisions$valid,][1,]$primary
    s = collisions[collisions$valid,][1,]$secondary
    collisions[collisions$valid,][1,]$valid = FALSE
    if (nrow(collisions[collisions$primary == p,]) > 0) {
      collisions[collisions$primary == p,]$valid = FALSE
    }
    if (nrow(collisions[collisions$secondary == p,]) > 0) {
      collisions[collisions$secondary == p,]$valid = FALSE
    }
    if (s < 1) {
      dt.ver = wall(M[p, 1], M[p, 3], size)
      dt.hor = wall(M[p, 2], M[p, 4], size)

      if (!is.na(dt.ver)) {
        if (!is.na(dt.hor)) {
          if (dt.hor < dt.ver) {
            M[p, 4] = -1 * M[p, 4]
          } else {
            M[p, 3] = -1 * M[p, 3]
          }
        } else {
          M[p, 3] = -1 * M[p, 3]
        }
      } else if (!is.na(dt.hor)) {
        M[p, 4] = -1 * M[p, 4]
      }
    } else {
      collisions[collisions$primary == s,]$valid = FALSE
      collisions[collisions$secondary == s,]$valid = FALSE
      impulse = impulse(
```

```
        M[p, 1], M[p, 2], M[p, 3], M[p, 4],
        M[s, 1], M[s, 2], M[s, 3], M[s, 4])
      M[p, 3] = impulse$vxi
      M[p, 4] = impulse$vyi
      M[s, 3] = impulse$vxj
      M[s, 4] = impulse$vyj
      collisions[nrow(collisions) + 1,] =
        find.nearest.collision(M, size, s)
    }

    collisions[nrow(collisions) + 1,] =
      find.nearest.collision(M, size, p)

    collisions = collisions[order(collisions$time),]
  }

  return(list(M = M, cols = collisions, dt = dt))
}
```

Zbývá implementovat dvojici grafických funkcí. První z nich bude vykreslovat polohu všech částic v systému. Základní princip je jednoduchý – budeme v cyklu procházet matici `M` a pro každou částici zobrazíme kružnici na odpovídajících souřadnicích. Tento postup pouze mírně zkomplikujeme tím, že částice může mít tři různé barvy: ve výchozím stavu je bílá, při kolizi (těsně před a po) se stěnou je žlutá, při kolizi s druhou částicí jsou obě červené. Nejprve nadefinujeme grafické parametry a vytvoříme prázdný graf. Pro každou částici v systému provedeme následující: nastavíme částici žlutou barvu, pokud se její střed nachází méně než 1.2 poloměru od kterékoliv ze stěn, jinak dáme částici barvu bílou. Poté procházíme všechny ostatní částice, pokud se nějaká nachází méně než 2.4 poloměru daleko, obarvíme aktuální částici červeně. Vykreslení částice je realizováno funkcí `draw.circle` z výše importovaného balíčku `plotrix`.

```
display = function(M, size) {
  par(
    fg = "transparent",
    col.axis = "transparent",
    col.lab = "transparent",
    bty = "n",
    mai = c(0, 0, 0, 0),
    mar = c(0, 0, 0, 0)
  )

  plot(c(0,1), c(0,1), t = "n")

  lines(c(0,0,1,1,0),c(0,1,1,0,0), col = "black")
  for (i in 1:nrow(M)) {
```

```
    col = if (
      M[i,1] < 1.2*size ||
      M[i,1] > 1-(1.2*size) ||
      M[i,2] < 1.2*size ||
      M[i,2] > 1-(1.2*size))
    "yellow" else "white"

    for (j in 1:nrow(M)) {
      if (i != j) {
        if (dis(M[i,1], M[i,2], M[j,1], M[j,2]) < 2.4*size) {
          col = "red"
        }
      }
    }

    draw.circle(M[i,1], M[i,2], size,
      nv = 1000, border = "black", col = col, lty = 1, lwd = 1)
  }
}
```

Druhou grafickou funkcí je histogram, který použijeme pro znázornění rozdělení různých složek rychlosti všech částic. Histogram vykreslíme funkcí `hist`, následně přes něj zakreslíme křivku hustoty pravděpodobnosti funkcí `density`. Přitom funkce `hist` vytvoří nový graf, abychom ho nepřekreslili křivkou hustoty, nejprve výstup funkce `density` uložíme do proměnné a tu pak zobrazíme funkcí `polygon`.

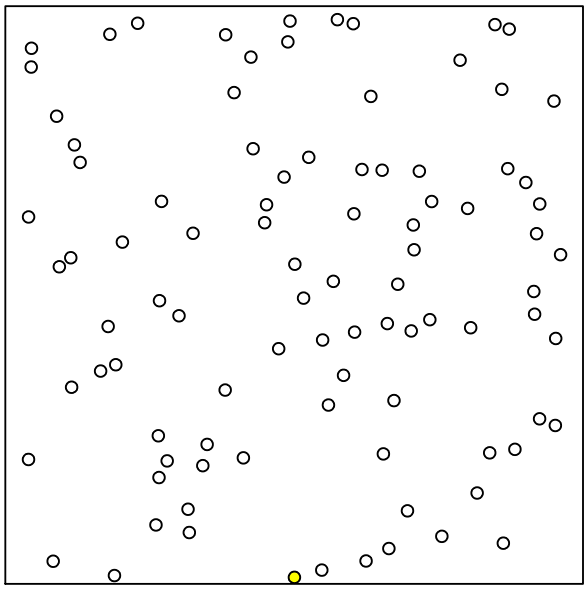
```
histogram = function(vec) {
  par(col.main = "transparent", fg = "black", col.lab = "transparent", mai = c(1, 1, 1, 1))
  hist(vec, breaks = 20, freq = FALSE, density = 20)
  d = density(vec)
  polygon(d)
}
```

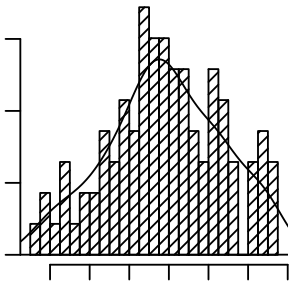
10.1. knitr

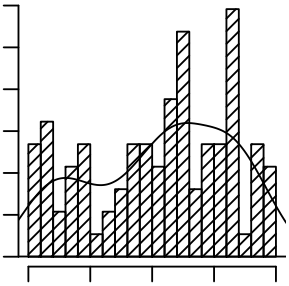
Ukázka výše popsaných grafů: vizualizace stavu systému a trojice histogramů. Histogramy znázorňují rozdělení složek rychlosti x a y, stejně jako rozdělení celkové rychlosti.

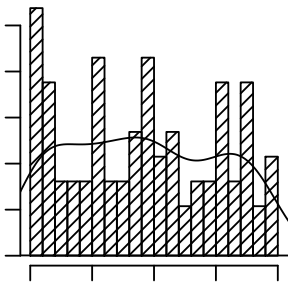
```
n = 100 # počet částic
min.speed = -10
max.speed = 10
size = 0.01
M = init(n, min.speed, max.speed, size)
dt = 0.0003
```

```
collisions = get.collisions(M, size)
par(mfrow = c(2,2))
display(M, size)
histogram(sqrt(M[,3]^2 + M[,4]^2))
histogram(M[,3])
histogram(M[,4])
```

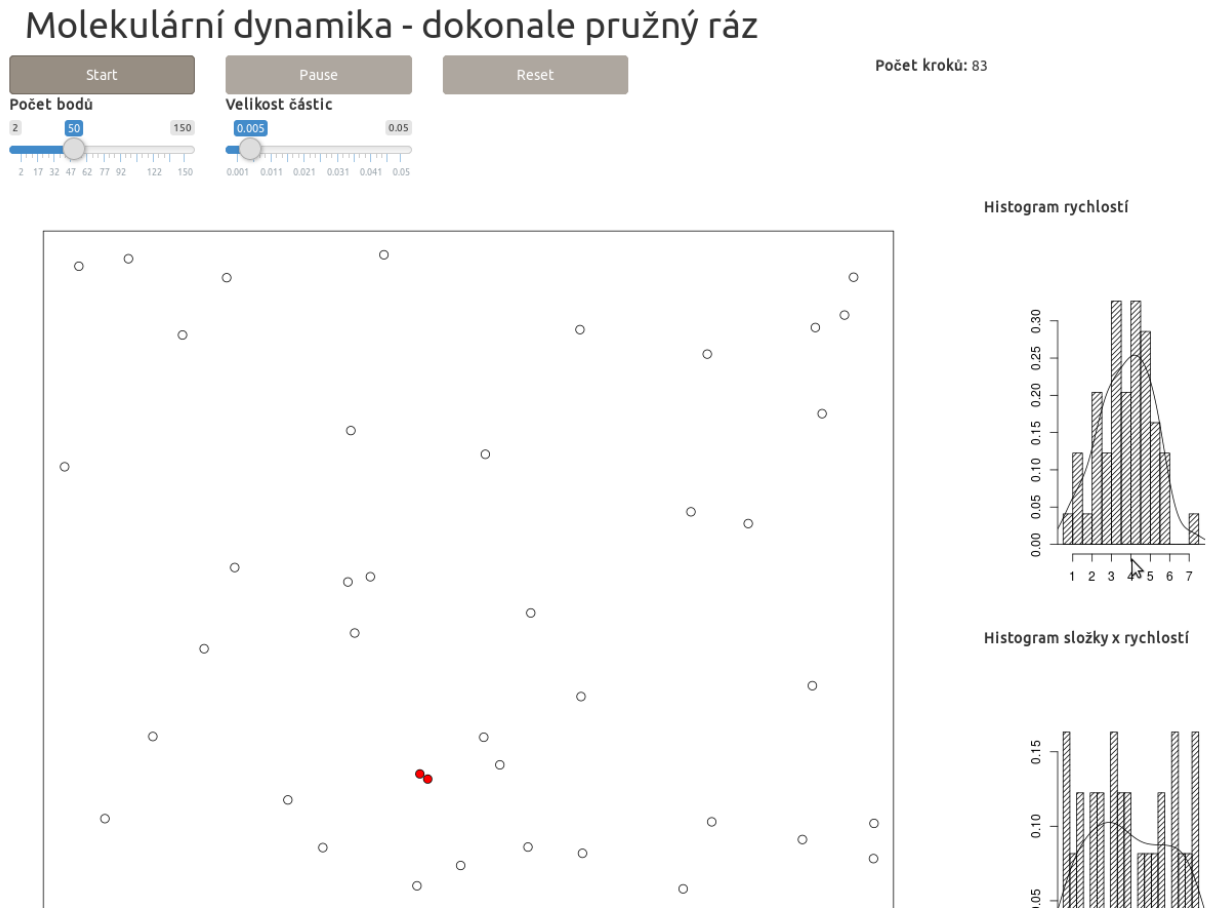








10.2. Shiny



Obrázek 10.1.: Molekulární dynamika systému tuhých disků - Shiny aplikace

Běžný import knihoven `shiny` a `shinythemes` spolu s backendem.

```
library(tictoc)
library(shiny)
library(shinythemes)
source("./mol.R")
```

V globálním oboru platnosti nadefinujeme několik konstant. Jsou jimi čítač kroků simulace, příznak spuštěné simulace, celkový počet částic v systému, rozsah rychlostí, poloměr částic (stejný pro všechny), výchozí a aktuální délka jednoho kroku simulace, a interval pro překreslování grafů (v milisekundách). Dále na základě těchto hodnot vytvoříme matici `M`.

```
step = 0
run = FALSE
point.count = 50
min.speed = -5
max.speed = 5
```

```
size = 0.005
default.dt = 0.001
dt = default.dt
renderT = 100
```

Tento reaktivní výraz implementuje jeden krok simulace. Pokud je nastaven příznak značící, že je simulace spuštěna, zvýší čítač kroků o jedna, zavolá metodu backendu realizující jeden krok simulace, hodnoty vrácené touto metodou uloží v globálním kontextu, a nastaví sám sebe pro okamžité znovuvyhodnocení (`invalidateLater(0)`).

```
server = function(input, output, session) {
  M = init(point.count, min.speed, max.speed, size)

  observe({
    if (run) {
      step <<- step + 1
      l = single.step(M, collisions, size, default.dt)
      M <<- l$M
      collisions <<- l$cols
      dt <<- l$dt
    }
    invalidateLater(0)
  })
```

Následující trojice observerů realizuje výstupy simulace, jedná se o vizualizaci stavu systému, trojici histogramů (opět znázorňující rozdělení rychlosti a x a y složek rychlosti) a aktuální počet kroků uskutečněných v simulaci. Všechny tyto výstupní metody se překreslují v intervalu daném hodnotou `renderT`.

```
  output$plot = renderPlot({
    display(M, size)
    invalidateLater(renderT)
  })

  observe({
    output$hist.total = renderPlot(
      histogram(sqrt(M[,3]^2 + M[,4]^2)))
    output$hist.x = renderPlot(histogram(M[,3]))
    output$hist.y = renderPlot(histogram(M[,4]))
    invalidateLater(renderT)
  })

  output$step = renderText({
    invalidateLater(renderT)
    return(step)
  })
```

Poslední částí serverové funkce je obsluha trojice tlačítek *start*, *pause* a *reset*. Tlačítko pro spuštění simulace zavolá funkci backendu pro výpočet všech nadcházejících kolizí, výstup této funkce opět uloží v globálním kontextu, načež nastaví příznak běžící simulace na `TRUE`. Tlačítko pro pozastavení pouze nastaví hodnotu příznaku na nepravdivou. Při restartu simulace se resetuje čítač kroků a inicializuje nový systém.

```
  observeEvent(input$start, {
    if (step == 0) collisions <<- get.collisions(M, size)
    run <<- TRUE
  })

  observeEvent(input$pause, {
    run <<- FALSE
  })

  observeEvent(input$reset, {
    size <<- input$size
    point.count <<- input$points
    run <<- FALSE
    step <<- 0
    M <<- init(point.count, min.speed, max.speed, size)
  })
}
```

Uživatelské rozhraní je relativně jednoduché, tvořeno trojicí tlačítek, textovým výstupem, a čtyřmi grafy. Definujeme si konstantu pro velikost (čtvercových) bočních grafů (zobrazujících histogramy), nastavíme CSS styl a odstraníme vyblednutí grafu při přepočítávání.

```
ui = fluidPage(
  theme = shinytheme("united"),
  tags$style(type = "text/css",
    ".recalculating {opacity: 1.0;}"
  ),
```

Stránka je rozdělena na tři řádky. Pod nadpisem následuje řádek obsahující trojici tlačítek a textový výstup pro zobrazení počtu kroků. Na dalším řádku se nachází dvojice *sliderInput*ů pro nastavení počtu a velikosti částic.

```
  headerPanel("Molekulární dynamika - dokonale pružný ráz"),
  fluidRow(
    column(2,
      actionButton("start", label = "Start", width = "100%")),
    column(2,
      actionButton("pause", label = "Pause", width = "100%")),
    column(2,
      actionButton("reset", label = "Reset", width = "100%")),
```

```
    column(2),
    column(2,
      tags$b("Počet kroků: "),
      textOutput("step", inline = TRUE))
  ),
  fluidRow(
    column(2, sliderInput("points", label = "Počet bodů",
      min = 2, max = 150, value = point.count)),
    column(2, sliderInput("size", label = "Velikost částic",
      min = 0.001, max = 0.05, value = size))
  ),
```

Druhý řádek je rozdělen na dva sloupce v poměru 1:2. V hlavním sloupci nalevo se nachází primární graf, zobrazující polohu jednotlivých částic v systému. V menším sloupci napravo je umístěna trojice histogramů.

```
  fluidRow(
    column(9,
      plotOutput("plot", width = "70vw", height = "70vw")
    ),
    column(3,
      verticalLayout(
        tags$b("Histogram rychlostí"),
        plotOutput("hist.total", width = "100%"),
        tags$b("Histogram složky x rychlostí"),
        plotOutput("hist.x", width = "100%"),
        tags$b("Histogram složky y rychlostí"),
        plotOutput("hist.y", width = "100%")
      )
    )
  )
)
```

Spuštění aplikace:

```
runApp(appDir = shinyApp(ui, server), port = 8010, host = "127.0.0.1")
```

Část III.

Závěr

Cílem této práce bylo ukázat možnosti nástrojů *knitr* a *Shiny* pro vizualizaci a reprodukovatelný výzkum. Bylo implementováno několik ukázkových úloh z rozličných oblastí, na kterých jsem ukázal použití těchto knihoven.

Knitr považuji za unikátní způsob pro tvorbu dokumentů, obzvláště takových, kde buď přímo popisujeme nějaký zdrojový kód, nebo z velké míry využíváme výstupů generovaných takovým kódem. Možnost kód – v řadě různých jazyků – přímo vkládat do textu a volat knihovny, skripty a programy, přičemž jejich výstupy jsou automaticky vloženy do výstupního dokumentu, nejenže výrazně usnadňuje práci, ale současně umožňuje mít přehlednější, uspořádanější vývojové prostředí. Za největší sílu *knitru* považuji jeho kombinaci s texem při popisu a implementaci matematických algoritmů – můžeme tak snadno v jednom dokumentu mít rovnice popisující takové algoritmy, jejich programovou implementaci a současně i výstupy v podobě výpočtů, grafů a tabulek. Příkladem takového dokumentu je přímo tato práce, jejíž zdrojový kód je k dispozici v příloze.

Framework na tvorbu webových aplikací *Shiny* mě překvapil svou hloubkou. Je snadné nechat se zmást tím, jak snadné je v *Shiny* vytvořit jednoduchou aplikaci, a myslet si, že je to vše, co umí. Ovšem, jak bylo snad dostatečně ukázáno v této práci, možnosti a komplexita aplikací tvořených pomocí této knihovny nejsou nijak zásadně omezeny. Přitom rozšiřování takové aplikace může být libovolně pozvolné, od využívání základních funkcí samotné knihovny, přes rozšiřující balíčky, až po psaní vlastního HTML, CSS a JavaScriptového kódu.

Paradigma reaktivního programování může být pro leckoho matoucí, a občas je potřeba nemálo experimentovat se zdrojovým kódem, než člověk pochopí, jaké vztahy existují mezi jednotlivými komponentami a jak se aplikace jako celek chová. Nicméně nejedná se o zbytečnou komplexitu, reaktivní programování odstraňuje velké množství „boilerplate“ kódu, sloužícího jen k repetitivnímu definování vzájemných vztahů mezi jednotlivými funkcemi a proměnnými – místo toho se autor může soustředit na psaní kódu, na kterém skutečně záleží.

Za největší slabinu obou nástrojů považuji samotný programovací jazyk R. R shledávám na jednu stranu zbytečně komplikovaným, na druhou stranu často postrádá elementární prvky. U *knitru* tato nevýhoda není tak silná vzhledem k možnosti kombinovat jej s rozličnou škálou jazyků, ačkoli R jakožto nativní jazyk *knitru* má v této oblasti určité výhody. V případě *Shiny* ovšem není úniku, pokud chceme použít tento framework, znamená to použít R. Navzdory tomu je *Shiny* velice příjemným způsobem pro vytváření (nejen) jednoduchých webových aplikací.

Literatura

[1] The R Foundation *What is R?* [online]. [cit. 2018-01-22]. Dostupné z: https://www.r-project.org/about.html

[2] *CRAN - Contributed Packages* [online]. [cit. 2018-01-22]. Dostupné z: https://cran.r-project.org/web/packages/index.html

[3] CHAMBERS, John. *Evolution of S* [online]. 2000 [cit. 2018-01-22]. Dostupné z: http://ect.bell-labs.com/sl/S/history.html

[4] IHAKA, Ross. *The R Project: A Brief History and Thoughts About the Future* [online]. 2009 [cit. 2018-01-22]. Dostupné z: https://www.stat.auckland.ac.nz/~ihaka/downloads/Massey.pdf

[5] LANDER, Jared P. *R for Everyone: Advanced Analytics and Graphics.* 978-0-321-88803-7. Upper Saddle River, New Jersey: Addison-Wesley, 2014, s. 21-36.

[6] *How To Install R on Ubuntu 16.04* [online]. [cit. 2018-01-22]. Dostupné z: https://www.r-bloggers.com/how-to-install-r-on-linux-ubuntu-16-04-xenial-xerus/

[7] *RStudio* [online]. RStudio, 2011 - 2017 [cit. 2018-01-22]. Dostupné z: https://www.rstudio.com/products/RStudio/

[8] *R Development Tools* [online]. Microsoft, 2018 [cit. 2018-01-22]. Dostupné z: https://www.visualstudio.com/vs/rtvs/

[9] *R Language Support* [online]. JetBrains, 2018 [cit. 2018-01-22]. Dostupné z: https://plugins.jetbrains.com/plugin/6632-r-language-support

[10] *StatET for R* [online]. WalWare, 2016 [cit. 2018-01-22]. Dostupné z: http://www.walware.de/goto/statet

[11] *R package for Sublime Text.* GitHub, Inc. [online]. 2018 [cit. 2018-01-22]. Dostupné z: https://github.com/randy3k/R-Box

[12] WICKHAM, Hadley. *Functional programming* [online]. 2018 [cit. 2018-01-22]. Dostupné z: http://adv-r.had.co.nz/Functional-programming.html

[13] WHITE, John Myles. *Higher Order Functions in R* [online]. [cit. 2018-05-01]. Dostupné z: http://www.johnmyleswhite.com/notebook/2010/09/23/higher-order-functions-in-r/

[14] XIE, Yihui. *Knitr: Elegant, flexible, and fast dynamic report generation with R* [online]. 2018, 2005 - 2018 [cit. 2018-04-22]. Dostupné z: https://yihui.name/knitr/

[15] *Reactivity - An overview* [online]. RStudio, 2017 [cit. 2018-04-22]. Dostupné z: https://shiny.rstudio.com/articles/reactivity-overview.html

[16] *Introduction to K-means Clustering* [online]. [cit. 2018-04-28]. Dostupné z: https://www.datascience.com/blog/k-means-clustering

[17] *Convergence in Hartigan-Wong k-means method and other algorithms* [online]. [cit. 2018-04-28]. Dostupné z: https://datascience.stackexchange.com/questions/9858/convergence-in-hartigan-wong-k-means-method-and-other-algorithms

[18] *K-Means: Lloyd,Forgy,MacQueen,Hartigan-Wong* [online]. [cit. 2018-04-28]. Dostupné z: https://stackoverflow.com/questions/20446053/k-means-lloyd-forgy-macqueen-hartigan-wong

[19] *Data Mining Algorithms In R/Clustering/K-Means* [online]. [cit. 2018-04-28]. Dostupné z: https://en.wikibooks.org/wiki/Data_Mining_Algorithms_In_R/Clustering/K-Means#Algorithm

[20] *Hartigan's K-means versus Lloyd's K-means: is it time for a change?* [online]. [cit. 2018-04-28]. Dostupné z: https://www.researchgate.net/publication/262246454_Hartigan%27s_K-means_versus_Lloyd%27s_K-means_is_it_time_for_a_change

[21] *Word Cloud* [online]. [cit. 2018-04-28]. Dostupné z: http://www.betterevaluation.org/en/evaluation-options/wordcloud

[22] *Schulze method* [online]. 2017 [cit. 2018-04-22]. Dostupné z: https://wiki.electorama.com/wiki/Schulze_method

[23] *Beatpath* [online]. 2006 [cit. 2018-04-22]. Dostupné z: https://wiki.electorama.com/wiki/Beatpath

[24] *(Partial) Explanation of Schulze "beatpath voting," a complicated Condorcet-type voting system.* [online]. [cit. 2018-04-22]. Dostupné z: http://rangevoting.org/SchulzeComplic.html

[25] POSPÍŠIL, Lukáš a Vít VONDRÁK. *NUMERICKÉ METODY I.*

[26] VICHER, Miroslav. *NUMERIKA*. 2001.

[27] ŠKVÁRA, Jiří. *Detekce rozhraní na množině bodů a určení jeho vlastností.* BAKALÁŘSKÁ PRÁCE. Univerzita Jana Evangelisty Purkyně v Ústí nad Labem. Vedoucí práce RNDr. Jiří Škvor, Ph.D.

[28] FITZPATRICK, Richard. *The Ising model* [online]. [cit. 2018-04-22]. Dostupné z: http://farside.ph.utexas.edu/teaching/329/lectures/node110.html

[29] SELINGER, Jonathan. *Introduction to the Theory of Soft Matter: From Ideal Gases to Liquid Crystals.* 1. Springer International Publishing, 2016. ISBN 978-3-319-21054-4.

[30] *MOLEKULÁRNÍ DYNAMIKA V MATERIÁLOVÉM VÝZKUMU* [online]. [cit. 2018-04-28]. Dostupné z: http://www.cdm.cas.cz/publications/hora/ph_habl_sld.pp4.pdf

[31] *Molecular Dynamics Simulation of Hard Spheres* [online]. [cit. 2018-04-28].

Dostupné z:
https://introcs.cs.princeton.edu/java/assignments/collisions.html

[32] Elastic collision. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001-, 28. 12. 2017 [cit. 2018-05-03]. Dostupné z:
https://en.wikipedia.org/wiki/Elastic_collision

.1. Obsah přiloženého CD

soubor	obsah
bp.pdf	text bakalářské práce ve formátu .pdf
bp.Rnw	text bakalaářské práce ve formátu .Rnw - zdrojový kód
colors.json	pomocný soubor obsahující seznam barev
point-gen.R	pomocný soubor pro generování náhodných bodů
delaunay.R	backend úlohy Delaunayovy triangulace
delaunay-shiny.R	Shiny aplikace Delaunayovy triangulace
ising.R	backend úlohy Isingova modelu
ising-shiny.R	Shiny aplikace Isingova modelu
kmeans.R	backend úlohy K-means
kmeans-shiny.R	Shiny aplikace K-means
mol.R	backend úlohy molekulární simulace tuhých disků
mol-shiny.R	Shiny aplikace molekulární simulace tuhých disků
root.R	backend úlohy hledání kořenu funkce
root-shiny.R	Shiny aplikace hledání kořenu funkce
schulze.R	backend úlohy Schulzeho metody
schulze-shiny.R	Shiny aplikace Schulzeho metody
wordcloud.R	backend úlohy word cloud
wordcloud-shiny.R	Shiny aplikace word cloud