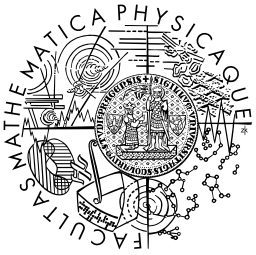


MATEMATICKO-FYZIKÁLNÍ FAKULTA
Univerzita Karlova

**BAKALÁŘSKÁ PRÁCE**

Maximilian Kulikov

# Emulátor zvukových syntezátorů

Ústav formální a aplikované lingvistiky

Vedoucí bakalářské práce: Mgr. David Klusáček, Ph.D.
Studijní program: Informatika
Studijní obor: IOI

Praha 2022

Prohlašuji, že jsem tuto bakalářskou práci vypracoval(a) samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů. Tato práce nebyla využita k získání jiného nebo stejného titulu.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V ..... dne .....  .....

Podpis autora

Děkuji Mgr. Davidovi Klusáčkovi, Ph.D. za veškerou pomoc s mnohými teoretickými i praktickými otázkami v průběhu práce a nasměrování k výsledné podobě práce.

Název práce: Emulátor zvukových syntezátorů

Autor: Maximilian Kulikov

Ústav: Ústav formální a aplikované lingvistiky

Vedoucí bakalářské práce: Mgr. David Klusáček, Ph.D., Ústav formální a aplikované lingvistiky

Abstrakt: Nástroj na tvoření emulátorů zvukových syntezátorů. Základem práce je imperativní programovací jazyk Cynth popisující signály tvořící výsledný zvuk. Kód v jazyce Cynth se přeloží do jazyka C pro následující slinkování s programem řídícím GUI a MIDI vstupní ovládání a výstupní monitorování a napojení zvukové karty. Mezikrok s překladem do jazyka C přináší výhodu z využití optimalizací překladače C. Jazyk Cynth je omezený tak, aby za běhu nedocházelo k žádné dynamicé alokaci, ale zároveň umožňuje komplexní programování za překladu a práci se staticky alokovanými datovými strukrurami určenými k expresivnímu popisu signálů. Program je cílený na platformu Windows a zvukové karty podporující technologii ASIO.

Klíčová slova: syntezátor emulátor programovací jazyk

Title: Sound Synthesizer Emulator

Author: Maximilian Kulikov

Institute: Institute of Formal and Applied Linguistics

Supervisor: Mgr. David Klusáček, Ph.D., Institute of Formal and Applied Linguistics

Abstract: A tool for creation of emulators of audio synthesizers. The base of the work is an imperative programming language Cynth that describes singnals of the resulting sound. Cynth code is translated into C code for further linkage with a program that controls GUI and MIDI input controls and output monitoring and connection with a sound card. The intermediate step of translation into C allows taking advantage of the C compiler optimizations. The Cynth language is restricted in a way that eliminates any dynamic allocations at run-time while allowing complex compile-time programming and working with statically allocated data structures for expressive description of signals. The program's target platform is Windows and sound card's supporting the ASIO technology.

Keywords: synthesizer emulator programming language

# Obsah

# Úvod

Původním cílem bylo vytvořit nástroj nástroj na tvoření emulátorů zvukových syntezátorů v následujícím rozsahu: Programovací jazyk Cynth, který umožní popsat signály a deklarovat vstupy, výstupy a ovládací prvky. Kód v jazyce Cynth se přeloží do jazyka C pro následující statické slinkování s předkompilovaným řídícím programem. Tento řídící program měl mít na starosti komunikaci se zvukovou kartou a vykreslení GUI a napojení MIDI k ovládání a monitorování. V průběhu práce se ukázalo, že jde o mnohem komplexnější cíl, než jsem si původně představoval. Kompletní práci dle původního zadání se mi bohužel dokončit včas nepodařilo. Tato práce je omezena jen na uvedený konfigurační programovací jazyk a jeho překladač do jazyka C. Práce obsahuje popis implementace, neformální specifikaci jazyka, uživatelský manuál, uvedení vlastností a konstruktů jazyka Cynth, které byly pro uvedené napojení s řídícím programem navrženy a příklady implementace různých praktických signálů.

# 1. Specifikace

## 1.1 Syntaxe

Syntaxe jazyka je při překladu analyzována ve dvou fázích, a to lexerem a parserem. Lexer vstupní text rozdělí na lexikální prvky, tzv **tokeny**, se kterými dále pracuje parser. Parser z přečtených tokenů sestaví AST (abstract syntax tree), které se dále transformují do pomocných sémantických struktur nebo do výsledného kódu v jazyce C.

### 1.1.1 Tokeny

Mezi tokeny patří několik tzv. **klíčových slov** (neboly **keywordů**). Ty jsou definovány jako řetězce písmen necitlivé na velikost písmen. Uvedu je výčtem:

```
buffer by const
else false fn
for if in
out return self
to true type
void when while
```

`self` je jen vyhrazeno pro další verze. (Bude reprezentovat rekurzivní volání funkce.) Dále jde o tzv. **symboly** definovné jako řetězce speciálních znaků. Opět uvedu výčtem:

```
( ) [ ] { }
+ - * / % **
! && ||
== != >= <= > <
, ; = \$ ...
```

Symbol **auto** (`$`) je vyhrazen pro odvození typu, ale není implementován v této verzi.

TODO: Escaping dollar signs.

TODO: Hyphenation.

Zbylé tokeny jsou specifické řetězce definovné regulárními výrazy. Uvedu je výčtem názvů tokenů a odpovídajících regulárních výrazů:

```
blank     [ \t]
endl      [\n\r]
comment   ("//"|#).*
multicom  "/*"([^*]|(\*+[^*/]))*\*+\/
name      [a-z][a-zA-Z0-9_]*
type_name [A-Z][a-zA-Z0-9_]*
int       [0-9]+([eE][+-]?[0-9]+)?
float     ([0-9]+\.[0-9]*|\.[0-9]+)([eE][+-]?[0-9]+)?
```

Merezy, konce řádků a komentáře (`blank`, `endl`, `comment`, `multicom`) parser ignoruje. Ostatní tokeny z výčtu výše nesou sémantickou hodnotu danou odpovídajícími řetězcemi, se kterou parser dál pracuje.

Jména (`name`), resp. jména typů (`type_name`), reprezentují názvy proměnných, resp. typů. Pro jednodušší analýzu jsou rozlišeny počátečním znakem. Jména začínají malým písmenem, zatímco jména typů začínají velkým písmenem. Ani jména ani jména typů se nemohou shodovat s žádným z klíčových slov. `int`, `float` a `string` reprezentují literály odpovídajících typů. Číselné literály (`int` i `float`) umožnují zápis ve vědecké notaci. `float` umožňuje vynechat část před, nebo po desetinné tečce.

Jsou podporovány řádkové komentáře `//...` a `#...` a také víceřádkové `/*...*/` ve stylu C. Víceřádkové komentáře nepodporují vnořené komentáře stejného typu. *Pro podporu takového vnoření by byl potřeba zásobníkový automat při lexikální analýze a pouhé regulární výrazy by pro popis takového tokenu nestačily.*

### 1.1.2 Gramatika

Gramatiku uvedu ve formátu Bisonu s drobnými zjednodušeními v zápisu terminálů.

Gramatika obsahuje následující terminály: `int`, `float`, `string`, `name`, `type_name` a znaky uvozené v ‘...‘. Terminály mohou být pro stručnost spojeny do jednoho uvození ‘...‘ a odděleny jen mezerou. Vše ostatní jsou neterminály.

Kořen:

```
start: empty | stmt_list | stmt_list ‘;‘
```

Sémantické kategorie: (Odpovídají sémantickým prvkům.)

```
cat_declaration: node_declaration | paren_decl
cat_range_decl: node_range_decl | paren_range_decl
cat_array_elem: node_range_to | node_range_to_by | node_spread |
    cat_expression
cat_type: node_auto | node_type_name | node_function_type |
    node_array_type | node_buffer_type | node_const_type |
    node_in_type | node_out_type | paren_type
cat_expression: expr_or | expr_right
cat_statement: pure | cat_expression
```

Syntaktické kategorie: (Neodpovídají žádným sémantickým prvkům, pouze zajišťují správné přednosti operátorů.)

```
pure: node_declaration | node_definition | node_assignment |
    node_type_def | node_function_def | node_return | node_if |
    node_for | node_while | node_when
expr_or: node_or | expr_and
expr_and: node_and | expr_eq
expr_eq: node_eq | node_ne | expr_ord
expr_ord: node_ge | node_le | node_gt | node_lt | expr_add
expr_add: node_add | node_sub | expr_mul
expr_mul: node_mul | node_div | node_mod | expr_pow
```

```
expr_pow: node_pow | expr_pre
expr_pre: node_minus | node_plus | node_not | expr_post
expr_post: node_application | node_conversion | node_subscript |
    expr_atom
expr_atom: node_name | node_bool | node_int | node_float | node_string |
    node_block | node_array | paren_expr
expr_right: node_expr_if | node_expr_for | node_function
expr_assgn_target: expr_post
```

Typy:

```
paren_type: ‘(‘ cat_type ‘)‘ | ‘(‘ type_list ‘)‘ | ‘(‘ type_list ‘, )‘
void_type: ‘( )‘ | ‘void‘
node_auto: ‘\$‘
node_type_name: type_name
node_const_type: cat_type ‘const‘
node_in_type: cat_type ‘in‘
node_out_type: cat_type ‘out‘
node_function_type: cat_type paren_type | void_type paren_type |
    cat_type void_type | void_type void_type
node_array_type: cat_type ‘[‘ cat_expression ‘]‘ | cat_type ‘[ \$ ]‘ |
    cat_type ‘[ ]‘ | cat_type ‘[‘ cat_declaration ‘]‘
node_buffer_type: ‘buffer[‘ cat_expression ‘]‘
```

Deklarace:

```
paren_range_decl: ‘(‘ cat_range_decl ‘)‘ | ‘(‘ range_decl_list ‘)‘ |
    ‘(‘ range_decl_list ‘, )‘
paren_decl: ‘(‘ cat_declaration ‘)‘ | ‘(‘ decl_list ‘)‘ |
    ‘(‘ decl_list ‘, )‘
void_decl: ‘( )‘
node_declaration: cat_type node_name
node_range_decl: cat_declaration ‘in‘ cat_expression
```

Speciální prvky polí:

```
node_range_to: cat_expression ‘to‘ cat_expression
node_range_to_by: cat_expression ‘to‘ cat_expression ‘by‘ cat_expression
node_spread: ‘...‘ cat_expression
```

Literály:

```
node_bool: ‘true‘ | ‘false‘
node_int: int
node_float: float
node_string: string
node_function: cat_type ‘fn‘ paren_decl cat_expression |
    void_type ‘fn‘ paren_decl cat_expression |
    cat_type ‘fn‘ void_decl cat_expression |
    void_type ‘fn‘ void_decl cat_expression
node_array: ‘[ ]‘ | ‘[‘ array_elem_list ‘]‘ | ‘[‘ array_elem_list ‘; ]‘
```

Operátory:

```
node_or: expr_or ‘||‘ expr_and
node_and: expr_and ‘&&‘ expr_eq
```

```
node_eq: expr_eq `==` expr_ord
node_ne: expr_eq `!=` expr_ord
node_ge: expr_ord `>=` expr_add
node_le: expr_ord `<=` expr_add
node_gt: expr_ord `>` expr_add
node_lt: expr_ord `<` expr_add
node_add: expr_add `+` expr_mul
node_sub: expr_add `-` expr_mul
node_mul: expr_mul `*` expr_pow
node_div: expr_mul `/` expr_pow
node_mod: expr_mul `\%` expr_pow
node_pow: expr_pre `**` expr_pow
node_minus: `-` expr_pre
node_plus:  `+` expr_pre
node_not:   `!` expr_pre
node_application: expr_post paren_expr | expr_post `void`
node_conversion: cat_type paren_expr
node_subscript: expr_post `[` array_elem_list `]` | expr_post `[ ]`
node_expr_if: `if` paren_expr cat_expression `else` cat_expression
node_expr_for: `for` paren_range_decl cat_expression
```

Ostatní výrazy:

```
paren_expr: `(` cat_expression `)` | `(` expr_list `)` |
    `(` expr_list `, )`
void: `( )`
node_name: name
node_block: `{ }` | `{` stmt_list `}` | `{` stmt_list `; }`
```

Příkazy:

```
node_definition: cat_declaration `=` cat_expression
node_assignment: expr_assgn_target `=` cat_expression
node_type_def: `type` node_type_name `=` cat_type
node_function_def: cat_type node_name paren_decl cat_expression |
    void_type node_name paren_decl cat_expression |
    cat_type node_name void_decl cat_expression |
    void_type node_name void_decl cat_expression
node_return: `return` cat_expression | `return void` | `return`
node_if: `if` paren_expr pure `else` pure |
    `if` paren_expr pure `; else` pure |
    `if` paren_expr cat_expression `else` pure |
    `if` paren_expr cat_expression SEMI `else` pure |
    `if` paren_expr pure `else` cat_expression |
    `if` paren_expr pure SEMI `else` cat_expression
node_when: `when` paren_expr cat_statement
node_for: `for` paren_range_decl pure
node_while: `while` paren_expr cat_statement
```

Pomocné struktury:

```
array_elem_list: cat_array_elem | array_elem_list `,` cat_array_elem
stmt_list: cat_statement | stmt_list `;` cat_statement
type_list: cat_type `,` cat_type | type_list `,` cat_type
```

```
expr_list: cat_expression ‘,‘ cat_expression |
    expr_list ‘,‘ cat_expression
decl_list: cat_declaration ‘,‘ cat_declaration |
    decl_list ‘,‘ cat_declaration
range_decl_list: cat_range_decl ‘,‘ cat_range_decl |
    range_decl_list ‘,‘ cat_range_decl
```

V zásadě je Cynth na první pohled syntakticky podobný jazyku C. Podoba s jazykem C ale nebyla zásadním kritériem při návrhu syntaxe. Spíše jsem se snažil o zobecnění syntaxe C do něčeho více konzistentního, bez zbytečných výjimek z obecných pravidel.

Celý program je výraz, konkrétně výraz block (`node_block`) s implicitními složenými závorkami. Block je výraz, který je tvořen složenými závorkami uzávorkovanými a středníky oddělenými příkazy. Přikazy zahrnují i výrazy. Každý výraz je n-ticí výrazů. N-tice jsou tvořeny uzávorkovaným seznamem výrazů oddělených čárkami, nebo, v případě 1-tice, samotným neuzávorkovaným výrazem. N-tice jsou ploché, tedy n-tice obsahující právě dvé dvojice je čtveřicí stejně jako n-tice obsahující právě 4 1-tice. Uzávorkování v n-tici nemění její sémantiku.

```
1;          # 1-tice
(1);        # 1-tice
(1, 2);     # dvojice
(1, 2, 3)   # trojice
(1, (2, 3)) # trojice
```

To samé platí pro n-tice typů a n-tice deklarací.

```
type T1 = Int;          # 1-tice typů
type T2 = (Int);        # 1-tice typů
type T3 = (Int, Float); # dvojice typů
Int v1;                 # 1-tice deklarací
(Int v2, Float v3);     # dvojice deklarací
(Int, Int) v4;          # 1-tice deklarací s dvojicí typů
```

Speciálním případem jsou 0-tice. Výskyt 0-tic je syntakticky omezen na několik specifických případů, kde mají smysl. 0-tice se také nazývají prázdné hodnoty (výrazy), typy, či deklarace. Navíc jsem na zkoušku přidal alternativní syntaxi prázdných výstupních typů funkcí s klíčovým slovem `void` pro případ, že by se syntaxe s prázdnými závorkami osvědčila jako příliš matoucí pro uživatele zvyklé na exvivlentní syntaxi v C.

```
f();                # prázdná hodnota v aplikaci funkce
return ();          # prázdná hodnota ve vrácení z procedury
type F1 = Int ();   # prázdný vstupní typ funkce
type F2 = () ();    # prázdný vstupní i výstupní typ funkce
type F2 = void ();  # prázdný vstupní i výstupní typ funkce
Int fn () {};       # prázdná deklarace parametrú funkce
() fn (Int x) {};   # prázdný výstupní typ funkce
void fn (Int x) {}; # prázdný výstupní typ funkce
```

TODO: More notes about the syntax?

## 1.2 Sémantika

Program je složený z deklarací, typů, příkazů a výrazů. Mezi příkazy kromě čistě příkazových konstruktů patří deklarace a výrazy. Příkazy se vykonávají. Výrazy se vyhodnocují. Vyhodnocený výraz má vždy tzv. **výslednou hodnotu**, zatímco vykonaný příkaz může (ale nemusí) mít tzv. **návratovou hodnotu**. Zkráceně řečeno: výraz „**má hodnotu**“ a příkaz „**vrací**“. Výsledné i návratové hodnoty jsou dány n-ticemi. Vykonání příkazu tvořeným výrazem znamená vyhodnocení výrazu bez použití jeho hodnoty. Takto vykonané příkazy nemají návratovou hodnotu. Sémantiku vykonávání deklarací a čistě příkazových konstruktů popíšu dále individuálně.

### 1.2.1 Typy

Každý výraz má určený **typ**, resp. n-tici typů.

Základem jsou tzv. **jednoduché typy**: **Bool**, **Int**, **Float**. Hodnoty typu Bool reprezentují binární booleovské hodnoty, které lze vytvořit literály `true` a `false`. Hodnoty typu Int reprezentují celá čísla. Jsou omezeny shora i zdola hodnotami danými implementací. Sémantika přesahu krajních hodnot je také dána implementací. Tyto hodnoty lze vyrvořit **celočíselným literálem** (token `int`). Typ Float reprezentuje reálná čísla. Implementačně jde o floating-point hodnotu, tedy opět je třeba počítat s omezenými hodnotami a také s omezenou přesností. Float hodnoty lze vyrvořit **destinným literálem** (token `float`).

*V aktuální verzi je Int implementován pomocí typu int v C++ compile-time interpreteru i v C run-time kódu, velikost kterého závisí na platformě. V C++ je dvojkový doplněk zaručen. V C tomu tak být nemusí. Vzhledem k omezení cílové platformy na moderní verze Windows a překladače Clang a GCC se však lze spolehnout na dvojkový doplněk a velikost alespoň 32 bitů. Float je implementován typem float v obou prostředích. Standardní IEEE floaty nejsou zaručeny standardem, ale opět vzhledem k omezené cílové platformě se s tím dá počítat. V dalších verzích bude vhodné přidat konfiguraci požadavků na tyto typy v implementaci.*

Hodnoty jednoduchých typů se **předávají** tzv. **hodnotou** (**kopií**). (Předávání bude podrobněji definováno později.) To znamená, že při předávání se jejich hodnota kopíruje a ta původní zůstává nedotčená a z nové skopírované hodnoty nepřístupná.

TODO: Předávání. (Pass)

Dále jsou k dispozici tzv. **referenční typy**, které se předávají **referencí** (neboli **odkazem**). To znamená, že tyto hodnoty reprezentují **referenci** (**odkaz**) na jinou hodnotu. Samotná reference se předává hodnotou a nelze vytvořit další referenci odkazující na ní. (V aktuální verzi.) Odkazovaná hodnota se ale při předávání této reference nekopíruje. Skrze odkaz lze hodnotu modifikovat.

**Pole** (neboli **array**) reprezentuje referenci na pevně daný počet jednoduchých hodnot umístěných v daném pořadí. Typům polí v gramatice odpovídá neterminál `node_array_type`. Určuje ho typ odkazované hodnoty a počet odkazovaných (neboli **obsažených**) hodnot. **Vstupní** (neboli **input**), resp. **výstupní** (neboli **out-**

**put**), typy (zkráceně **IO typy**) reprezentují referenci na hodnotu, kterou lze jen číst, resp. jen modifikovat. Těmto typům odpovídají neterminály `node_in_type` a `node_out_type`. Určuje ho typ obsažené hodnoty. **Buffery** reprezentují referenci na peveně daný počet hodnot typu Float umístěných v daném pořadí, které se za běhu cyklicky protáčí. Bufferům odpovídá neterminál `node_buffer_type`. Určuje ho jen počet obsažených hodnot.

Mimo roztřídění na jednoduché a referenční typy jsou funkce. Funkce jsou hodnoty a z pohledu uživatele se předávají hodnotou. Nejsou ale považovány za jednoduché typy kvůli odlišným pravidlům mutability. Funkce představují zobrazení mezi hodnotami se side effecty, tedy obsahují spustitelnou paramertizovatelnou část programu, která může při spuštění ovlivnit zbytek programu.

K typům lze navíc specifikovat, zda jde o **imutabilní** (neboli **konstantní**) hodnotu přidáním keywordu **cost** za daný typ. Tomu odpovídá neterminál gramatiky `node_const_type`. Jednoduché typy mohou být konstantní i nekonstantní. Pole mohou mít konstantní i nekonstantní referenci na konstantní i nekonstantní hodnoty. Imutabilita reference se určí keywordem const za celým typem funkce, zatímco imutabilita hodnot se určí keywordem const za typem hodnot uvnitř celého typu pole. IO typy mají konstantní referenci na nekonstantní hodnoty. Přidání keywordu const za typ hodnoty je chybou, zatímco přidání keywordu const za celý IO typ je zbytečné, ale není chybou. Pro buffery platí to samé. Přidání keywordu const za typ bufferu je zbytečné, ale ne chybné. Funkce jsou imutabilní. Přidání keywordu const je opět zbytečné, ale není chybou.

### 1.2.2 Block

Odpovídající neterminál v gramatice: `node_block`.

**Block** (neboli **blok**, obzvlášť při skloňování) je výraz, či příkaz, který obsahuje seznam tzv. **vnitřních příkazů**. Block je výrazem, pokud se nachází na místě, kde gramatika očekává výraz, nikoliv příkaz. V opačném případě je čistě příkazovým konstruktem. Vyhodnocení bloku v obou případech znamená vytvoření **scopu** a vykonání všech vnitřních příkazů v uvedeném pořadí. (Scope bude podrobněji definován dále.) Výrazový block se navíc vyhodnotí s výslednou hodnotou danou návratovou hodnotou prvního vykonaného vnitřního příkazu, který vrací. Pokud žádný takový příkaz ve výrazovém bloku není, jedná se o sémantickou chybu. Příkazový block se vykoná s návratovou hodnotou danou prvním vnitřním příkazem, který vrací. Příkazový block vrací vždy, pokud obsahuje vnitřní příkaz, který vrací vždy. Příkazový block nevrací, pokud neobsahuje žádný vnitřní příkaz, který vrací. Všechny vnitřní příkazy se musí shodovat v typu návratové hodnoty.

### 1.2.3 Return

Odpovídající neterminál v gramatice: `node_return`.

**Return**, neboli **vrácení**, je příkaz, který vždy vrací uvedenou hodnotu. Return může vracet i prázdnou hodnotu. Pokud se hodnota neuvede (`return`), implicitně se vrátí prázdná hodnota (`return ()`).

### 1.2.4 Deklarace

Stejně jako typy a hodnoty jsou i deklarace n-ticemi. 1-tice deklarace je dána typem (resp. n-ticí typů) a jménem. Takové deklarace lze skládat do libovolných n-tic. Reprezentují pojmenované proměnné obsahující hodnotu daného typu. Deklarace mohou samy o sobě tvořit **příkaz deklarace**, nebo mohou být použity jako deklarace parametrů funkce.

Některé příkazy vytváří tzv. **scope**, což je část programu, která určuje oblast přístupnosti proměnných.

TODO: Scope.

# 2. Nápověda k sazbě

## 2.1 Úprava práce

Vlastní text bakalářské práce je uspořádaný hierarchicky do kapitol a podkapitol, každá kapitola začíná na nové straně. Text je zarovnán do bloku. Nový odstavec se obvykle odděluje malou vertikální mezerou a odsazením prvního řádku. Grafická úprava má být v celém textu jednotná.

Práce se tiskne na bílý papír formátu A4. Okraje musí ponechat dost místa na vazbu: doporučen je horní, dolní a pravý okraj 25 mm, levý okraj 40 mm. Číslují se všechny strany kromě obálky a informačních stran na začátku práce; první číslovaná strana bývá obvykle ta s obsahem.

Písmo se doporučuje dvanáctibodové (12 pt) se standardní vzdáleností mezi řádky (pokud píšete ve Wordu nebo podobném programu, odpovídá tomu řádkování 1,5; v TEXu není potřeba nic přepínat). Pro běžný text používejte vzpřímené patkové písmo. Text matematických vět se obvykle tiskne pro zdůraznění skloněným (slanted) písmem, není-li k dispozici, může být zastoupeno kurzívou.

Primárně je doporučován jednostranný tisk (příliš tenkou práci lze obtížně svázat). Delší práce je lepší tisknout oboustranně a přizpůsobit tomu velikosti okrajů: 40 mm má vždy *vnitřní* okraj. Rub titulního listu zůstává nepotištěný.

Zkratky použité v textu musí být vysvětleny vždy u prvního výskytu zkratky (v závorce nebo v poznámce pod čarou, jde-li o složitější vysvětlení pojmu či zkratky). Pokud je zkratek více, připojuje se seznam použitých zkratek, včetně jejich vysvětlení a/nebo odkazů na definici.

Delší převzatý text jiného autora je nutné vymezit uvozovkami nebo jinak vyznačit a řádně citovat.

## 2.2 Jednoduché příklady

Čísla v českém textu obvykle sázíme v matematickém režimu s desetinnou čárkou: $\pi \doteq 3{,}141\,592\,653\,589$. V matematických textech se považuje za přípustné používat desetinnou tečku (pro lepší odlišení od čárky v roli oddělovače). Numerické výsledky se uvádějí s přiměřeným počtem desetinných míst.

Mezi číslo a jednotku patří úzká mezera: šířka stránky A4 činí 210 mm, což si pamatuje pouze 5 % autorů. Pokud ale údaj slouží jako přívlastek, mezeru vynecháváme: 25mm okraj, 95% interval spolehlivosti.

Rozlišujeme různé druhy pomlček: červeno-černý (krátká pomlčka), strana 16–22 (střední), $45 - 44$ (matematické minus), a toto je — jak se asi dalo čekat — vložená věta ohraničená dlouhými pomlčkami.

V českém textu se používají „české“ uvozovky, nikoliv “anglické”.

Na některých místech je potřeba zabránit lámání řádku (v~TEXu značíme vlnovkou): u~předložek (neslabičnych, nebo obecně jednopísmenných), vrchol~$v$, před

$k$~kroky, a~proto, … obecně kdekoliv, kde by při rozlomení čtenář „škobrtnul“.

## 2.3 Matematické vzorce a výrazy

Proměnné sázíme kurzívou (to TEX v matematickém módu dělá sám, ale nezapomínejte na to v okolním textu a také si matematický mód zapněte). Názvy funkcí sázíme vzpřímeně. Tedy například: $\operatorname{var}(X) = \mathsf{E}\, X^2 - \big(\mathsf{E}\, X\big)^2$.

Zlomky uvnitř odstavce (třeba $\frac{5}{7}$ nebo $\frac{x+y}{2}$) mohou být příliš stísněné, takže je lepší sázet jednoduché zlomky s lomítkem: $5/7$, $(x+y)/2$.

Nechť

$$\mathbb{X} = \begin{pmatrix} \boldsymbol{x}_1^\top \\ \vdots \\ \boldsymbol{x}_n^\top \end{pmatrix}.$$

Povšimněme si tečky za maticí. Byť je matematický text vysázen ve specifickém prostředí, stále je gramaticky součástí věty a tudíž je zapotřebí neopomenout patřičná interpunkční znaménka. Výrazy, na které chceme později odkazovat, je vhodné očíslovat:

$$\mathbb{X} = \begin{pmatrix} \boldsymbol{x}_1^\top \\ \vdots \\ \boldsymbol{x}_n^\top \end{pmatrix}. \tag{2.1}$$

Výraz (2.1) definuje matici $\mathbb{X}$. Pro lepší čitelnost a přehlednost textu je vhodné číslovat pouze ty výrazy, na které se autor někde v další části textu odkazuje. To jest, nečíslujte automaticky všechny výrazy vysázené některým z matematických prostředí.

Zarovnání vzorců do několika sloupečků:

$$\begin{aligned} S(t) &= \mathsf{P}(T > t), \qquad t > 0 \qquad \text{(zprava spojitá)}, \\ F(t) &= \mathsf{P}(T \le t), \qquad t > 0 \qquad \text{(zprava spojitá)}. \end{aligned}$$

Dva vzorce se spojovníkem:

$$\left.\begin{aligned} S(t) &= \mathsf{P}(T > t) \\ F(t) &= \mathsf{P}(T \le t) \end{aligned}\right\} \quad t > 0 \qquad \text{(zprava spojité)}. \tag{2.2}$$

Dva centrované nečíslované vzorce:

$$\boldsymbol{Y} = \mathbb{X}\boldsymbol{\beta} + \boldsymbol{\varepsilon},$$

$$\mathbb{X} = \begin{pmatrix} 1 & \boldsymbol{x}_1^\top \\ \vdots & \vdots \\ 1 & \boldsymbol{x}_n^\top \end{pmatrix}.$$

Dva centrované číslované vzorce:

$$\boldsymbol{Y} = \mathbb{X}\boldsymbol{\beta} + \boldsymbol{\varepsilon}, \tag{2.3}$$

$$\mathbb{X} = \begin{pmatrix} 1 & \boldsymbol{x}_1^\top \\ \vdots & \vdots \\ 1 & \boldsymbol{x}_n^\top \end{pmatrix}. \tag{2.4}$$

Definice rozdělená na dva případy:

$$P_{r-j} = \begin{cases} 0, & \text{je-li } r-j \text{ liché,} \\ r!\,(-1)^{(r-j)/2}, & \text{je-li } r-j \text{ sudé.} \end{cases}$$

Všimněte si použití interpunkce v této konstrukci. Čárky a tečky se dávají na místa, kam podle jazykových pravidel patří.

$$\begin{aligned} x &= y_1 - y_2 + y_3 - y_5 + y_8 - \cdots = && \text{z (2.3)} \\ &= y' \circ y^* = && \text{podle (2.4)} \\ &= y(0)y' && \text{z Axiomu 1.} \end{aligned} \tag{2.5}$$

Dva zarovnané vzorce nečíslované:

$$\begin{aligned} L(\boldsymbol{\theta}) &= \prod_{i=1}^{n} f_i(y_i;\, \boldsymbol{\theta}), \\ \ell(\boldsymbol{\theta}) &= \log\Big\{L(\boldsymbol{\theta})\Big\} = \sum_{i=1}^{n} \log\Big\{f_i(y_i;\, \boldsymbol{\theta})\Big\}. \end{aligned}$$

Dva zarovnané vzorce, první číslovaný:

$$\begin{aligned} L(\boldsymbol{\theta}) &= \prod_{i=1}^{n} f_i(y_i;\, \boldsymbol{\theta}), \\ \ell(\boldsymbol{\theta}) &= \log\Big\{L(\boldsymbol{\theta})\Big\} = \sum_{i=1}^{n} \log\Big\{f_i(y_i;\, \boldsymbol{\theta})\Big\}. \end{aligned} \tag{2.6}$$

Vzorec na dva řádky, první řádek zarovnaný vlevo, druhý vpravo, nečíslovaný:

$$\begin{aligned} \ell(\mu, \sigma^2) = \log\Big\{L(\mu, \sigma^2)\Big\} = \sum_{i=1}^{n} \log\Big\{f_i(y_i;\, \mu, \sigma^2)\Big\} = \\ = -\frac{n}{2}\,\log(2\pi\sigma^2) \,-\, \frac{1}{2\sigma^2}\sum_{i=1}^{n}(y_i - \mu)^2. \end{aligned}$$

Vzorec na dva řádky, zarovnaný na =, číslovaný uprostřed:

$$\begin{aligned} \ell(\mu, \sigma^2) = \log\Big\{L(\mu, \sigma^2)\Big\} = \sum_{i=1}^{n} \log\Big\{f(y_i;\, \mu, \sigma^2)\Big\} = \\ = -\frac{n}{2}\,\log(2\pi\sigma^2) \,-\, \frac{1}{2\sigma^2}\sum_{i=1}^{n}(y_i - \mu)^2. \end{aligned} \tag{2.7}$$

## 2.4 Definice, věty, důkazy, ...

Konstrukce typu definice, věta, důkaz, příklad, ... je vhodné odlišit od okolního textu a případně též číslovat s možností použití křížových odkazů. Pro každý typ těchto konstrukcí je vhodné mít v souboru s makry (`makra.tex`) nadefinované jedno prostředí, které zajistí jak vizuální odlišení od okolního textu, tak automatické číslování s možností křížově odkazovat.

**Definice 1.** *Nechť náhodné veličiny $X_1, \ldots, X_n$ jsou definovány na témž pravděpodobnostním prostoru $(\Omega, \mathcal{A}, P)$. Pak vektor $\boldsymbol{X} = (X_1, \ldots, X_n)^\top$ nazveme* náhodným vektorem.

**Definice 2** (náhodný vektor)**.** *Nechť náhodné veličiny $X_1, \ldots, X_n$ jsou definovány na témž pravděpodobnostním prostoru $(\Omega, \mathcal{A}, P)$. Pak vektor $\boldsymbol{X} = (X_1, \ldots, X_n)^\top$ nazveme* náhodným vektorem.

Definice 1 ukazuje použití prostředí pro sazbu definice bez titulku, definice 2 ukazuje použití prostředí pro sazbu definice s titulkem.

**Věta 1.** *Náhodný vektor $\boldsymbol{X}$ je měřitelné zobrazení prostoru $(\Omega, \mathcal{A}, P)$ do $(\mathbb{R}_n, \mathcal{B}_n)$.*

**Lemma 2** (Anděl, 2007, str. 29)**.** *Náhodný vektor $\boldsymbol{X}$ je měřitelné zobrazení prostoru $(\Omega, \mathcal{A}, P)$ do $(\mathbb{R}_n, \mathcal{B}_n)$.*

*Důkaz.* Jednotlivé kroky důkazu jsou podrobně popsány v práci Anděl (2007, str. 29).

□

Věta 1 ukazuje použití prostředí pro sazbu matematické věty bez titulku, lemma 2 ukazuje použití prostředí pro sazbu matematické věty s titulkem. Lemmata byla zavedena v hlavním souboru tak, že sdílejí číslování s větami.

# 3. Odkazy na literaturu

Odkazy na literaturu vytváříme nejlépe pomocí příkazů `\citet`, `\citep` atp. (viz LaTeXový balíček natbib) a následného použití BibTeXu. V matematickém textu obvykle odkazujeme stylem „Jméno autora/autorů (rok vydání)“, resp. „Jméno autora/autorů [číslo odkazu]“. V českém/slovenském textu je potřeba se navíc vypořádat s nutností skloňovat jméno autora, respektive přechylovat jméno autorky. Je potřeba mít na paměti, že standardní příkazy `\citet`, `\citep` produkují referenci se jménem autora/autorů v prvním pádě a jména autorek jsou nepřechýlena.

Pokud nepoužíváme bibTeX, řídíme se normou ISO 690 a zvyklostmi oboru.

Jména časopisů lze uvádět zkráceně, ale pouze v kodifikované podobě.

## 3.1 Několik ukázek

Mezi nejvíce citované statistické články patří práce Kaplana a Meiera a Coxe (Kaplan a Meier, 1958; Cox, 1972). Student (1908) napsal článek o t-testu.

Prof. Anděl je autorem učebnice matematické statistiky (viz Anděl, 1998). Teorii odhadu se věnuje práce Lehmann a Casella (1998). V případě odkazů na specifickou informaci (definice, důkaz, ...) uvedenou v knize bývá užitečné uvést specificky číslo kapitoly, číslo věty atp. obsahující požadovanou informaci, např. viz Anděl (2007, Věta 4.22) nebo (viz Anděl, 2007, Věta 4.22).

Mnoho článků je výsledkem spolupráce celé řady osob. Při odkazování v textu na článek se třemi autory obvykle při prvním výskytu uvedeme plný seznam: Dempster, Laird a Rubin (1977) představili koncept EM algoritmu. Respektive: Koncept EM algoritmu byl představen v práci Dempstera, Lairdové a Rubina (Dempster, Laird a Rubin, 1977). Při každém dalším výskytu již používáme zkrácenou verzi: Dempster a kol. (1977) nabízejí též několik příkladů použití EM algoritmu. Respektive: Několik příkladů použití EM algoritmu lze nalézt též v práci Dempstera a kol. (Dempster a kol., 1977).

U článku s více než třemi autory odkazujeme vždy zkrácenou formou: První výsledky projektu ACCEPT jsou uvedeny v práci Genbergové a kol. (Genberg a kol., 2008). V textu *nenapíšeme*: První výsledky projektu ACCEPT jsou uvedeny v práci Genberg, Kulich, Kawichai, Modiba, Chingono, Kilonzo, Richter, Pettifor, Sweat a Celentano (2008).

# 4. Tabulky, obrázky, programy

Používání tabulek a grafů v odborném textu má některá společná pravidla a některá specifická. Tabulky a grafy neuvádíme přímo do textu, ale umístíme je buď na samostatné stránky nebo na vyhrazené místo v horní nebo dolní části běžných stránek. LaTeX se o umístění plovoucích grafů a tabulek postará automaticky.

Každý graf a tabulku očíslujeme a umístíme pod ně legendu. Legenda má popisovat obsah grafu či tabulky tak podrobně, aby jim čtenář rozuměl bez důkladného studování textu práce.

Na každou tabulku a graf musí být v textu odkaz pomocí jejich čísla. Na příslušném místě textu pak shrneme ty nejdůležitější závěry, které lze z tabulky či grafu učinit. Text by měl být čitelný a srozumitelný i bez prohlížení tabulek a grafů a tabulky a grafy by měly být srozumitelné i bez podrobné četby textu.

Na tabulky a grafy odkazujeme pokud možno nepřímo v průběhu běžného toku textu; místo *„Tabulka 4.1 ukazuje, že muži jsou v průměru o* 9,9 kg *těžší než ženy“* raději napíšeme *„Muži jsou o* 9,9 kg *těžší než ženy (viz Tabulka 4.1)“*.

## 4.1 Tabulky

U **tabulek** se doporučuje dodržovat následující pravidla:

- Vyhýbat se svislým linkám. Silnějšími vodorovnými linkami oddělit tabulku od okolního textu včetně legendy, slabšími vodorovnými linkami oddělovat záhlaví sloupců od těla tabulky a jednotlivé části tabulky mezi sebou. V LaTeXu tuto podobu tabulek implementuje balík `booktabs`. Chceme-li výrazněji oddělit některé sloupce od jiných, vložíme mezi ně větší mezeru.
- Neměnit typ, formát a význam obsahu políček v tomtéž sloupci (není dobré do téhož sloupce zapisovat tu průměr, onde procenta).
- Neopakovat tentýž obsah políček mnohokrát za sebou. Máme-li sloupec *Rozptyl*, který v prvních deseti řádcích obsahuje hodnotu 0,5 a v druhých deseti řádcích hodnotu 1,5, pak tento sloupec raději zrušíme a vyřešíme to jinak. Například můžeme tabulku rozdělit na dvě nebo do ní vložit popisné řádky, které informují o nějaké proměnné hodnotě opakující se v následujícím oddíle tabulky (např. *„Rozptyl* = 0,5 “ a níže *„Rozptyl* = 1,5 “).

| **Efekt** | **Odhad** | **Směrod. chyba**[a] | **P-hodnota** |
|---|---|---|---|
| Abs. člen | −10,01 | 1,01 | — |
| Pohlaví (muž) | 9,89 | 5,98 | 0,098 |
| Výška (cm) | 0,78 | 0,12 | $< 0{,}001$ |

*Pozn:* [a] Směrodatná chyba odhadu metodou Monte Carlo.

Tabulka 4.1: Maximálně věrohodné odhady v modelu M.

- Čísla v tabulce zarovnávat na desetinnou čárku.
- V tabulce je někdy potřebné používat zkratky, které se jinde nevyskytují. Tyto zkratky můžeme vysvětlit v legendě nebo v poznámkách pod tabulkou. Poznámky pod tabulkou můžeme využít i k podrobnějšímu vysvětlení významu některých sloupců nebo hodnot.

## 4.2 Obrázky

Několik rad týkajících se obrázků a grafů.

- Graf by měl být vytvořen ve velikosti, v níž bude použit v práci. Zmenšení příliš velkého grafu vede ke špatné čitelnosti popisků.
- Osy grafu musí být řádně popsány ve stejném jazyce, v jakém je psána práce (absenci diakritiky lze tolerovat). Kreslíme-li graf hmotnosti proti výšce, nenecháme na nich popisky `ht` a `wt`, ale osy popíšeme *Výška [cm]* a *Hmotnost [kg]*. Kreslíme-li graf funkce $h(x)$, popíšeme osy $x$ a $h(x)$. Každá osa musí mít jasně určenou škálu.
- Chceme-li na dvourozměrném grafu vyznačit velké množství bodů, dáme pozor, aby se neslily do jednolité černé tmy. Je-li bodů mnoho, zmenšíme velikost symbolu, kterým je vykreslujeme, anebo vybereme jen malou část bodů, kterou do grafu zaneseme. Grafy, které obsahují tisíce bodů, dělají problémy hlavně v elektronických dokumentech, protože výrazně zvětšují velikost souborů.
- Budeme-li práci tisknout černobíle, vyhneme se používání barev. Čáry rozlišujeme typem (plná, tečkovaná, čerchovaná,...), plochy dostatečně rozdílnými intensitami šedé nebo šrafováním. Význam jednotlivých typů čar a ploch vysvětlíme buď v textové legendě ke grafu anebo v grafické legendě, která je přímo součástí obrázku.
- Vyhýbejte se bitmapovým obrázkům o nízkém rozlišení a zejména JPEGům (zuby a kompresní artefakty nevypadají na papíře pěkně). Lepší je vytvářet obrázky vektorově a vložit do textu jako PDF.

## 4.3 Programy

Algoritmy, výpisy programů a popis interakce s programy je vhodné odlišit od ostatního textu. Jednou z možností je použití LaTeXového balíčku `fancyvrb` (fancy verbatim), pomocí něhož je v souboru `makra.tex` nadefinováno prostředí `code`. Pomocí něho lze vytvořit např. následující ukázky.

```
> mean(x)
[1] 158.90
> objekt$prumer
[1] 158.90
```

Menší písmo:

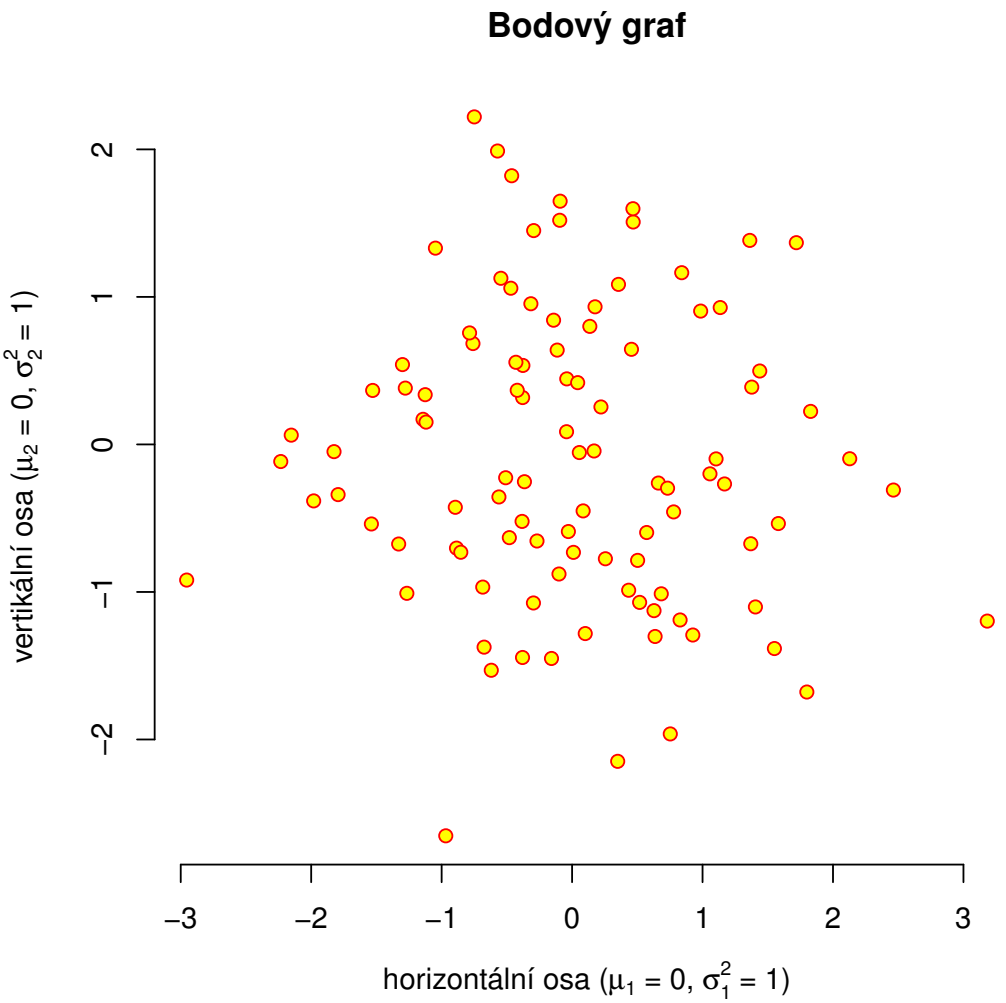
```
> mean(x)
[1] 158.90
> objekt$prumer
[1] 158.90
```

Bez rámečku:

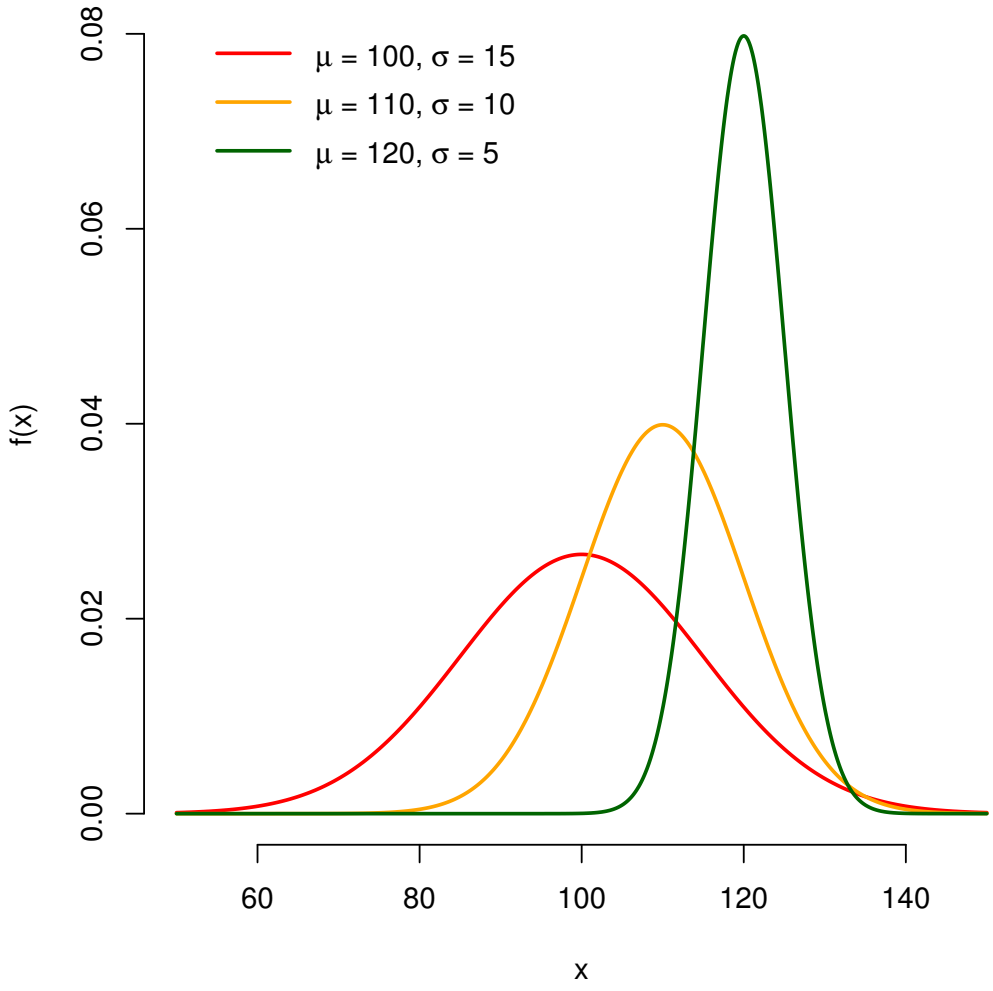
```
> mean(x)
[1] 158.90
> objekt$prumer
[1] 158.90
```

Užší rámeček:

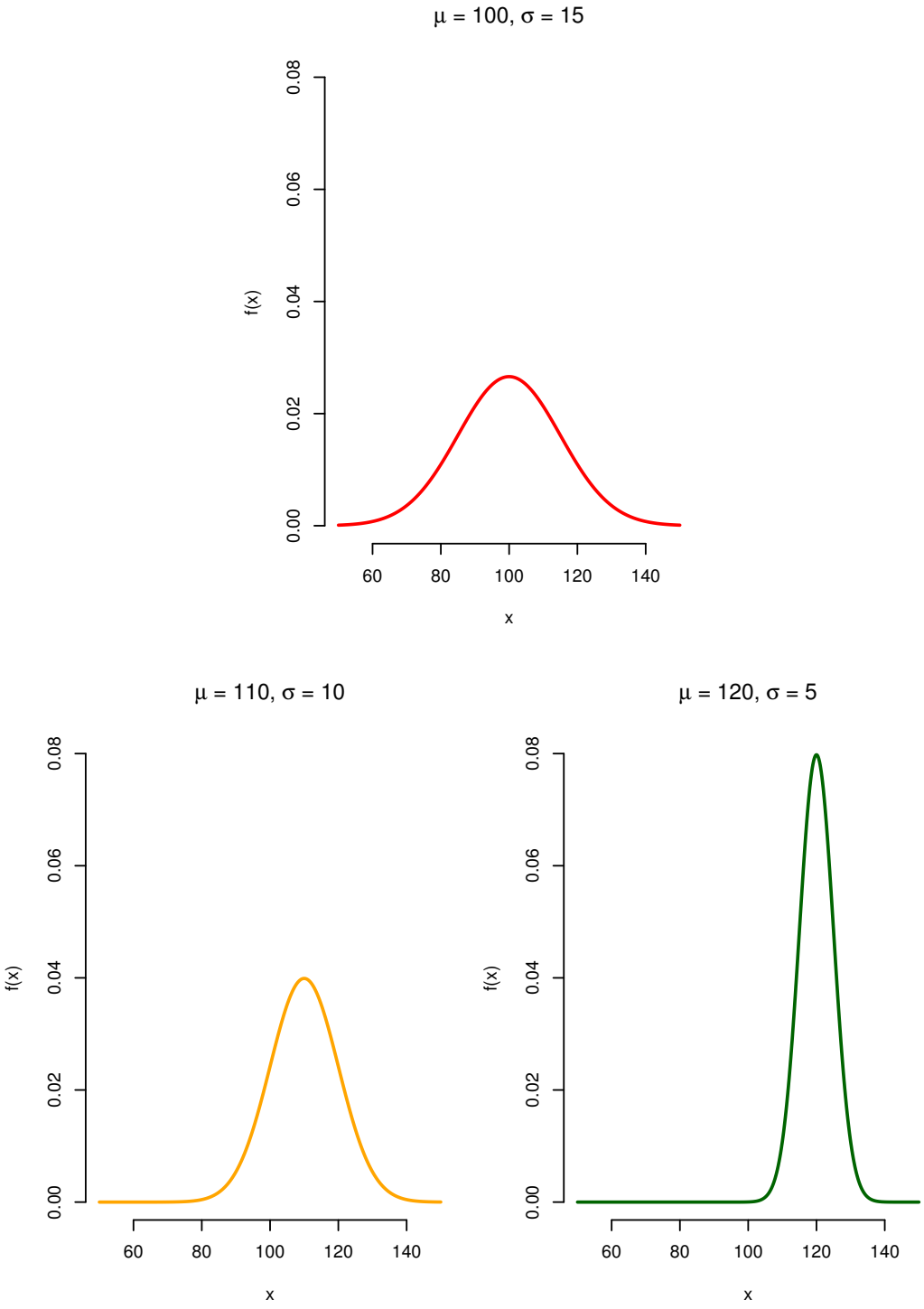
```
> mean(x)
[1] 158.90
> objekt$prumer
[1] 158.90
```




Obrázek 4.1: Náhodný výběr z rozdělení $\mathcal{N}_2(\mathbf{0}, I)$.




Obrázek 4.2: Hustoty několika normálních rozdělení.




Obrázek 4.3: Hustoty několika normálních rozdělení.

# 5. Formát PDF/A

Opatření rektora č. 13/2017 určuje, že elektronická podoba závěrečných prací musí být odevzdávána ve formátu PDF/A úrovně 1a nebo 2u. To jsou profily formátu PDF určující, jaké vlastnosti PDF je povoleno používat, aby byly dokumenty vhodné k dlouhodobé archivaci a dalšímu automatickému zpracování. Dále se budeme zabývat úrovní 2u, kterou sázíme TEXem.

Mezi nejdůležitější požadavky PDF/A-2u patří:

- Všechny fonty musí být zabudovány uvnitř dokumentu. Nejsou přípustné odkazy na externí fonty (ani na „systémové“, jako je Helvetica nebo Times).
- Fonty musí obsahovat tabulku ToUnicode, která definuje převod z kódování znaků použitého uvnitř fontu to Unicode. Díky tomu je možné z dokumentu spolehlivě extrahovat text.
- Dokument musí obsahovat metadata ve formátu XMP a je-li barevný, pak také formální specifikaci barevného prostoru.

Tato šablona používá balíček `pdfx`, který umí LaTeX nastavit tak, aby požadavky PDF/A splňoval. Metadata v XMP se generují automaticky podle informací v souboru `prace.xmpdata` (na vygenerovaný soubor se můžete podívat v `pdfa.xmpi`).

Validitu PDF/A můžete zkontrolovat pomocí nástroje VeraPDF, který je k dispozici na `http://verapdf.org/`.

Pokud soubor nebude validní, mezi obvyklé příčiny patří používání méně obvyklých fontů (které se vkládají pouze v bitmapové podobě a/nebo bez unicodových tabulek) a vkládání obrázků v PDF, které samy o sobě standard PDF/A nesplňují.

Další postřehy o práci s PDF/A najdete na `http://mj.ucw.cz/vyuka/bc/pdfaq.html`.

# Závěr

# Seznam použité literatury


Anděl, J. (1998). *Statistické metody.* Druhé přepracované vydání. Matfyzpress, Praha. ISBN 80-85863-27-8.

Anděl, J. (2007). *Základy matematické statistiky.* Druhé opravené vydání. Matfyzpress, Praha. ISBN 80-7378-001-1.

Cox, D. R. (1972). Regression models and life-tables (with Discussion). *Journal of the Royal Statistical Society, Series B*, **34**(2), 187–220.

Dempster, A. P., Laird, N. M. a Rubin, D. B. (1977). Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society, Series B*, **39**(1), 1–38.

Genberg, B. L., Kulich, M., Kawichai, S., Modiba, P., Chingono, A., Kilonzo, G. P., Richter, L., Pettifor, A., Sweat, M. a Celentano, D. D. (2008). HIV risk behaviors in sub-Saharan Africa and Northern Thailand: Baseline behavioral data from project Accept. *Journal of Acquired Immune Deficiency Syndrome*, **49**, 309–319.

Kaplan, E. L. a Meier, P. (1958). Nonparametric estimation from incomplete observations. *Journal of the American Statistical Association*, **53**(282), 457–481.

Lehmann, E. L. a Casella, G. (1998). *Theory of Point Estimation.* Second Edition. Springer-Verlag, New York. ISBN 0-387-98502-6.

Student (1908). On the probable error of the mean. *Biometrika*, **6**, 1–25.

# Seznam obrázků

# Seznam tabulek

# Seznam použitých zkratek

# A. Přílohy

## A.1 První příloha