Západočeská univerzita v Plzni

Fakulta aplikovaných věd

Katedra kybernetiky

DIPLOMOVÁ PRÁCE

PLZEŇ, 2012 **TOMÁŠ PRONĚK**

PROHLÁŠENÍ

Předkládám tímto k posouzení a obhajobě diplomovou práci zpracovanou na závěr studia na Fakultě aplikovaných věd Západočeské univerzity v Plzni.

Prohlašuji, že jsem diplomovou práci vypracoval samostatně a výhradně s použitím odborné literatury a pramenů, jejichž úplný seznam je její součástí.

V Plzni dne 30. srpna 2012 .....

vlastnoruční podpis

Poděkování

Velmi rád bych touto cestou poděkoval vedoucímu diplomové práce panu Ing. Miroslavu Flídrovi, Ph.D. za vedení práce, poskytnutí cenných rad, hodnotných podnětů a užitečných připomínek při vypracování předkládané diplomové práce.

Také bych rád poděkoval své rodině za podporu během celého studia na vysoké škole.

Tomáš Proněk

Abstrakt

Tato diplomová práce se zabývá návrhem a implementací uživatelského prostředí pro *toolbox NEF*. Cíl práce vychází z reálné potřeby takovéto podpory, jelikož v *toolboxu NEF* je vytvořeno výpočetní jádro implementující metody nelineárního odhadování, které ovšem nemá žádnou pokročilejší podporu pro správu a vizualizaci estimačních experimentů a jejich výsledků. V úvodu se bude práce nedlouho zabývat nelineárním odhadem stavu, která pod tuto práci spadá. Dále práce zahrnuje stručné seznámení s *toolboxem NEF*, kde se rozeberou jeho stěžejní funkce a komponenty, které jsou pro práci důležité. Na několika příkladech bude také ukázáno jak se v *toolboxu NEF* tvoří estimační experiment pro pozdější porovnání s vytvářením estimačních experimentů v nově navrženém a implementovaném uživatelském prostředí. Další kapitoly budou věnovány kompletnímu popisu samotného uživatelského prostředí. Uživatelské rozhraní bude bude implementováno v programovacím jazyce *Java*. Volání programového prostředí *Matlab*, ve kterém je toolbox NEF implementován, z nového užívatelského rozhraní bude zprostředkováno pomocí *java* knihovny *Matlabcontrol*.

Klíčová slova: uživatelské rozhraní, *Java*, *Matlabcontrol*, *NEF toolbox*, *Matlab*, *NEF* komponenta, třída, funkce, náhodná veličina, systém, estimace, vizualizace, experiment, *API*, *XML*, *nefLab*

Abstract

This diploma thesis deals with the design and implementation of the user interface for *NEF toolbox*. The aim of the thesis is based on the real requierements of such support, since there is created computing core in the *NEF toolbox*, that implements the nonlinear estimation methods, which however has no support for advanced management and visualization of estimation experiments and their results. In the introduction the work will deal with the nonlinear state estimation, under which the thesis belongs. The work also includes a brief introduction to the *NEF toolbox*, which break down the key features and components that are important for the work. A few examples will also be shown how there is created an estimation experiment in *NEF toolbox* for the later comparsion with the creation of estimation experiment in newly designed and implemented user interface. Other chapters are devoted to a complete description of the user interface itself. The user interface will be implemented in the *Java* programming language. Calling programming environment *Matlab*, in which the *NEF toolbox* is implemented, of the new user interface will be ensured through the java library *Matlabcontrol*.

Keywords: user interface, *Java*, *Matlabcontrol*, *NEF toolbox*, *Matlab*, *NEF* component, class, function, random variable, system, estimation, visualization, experiment, *API*, *XML*, *nefLab*

Obsah

Kapitola 1

Úvod

Nelineární odhad diskrétních stochastických systémů je rychle se rozvíjející obor, který hraje zásadní roli v mnoha oblastech, jako je zpracování signálu (*GPS*, audiovizuální signály), sledování a určování trajektorií měřených a odhadovaných fyzikálních veličin, satelitní navigace, detekce chyb a diagnostika systémů, řídící systémy (řízení strojů a technologických procesů), zpracování obrazu, komunikace (zpracování řeči), biomedicinské inženýrství, operační výzkum, dopravní systémy, předpověď počasí a tak dále. Představuje součást každého rozhodovacího procesu. Je tedy nasnadě mít možnost sáhnout po nástroji, který může usnadnit vývoj nebo testování nových technik odhadů, stejně tak jako využívat již připravenou kolekci známých technik odhadů.

1.1 Uvedení do problematiky práce

Z problematiky nelineárního odhadování vyvstávají požadavky na implementaci a testování jednotlivých metod. Tímto úkolem se zabývá nástroj *toolbox NEF*, dále již pouze *NEF*. V *NEF* je vytvořeno výpočetní jádro implementující metody nelineárního odhadování, které ovšem nemá žádnou pokročilejší podporu pro správu estimačních experimentů či vizualizaci výsledků estimačních experimentů.

Cílem této práce je právě vytvoření podpůrného uživatelského rozhraní pro *NEF*. NEF je implementován v programovém prostředí *Matlab*[1]. Programové prostředí *Matlab* je vhodné pro implementaci velkého množství složitých matematických funkcí a jejich optimalizaci či stabilitu. Tedy je vhodným programovým prostředím pro implementaci *NEF*. Ovšem pro účel vytvoření samostatné aplikace s podpůrným grafickým rozhraním již není zcela vhodný. Respektive existují jiné vhodnější varianty. Lepší variantou je plnohodnotné podpůrné uživatelské rozhraní vytvořené v jiném vývojovém prostředí, které by mohlo využívat výpočetního jádra *NEF*, konkrétněji jeho komponent, které s tímto jádrem pracují. *NEF* již totiž využívá nový matlabovský systém tříd, který je dostupný od verze 2007b. Tento systém tříd poskytuje podporu k vytváření balíků objektů (tj. v případě této práce komponent *NEF*) pomocí jednotlivých implementovaných tříd. To bylo pár slov k uvedení *NEF*, který bude podrobněji popsán v nadcházející kapitole a vraťme se zpět k uvedení do problematiky vytvoření podpůrného uživatelského rozhraní.

[1] MATLAB je programové prostředí a skriptovací programovací jazyk pro vědeckotechnické numerické výpočty, modelování, návrhy algoritmů, počítačové simulace, analýzu a prezentaci dat, měření a zpracování signálů, návrhy řídicích a komunikačních systémů.

Zopakujme, že úplné znění práce nese název uživatelské prostředí pro toolbox NEF. Toto prostředí bude implementováno v dobře známém programovacím jazyce *Java*. Vzhledem k tomu, že NEF je značně komplexní softwarový nástroj, je potřeba vytvořit dostatečně univerzální nadstavbu popisující *NEF*, kterou by mohlo nově vytvářené uživatelské prostředí převzít a použít. Jinými slovy je nutné popsat nějakým způsobem *API*[2] *NEF*. Takovýto požadavek bude realizován využitím jazyka *XML*, který je pro tento účel vhodný. Popis *API NEF* vznikne implementací značkovacího jazyka *XML*, který bude popisovat jednotlivé implementované třídy *NEF* vstupně výstupním chováním (tj. popisem vstupních a výstupních parametrů konstruktorů tříd a jejich metod). Tato nadstavba nesmí ovšem obsahovat přímo jeho části, tzn. že při návrhu značkovacího jazyka *XML* musí být brán zřetel na budoucí vývoj *NEF*. Takovéto prostředí musí jednoduše reagovat na změny v *NEF*, například při vydání nové verze, se kterou může dojít k úpravě jednotlivých tříd, či přidání zcela nových tříd. Takto popsané *API* bude potom načteno uživatelským rozhraním a na jeho základě se v něm vytvoří odpovídající objekty, které představují komponenty pro návrh estimačních experimentů. Po sestavení estimačního experimentu v uživatelském prostředí je nutno zaslat programovému prostředí *Matlab* hotový nakonfigurovaný *m-script* (tj. popis experimentu), aby provedl samotný výpočet. Toto bude zprostředkováno pomocí *java* knihovny *Matlabcontrol*, která umožňuje volat programové prostředí *Matlab* z *java* aplikací. Knihovna umožňuje aby, interakce probíhala jak z venčí tak zevnitř programového prostředí *Matlab*. Tato práce se zabývá pouze interakcí z venčí programového prostředí *Matlab*.

Dalším cílem pro toto uživatelské rozhraní je návrh a snadné vykonání komplexních estimačních experimentů a jejich následná vizualizace. Toto bude realizováno pomocí výše zmíněných objektů nebo-li výkonných a vizualizačních bloků. Uživatelské prostředí bude dále moci takto vytvořené estimační experimenty spravovat. To zahrnuje ukládání a načítání estimačních experimentů, exportování experimentálních dat, přeložení výsledných estimačních experimentů do zmíněných matlabovských skriptů a analyzování výsledků estimačních experimentů.

Uživatelské rozhraní je vytvářeno za účelem zprostit uživatele od detailní znalosti NEF. Tedy aby nemusel být obeznámen s implementací jednotlivých metod nelineárního odhadování. K tvorbě estimačních experimentů užitím nového rozhraní by měla postačit alespoň základní obeznámenost s programovým prostředím *Matlab*.

[2]Zkratka pro Application Programming Interface. V informatice a programování označuje rozhraní, pomocí kterého lze volat příkazy a funkce programu nebo aplikace zvenčí, většinou z jiné aplikace.

Kapitola 2

Toolbox NEF

Následující kapitola se zejména zaměřuje na popis *toolboxu NEF*, dále opět jen *NEF*[1]. Kapitola obsahuje krátkou teorii k nelineárnímu odhadu stavu, na jejímž základu *NEF* vznikl. Ve dvou subkapitolách se rozeberou jeho stěžejní komponenty a základní funkčnost. Na dvou krátkých příkladech zde bude předveden návrh tvorby estimačních experimentů. Závěr kapitoly shrnuje získané znalosti o *NEF*.

2.1 Úvod

NEF je nově navržený softwarový rámec pro nelineární odhad stavu diskrétních dynamických systémů. Cílem tohoto rámce je usnadnění implementace, testování a používání různých nelineárních metod na odhad stavu. Hlavní síla *NEF* spočívá v jeho přizpůsobivosti. Tedy v možnosti buďto strukturálního nebo pravděpodobnostního popisu problému. Tento nástroj je založen na objektově orientovaném přístupu, což zprošťuje uživatele od detailů, které pro něj nejsou důležité. To ovšem neubírá na jeho možnostech jako je plná kontrola nad experimenty a využití prostředků pro jednoduché rozšíření.

2.2 Nelineární odhad stavu

Práce se nepřímo vztahuje k problematice úlohy estimace stochastických procesů. Úloha se dělí na tři základní typy, těmi jsou filtrace, predikce a vyhlazování. V tomto stručném úvodu si teď úlohu estimace z části přiblížíme. Jedna z etap teorie estimace se zabývá teorií a aproximačním řešením problému nelineární estimace. Estimace stavu nelineárních a negaussovských systémů vede na problém nelineární estimace. Přístupy, které se tímto problémem zabývají můžeme rozdělit na lokální a globální, pokud sledujeme hledisko platnosti získaných výsledků ve stavovém prostoru nebo na analytické a numerické, pokud sledujeme způsob řešení bayesovských rekurzivních vztahů.

Diskrétní nelineární stochastický systém může být popsán neboli modelován dvěma způsoby. Prvním způsobem je pravděpodobnostní modelování (tj. pomocí funkcí podmíněných hustot pravděpodobnosti). Druhým způsobem je pak strukturální modelování (tj. použitím stochastických rovnic). Tyto dvě techniky modelování mohou být v mnoha případech zaměnitelné.

[1] Zkratka pro Nonlinear Estimation Framework.

Při použití pravděpodobnostního modelování je systém zcela popsán podmíněnými hustotami pravděpodobnosti.

$$p(x_{k+1})|x_k, u_k), k \ = 0, 1, \ldots, \tag{2.2.1}$$

$$p(z_k)|x_k, u_k), k \ = 0, 1, \ldots, \tag{2.2.2}$$

kde výraz (1) představuje tzv. přechodovou hustotu pravděpodobnosti a výraz (2) představuje tzv. hustotu pravděpodobnosti měření. $x_k \in \Re^{n_x}$ vyjadřuje neměřitelný stav systému, který je odhadován, $z_k \in \Re^{n_z}$ měření systému, které je známo a $u_k \in \Re^{n_u}$ řízení systému v čase k. Strukturální modelování má pak následující formu

$$x_{k+1} = f_k(x_k, u_k, w_k), k \ = 0, 1, \ldots, \tag{2.2.3}$$

$$z_k = h_k(x_k, u_k, v_k), k \ = 0, 1, \ldots, \tag{2.2.4}$$

kde $w_k \in \Re^{n_x}$ reprezentuje stavový bílý šum a $v_k \in \Re^{n_z}$ reprezentuje bílý šum měření.

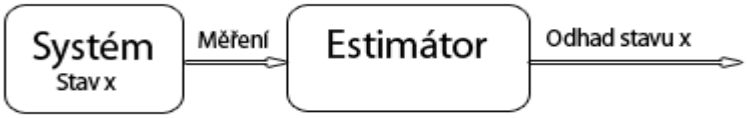
Bílý šum je stochastický proces, který nevykazuje žádné možnosti predikce neboli absolutně nezávislý nekorelovaný proces. Stavový šum nesmí vykazovat žádnou závislost do minulosti a lze jej vyjádřit následujícím vztahem

$$p(w^k) = p(w_0, w_1, \ldots, w_k) = p(w_0).p(w_1) \ldots p(w_k) \tag{2.2.5}$$

Obdobně pak lze vyjádřit vztah pro bílý šum měření

$$p(v^N) = p(v_0, v_1, \ldots, v_N) = p(v_0).p(v_1) \ldots p(v_N) \tag{2.2.6}$$

Dále je požadováno, aby stavový šum, šum měření a apriorní informace byly vzájemně nezávislé. Počáteční stav x_0 se v této pravděpodobnostní interpretaci stává náhodnou veličinou $p(x_0)$.



Obrázek 2.1: Základní blokové schéma popisující úlohu odhadu.

Hlavním cílem úlohy odhadu je najít podmíněnou hustotu pravděpodobnosti $p(x_k|z^l)$ stavu podmíněnou znalostí vektorem měření. Odhad stavu x_k je dán aposteriorní podmíněnou hustotou pravděpodobnosti $p(x_k|z^l, u^l)$, kde z^l je sekvence měření do času l, $z^l \triangleq [z_0^T, z_1^T, \ldots, z_l^T]^T$.

Na základě vztahu mezi l a k lze obecná úloha odhadování dělit na tři základní typy.

- Pokud $l = k$ jedná se o *filtraci*.
- Pokud $l < k$ jedná se o *predikci*.
- Pokud $l > k$ jedná se o *vyhlazování*.

Obecné řešení úlohy odhadu stavu je reprezentováno Bayesovským rekurzivními vztahy, ve kterých se v rekurzi střídá krok predikce a filtrace. Mějme odhad x_{k-1} založený na měření z_{k-1}, řekněme $\hat{x}_{k-1}$. Určeme odhad $\hat{x}_k$ z $\hat{x}_{k-1}$ a z^k. Nyní užitím Bayesova pravidla, které doplníme konvoluční rovnicí, dostaneme Bayesův rekurzivní vztah pro aposteriorní hustotu pravděpodobnosti nebo alternativně filtrační hustotu pravděpodobnosti (2.2.7).

$$p(x_k|z^k) = \frac{p(z_k|x_k).p(x_k|z^{k-1})}{p(z_k|z^{k-1})} \tag{2.2.7}$$

kde $p(z_k|z^{k-1}) = \int p(z_k|x_k)p(x_k|z^{k-1})dx_k$ je normalizační konstanta. A kde predikce $p(x_k|z^{k-1})$ je dána konvolučním vztahem

$$\begin{aligned} p(x_k|z^{k-1}) &= \int p(x^k, x_{k-1}|z^{k-1})dx_{k-1} \\ &= \int p(x_k|x_{k-1}).p(x_{k-1}|z^{k-1})dx_{k-1} \end{aligned} \tag{2.2.8}$$

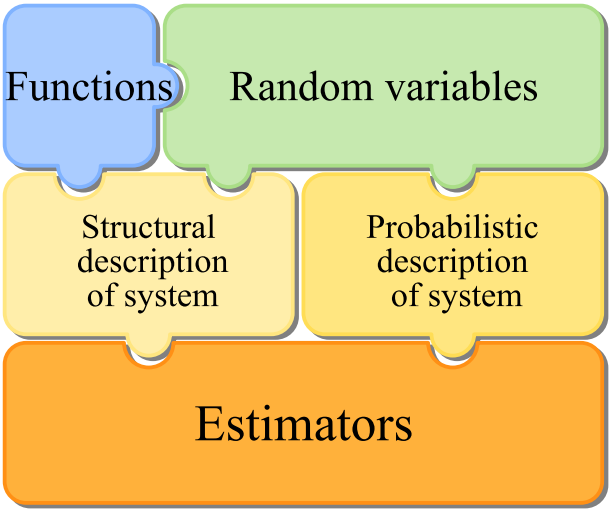
Rekurze (2.2.7) a (2.2.8) může být odstartována aplikací Bayesova pravidla a apriorního popisu x_0, tedy

$$p(x_0|z^0) = \frac{p(z_0|x_0).p(x_0)}{p(z^0)} \tag{2.2.9}$$

Z teoretického hlediska jsou předchozí vztahy úplné řešení problému estimace stavu, protože poskytují úplný pravděpodobnostní popis náhodných stavových veličin ve formě hustot pravděpodobnosti.

2.3 Komponenty estimačního toolboxu

Tato sekce poskytuje základní přehled o vlastních komponentách *NEF*. V terminologii *NEF* znamená komponenta soubor objektů, které slouží k popisu entit určitého typu (např. funkcí, náhodných veličin apod.). Komponenta sestává z tříd, které představují návod jak vytvářet objekty, které už představují nějakou konkrétní entitu, tj. určitý aspekt estimačního experimentu. Přehled se bude postupně zabývat čtyřmi základními komponenty a to sice funkcemi, náhodnými veličinami, systémy a estimátory, viz. obrázek 2.2. První tři komponenty jsou ve skutečnosti vzájemně provázány a slouží především pro náhodné veličiny a popis systému. Poslední z nich pak slouží jako základ pro realizaci různých estimačních technik. A také nabízí již hotovou implementaci mnoha známých estimačních technik.



Obrázek 2.2: Základní blokové schéma popisující úlohu odhadu.

2.3.1 Komponenty pro popis systému

Základní stavební kámen rámce představuje komponenta funkce. Obecně může funkce hrát roli na místech jako je strukturální popis systému, popis časově variantních náhodných veličin nebo reprezentace podmíněných hustot pravděpodobnosti. Parametry funkce jsou pak stav, řízení (vstup), šum a čas. Primárním účelem tříd spojených s komponentou funkce je poskytnout buďto základ pro pro stochastické rovnice, které popisují přechod stavu nebo měření nebo pro popis parametrů náhodné veličiny jakožto hustotu pravděpodobnosti, která je využita pro pravděpodobnostní popis.

Rodičovská třída všech tříd v komponentě funkce je *nefFunction*. Implementuje esenciální vlastnosti funkce a slouží jako šablona pro funkce definované uživatelem. Kromě této rodičovské třídy rámec dále poskytuje následující oděděné třídy funkce.

- konstantní funkce (*nefConstfunction*)
  $f_k(x_k, u_k, w_k) = K$,
- lineární funkce (*nefLinfunction*)
  $f_k(x_k, u_k, w_k) = Fx_k + Gu_k + Hw_k$,
- anonymní (handle) funkce (*nefHandlefunction*)
  libovolné $f_k = (x_k, u_k, w_k)$.
- funkce pro odmocninový rozklad (*nefCholFactorFunction*)
  $f(X, U, XI, T) = P = SS'$
- funkce pro UD rozklad (*nefUDFactorFunction*)
  $f(X, U, XI, T) = K = UDU'$ nebo $f(X, U, XI, T) = UDU' = K$

Pokud je požadováno, objekt (typu *funkce*) vytvořený použitím zmíněných tříd může vyhodnotit funkci v libovolném bodě a také její první a druhou derivaci s ohledem na stav nebo šum.

Široká použitelnost poskytuje především třída *nefHandleFunction*. S touto třídou může uživatel lehce definovat téměř jakoukoli funkci si zamane. Třída využívá tzv. anonymních funkcí. Programové prostředí *Matlab* vytvoří odkaz na lokální funkci uchovanou v paměti, kterou je pak možno volat. Pro více informací o těchto funkcích [8]. Nyní bude na příkladu v rychlosti demonstrována její síla. Uvažujme nelineární funkci

$$f_k(x_k, u_k, w_k) = arctan\frac{x_{2,k}}{x_{1,k}} + sin(2k)w_k \tag{2.3.1}$$

Třída *NefHandlefunction* poskytuje zcela jednoduchou a rychlou možnost jak specifikovat takovouto uvažovanou funkci. Objekt, který bude vyjadřovat tuto funkci může být vytvořen užitím následujícího příkazu

```
f = nefHandleFunction(@(x,u,w,k) atan(x(2)/x(1))+sin(2*k)*w, [2,0,1,1])
```

kde první parametr

```
@(x,u,w,k) atan(x(2)/x(1))+sin(2*k)*w
```

představuje již zmíněnou anonymní funkci a vektor `[2,0,1,1]` určuje dimenzi příslušných proměnných reprezentující stav x, řízení (vstup) u, šum w a čas k.

Komponenta náhodné veličiny také disponuje rodičovskou třídou nazvanou *nefRV*, která poskytuje základní metody pro vlastní popis náhodných veličin. Významnou vlastností nového rámce je fakt, že všechny parametry distribuce náhodné veličiny mohou být specifikovány dvěma způsoby. Buďto jsou uvažovány jako konstanty nebo funkce stavu, vstupu a času. Tedy jsou specifikovány číselně nebo užitím instance jedné z tříd funkcí. Tato vlastnost umožňuje specifikaci náhodných veličin s časově proměnnými parametry, ale také specifikaci podmíněných hustot pravděpodobností, které jsou stěžejní pro pravděpodobnostní popis uvažovaného systému. Co se týče vlastních náhodných veličin, tak aktuální rámec podporuje popis podmíněných hustot pravděpodobnosti těmito třídami

- vícerozměrné jednotné (*nefUniformRV*),
- vícerozměrné Gaussovské (*nefGaussianRV*),
- vícerozměrné Gaussovské součet(*nefGaussianSumRV*),
- vícerozměrné empirické (*nefEmpiricalRV*),
- jednorozměrné Beta (*nefBetaRV*),
- jednorozměrné Gamma (*nefGammaRV*)

Třídy náhodných veličin implementují tyto základní metody

- generování vzorku
- vyhodnocení podmíněných hustot pravděpodobnosti v určitém bodě
- poskytnutí střední hodnoty a rozptylu

- zobrazení průběhu podmíněných hustot pravděpodobnosti

Z konkrétních objektů, které jsou instancemi tříd komponenty funkce a náhodné veličiny se tedy dále parametrizuje objekt vytvořený z komponenty systému. Její rodičovská třída je *nefSystem* a poskytuje návod k definici obecného systému a seznamu metod, které třídy systému musí poskytovat. Tato třída má dvě oddděděné třídy reprezentující strukturální popis (*nefEqsystem* užitím stochastických rovnic) a pravděpodobnostní popis (*nefPDFSystem*). Strukturálně specifikovaný systém vyžaduje dva parametry reprezentující funkce stavu a měření a specifikaci tří náhodných veličin reprezentující šum stavu, měření a počáteční podmínky. Na druhou stranu pravděpodobnostně specifikovaný systém vyžaduje popis stavu pomocí přechodové podmíněné hustoty pravděpodobnosti, měření pomocí podmíněné hustoty pravděpodobnosti měření a počáteční podmínky. Jedinou metodu, kterou systém poskytuje je metoda simulace vlastní trajektorie pro daný časový horizont s ohledem na vstup.

2.3.2 Komponenta estimátoru

Komponenta estimátoru se skládá z rodičovské třídy, která definuje obecný estimátor založený na Bayesovském přístupu a dalších oddděděných tříd implementující různé estimační techniky. Rodičovská třída nese jméno *nefEstimator* a implementuje Bayesovský přístup pro všechny zmíněné základní estimační úlohy jako je predikce, filtrace a vyhlazování. Tot činí implementaci vlastního algoritmu odhadu přímočaré a jednoduché. Proto implementace nových estimačních technik vyžaduje specifikaci metod *timeUpdate* a *measurementUpdate* pro prediktor nebo filtr a pro vyhlazování také metodu *smoothUpdate*. NEF v současnosti poskytuje následující implementaci estimátorů

- Kalmanův filtr, vyhlazování a predikce,
- rozšířený Kalmanův filtr, vyhlazování a predikce,
- sigma bodové Kalmanovo filtry,
  - diferenční Kalmanův filtr (prvního a druhého řádu),
  - unscentovaný Kalmanův filtr,
- částicové filtry
  - Bootstrap filtr,
  - generic částicový filtr,
  - rozšířený simulační filtr.

Lokální filtry mají kromě klasické verze implementovány i numericky stabilní verze jako jsou například UD a odmocninové verze.

2.4 Příklady estimačních experimentů

Tato sekce bude zaměřena na demonstraci jednoduchosti vytváření estimačních experimentů použitím *NEF*. Budou zde nastíněny dva zřetelně odlišné příklady, které jsou převzaty z [3]. První

z nich bude prezentovat odhad stavu systému popsaného stochastickými rovnicemi s použitím unscentového Kalmanova filtru. Druhý příklad pak bude demonstrovat použití particle filtru pro systém popsaný podmíněnými hustotami pravděpodobnosti.

2.4.1 Odhad stavu nelineárního Gaussovského systému užitím UKF

Stav následujícího systému, který je odhadován je popsán vztahy

$$x_{k+1} = \begin{pmatrix} x_{1,k+1} \\ x_{2,k+1} \end{pmatrix} = \begin{pmatrix} x_{2,k} \cdot x_{1,k} \\ x_{2,k} \end{pmatrix} + w_k, \tag{2.4.1}$$

$$z_k = (1,0)x_k + v_k, \tag{2.4.2}$$

kde x_k označuje odhadovaný stav a z_k označuje měření. Stochastický šum w_k a v_k jsou popsány následujícími podmíněnými hustotami pravděpodobnosti

$$p(w_k) = N\left\{w_k; \begin{pmatrix} 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0.5 & 0 \\ 0 & 0.5 \end{pmatrix}\right\}, \tag{2.4.3}$$

$$p(v_k) = N\left\{v_k; 0, 0.01\right\} \tag{2.4.4}$$

a podmíněná hustota pravděpodobnosti počátečního stavu x_0 je

$$p(x_0) = N\left\{x_0; \begin{pmatrix} 10 \\ -0.85 \end{pmatrix}, \begin{pmatrix} 0.1 & 0 \\ 0 & 0.1 \end{pmatrix}\right\}. \tag{2.4.5}$$

Systém, který je dán vztahy (2.4.1)-(2.4.5) se v *NEF* popíše během tří kroků. Zaprvé se vytvoří stochastické funkce užitím instance třídy *nefHandleFunction* a *nefLinFunction*. Vícerozměrná nelineární funkce reprezentující stavovou rovnici je specifikována zavedením příkazu

```
fFun = @(x,u,w,k)[x(1)*x(2)+w(1);x(2)+w(2)]
f = nefHandleFunction(fFun, [2 0 2 0], 'diff1Noise', @(x,u,w,k) eye(2));
```

kde první parametr konstruktoru třídy *nefHandleFunction* je zmíněná anonymní funkce. Druhý parametr specifikuje dimenzi stavu, řízení, stavového šumu a času. Poslední dva parametry pak udávají první derivaci nelineární funkce se zřetelem na šum w_k. Rovnice měření je lineární a rámec poskytuje dva způsoby k reprezentaci takové funkce. První cesta je opětovné užití třídy *nefHandleFunction*, tj.

```
H = [1 0];
hFun = @(x,u,v,k) H(1)*x(1)+H(2)*x(2)+v;
h = nefHandleFunction(hFun, [2 0 1 0], 'diff1Noise', @(x,u,v,k) 1);
```

druhou cestou je pak možnost vytvoření instance třídy *nefLinFunction*

```
H = [1 0];
h = nefLinFunction(H, [], 1);
```

kde druhý a třetí parametr říká, že rovnice nebude obsahovat žádné řízení a že nebude přítomný šum.

Dalším krokem je pak popis náhodných veličin w_k, v_k a x_0. Každá z nich bude uvažována jako gaussovská a proto budou popsány instancemi tříd *nefGaussianRV*

```
w = nefGaussianRV([0 0]', eye(2)*0.05);
v≠ nefGaussianRV(0, 0.01);
x0 = nefGaussianRV([0.9; −0.85], 1e−1*eye(2));
```

Po tomto kroku je již vše připraveno k definici systému a jeho vlastní simulaci trajektorie. Jelikož je systém popsán stochastickými rovnicemi, je nutno využít třídu *nefEqsystem*

```
system = nefEqSystem(f, h, w, v, x0);
```

Simulace je pak provedena zavedením metody *simulate* příslušící objektu typu *system*. Systém bude simulován na časovém horizontu $k = 1, 2, \ldots, 20$ zavedením následujícími příkazy

```
nSteps = 20;
[z,x] = simulate(system, nSteps, []);
```

kde druhý parametr určuje počet časových okamžiků, na kterých bude provedena simulace trajektorie. Poslední parametr může být využit jako vstup řízení.

Konečně lze přejít k samotnému odhadu stavu. K tomuto příkladu byl vybrán unscentovaný Kalmanův filtr. Objekt, který popisuje estimátor je vytvořen zavedením příkazu

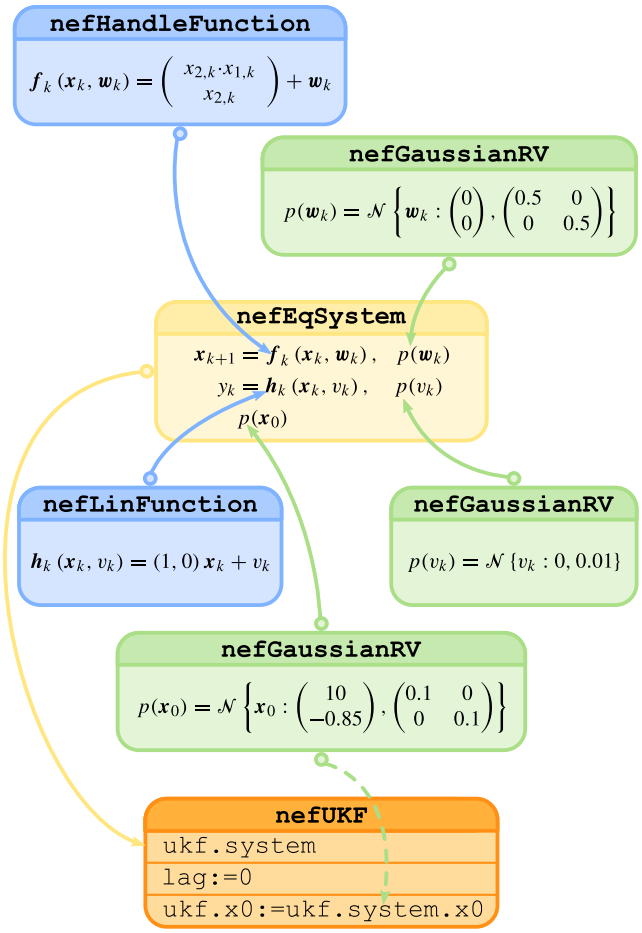
```
UKF = nefUKF(system);
```

jelikož zde není uvedený volitelný parametr *'taskType'* (volba typu úlohy odhadu) je použita jeho výchozí hodnota, tj. *'filtering'*. Rozdíl mezi časovými okamžiky k a l je počátečně nastaven na nulu, tj. v tomto případě hledán filtrační odhad $p(x_k|z^k)$.

Výsledný odhad je pak obdržen zavedením příkazu, který volá metodu *estimate* třídy *nefEstimator* na konkrétní estimátor.

```
estimates = estimate(UKF, z, []);
```

Celý estimační experiment je přehledně ilustrován na obrázku 2.3.



Obrázek 2.3: Komponenty estimačního experimentu příkladu 2.4.1.

2.4.2 Filtrace nelineárního gaussovského systému

Následující příklad představuje problém sledování trajektorie objektu. Uvažujme loď, která se pohybuje v souřadnicové rovině x-y se stacionárním měřícím zařízením mimo loď v počátku souřadnicové roviny. Předpokládá se, že loď bude zrychlovat a zpomalovat čistě náhodně vzhledem k času. Stav x_k je dán jako $x_k = (x_k, v_{x_k}, y_k, v_{y_k})^T$, kde x_k, y_k reprezentuje pozici lodě a v_{x_k},v_{y_k} jsou korespondující rychlosti. Dynamika stavu je popsána následující přechodovou podmíněnou hustotou pravděpodobnosti

$$p(x_{k+1}|x_k) = N\left\{x_{k+1}; \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix} x_k, 0.0001 I_4\right\}, \tag{2.4.6}$$

kde I_4 je $4x4$ jednotková matice. Měření modelu je směr, který je dán $tan^{-1}(\frac{y_k}{x_k})$ a odpovídající podmíněná hustota pravděpodobnosti měření je dána vztahem

$$p(z_k|x_k) = N\left\{z_k; tan^{-1}(\frac{y_k}{x_k}), 0.0001\right\}. \tag{2.4.7}$$

K specifikaci systému využijeme třídy *nefPDFSystem*. Přechodová hustota pravděpodobnosti a hustota pravděpodobnosti měření jsou obě podmíněnými hustotami pravděpodobnosti, které je možné vyjádřit tak, že jejich střední hodnota je dána přesně danou funkcí a jejich rozptyl pomocí kovarianční matice takto

```
F = [1 1 0 0; 0 1 0 0; 0 0 1 1; 0 0 0 1];
xMean = nefLinFunction(F, [], []);
xVariance = 0.0001*eye(4);
xPdf = nefGaussianRV(xMean, xVariance);
```

Obdobně tedy i pro hustotu pravděpodobnosti měření vyjádříme její střední hodnotu přesně danou funkcí a její rozptyl pomocí kovarianční matice následovně

```
mFun = @(x,u,v,t) atan(x(3)/x(1));
zMean = nefHandleFunction(mFun, [4 0 0 0]);
zVariance = 0.0001;
zPdf = nefGaussianRV(zMean, zVariance);
```

Počáteční podmínka je specifikována jako

```
x0Pdf = nefGaussianRV([-0.05 0.001 2 -0.055]', 0.01*eye(4));
```

Systém pak vytvoříme pomocí pravděpodobnostního popisu

```
system = nefPDFSystem(xPdf, zPdf, x0Pdf);
```

je-li potřeba provést simulaci trajektorie tohoto systému, která poskytuje posloupnost stavů systému a měření, tak ji lze provést vykonáním následujících příkazů

```
nSteps = 20;
[z,x] = simulate(system, nSteps, []);
```

kde hodnoty měření jsou uloženy v proměnné z a hodnoty stavu v proměnné x. Na základě připravených dat ze simulace pak můžeme provést estimační experiment. Pomocí posloupnosti měření v proměnné z docílíme odhadu stavu, užitím globálního filtru jediným příkazem. Výsledný odhad stavu pak můžeme porovnat se stavem, který je poskytnut simulací. Příkaz je specifikován popisem systému *system*, jelikož zde opět není uvedený volitelný parametr *'taskType'* (volba typu úlohy odhadu) je použita jeho výchozí hodnota, tj. *'filtering'*. Rozdíl mezi časovými okamžiky k a l je počátečně nastaven na nulu a vzorkovací hustota je dána parametrem *sampling density*.

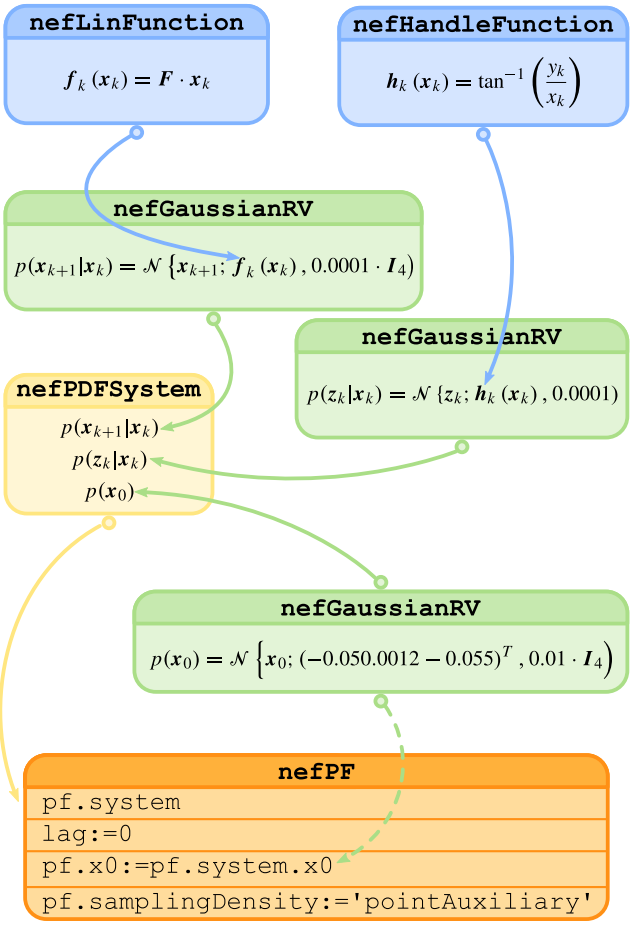
```
pfEstimator = nefPF(system, 'samplingDensity', 'pointAuxiliary');}
```

Pokud bychom nepožadovali úlohu filtrace, ale například predikce. Použili bychom parametr *'taskType'* s hodnotou *'prediction'*. Rozdíl mezi časovými okamžiky bychom pak mohli ovlivnit pomocí parametru *'taskPar'*. Poznamenejme, že nebylo použito dalších parametrů souvisejících s globálním filtrem, tudíž se pro ně použily výchozí hodnoty. Vlastní estimace je zavedena takto

```
estimates = estimate(pfEstimator, z, []);
```

Výsledné hodnoty odhadu filtrační hustoty pravděpodobnosti $p(x_k|z^k)$, která je dána empirickou hustotou pravděpodobnosti *nefEmpiricalRV*, jsou uloženy v *estimates*.

Obrázek 2.4 pak přehledně ilustruje celý estimační experiment.



Obrázek 2.4: Komponenty estimačního experimentu příkladu 2.4.2.

2.5 Shrnutí

V této kapitole jsme se seznámily se stěžejními komponenty a funkčností *NEF*, který usnadňuje implementaci, testování a používání metod nelineárního odhadu. Rámec je určen jak pro znalé uživatele tak i pro uživatele povrchních znalostí. Jeho používání je jednoduché a účinné. Rámec se skládá ze čtyř komponent. Dva z nich jsou cílené pro popis diskrétních časově proměnných funkcí a časově proměnných náhodných veličin. Třetí komponenta umožňuje popis systému buďto pravděpodobnostně nebo strukturálně. Čtvrtá komponenta implementuje různorodé nelineární estimační metody a především poskytuje jednoduché prostředky pro další

rozšíření rámce. Rozšíření je zcela v rukou uživatele, kdy má možnost specifikovat své nové metody odhadu.

Na příkladech bylo demonstrováno jak je lehké vytvořit estimační experiment. Experimenty mohou být navrženy co nejjednodušeji, přirozeně a efektivně.

Kapitola 3

Popis API NEF a experimentu pomocí XML

Jak již bylo nastíněno v úvodní kapitole popis *API NEF* představuje důležitou esenci celého programu. Popis *API NEF* a popis experimentu bude realizován pomocí jazyka *XML*. Ještě než se přistoupí k samotnému rozboru popisu *API NEF* a popisu experimentu bude třeba se seznámit se základem jazyka *XML*, především s tvorbou *XML* dokumentu. Poté se bude kapitola věnovat právě základnímu přístupu k popisu *API NEF* a popisu experimentu.

3.1 Jazyk XML

Jazyk *XML* spadá mezi tzv. značkovací jazyky. Jazyk je určen především pro výměnu dat mezi aplikacemi a pro publikování dokumentů, u kterých popisuje strukturu z hlediska věcného obsahu jednotlivých částí, nezabývá se vzhledem. Prezentace dokumentu může být definována pomocí kaskádových stylů. Další možností zpracování je transformace do jiného typu dokumentu, nebo do jiné aplikace *XML*.

3.1.1 Vlastnosti XML

Není vhodné zasílat dokumenty ve tvaru, který vyžaduje pro zpracování speciální software konkrétní firmy, jako je např. formát *DOC*, *XLS* nebo *T602*. Je používána celá řada operačních a informačních systémů a není možné předpokládat, že každý uživatel vlastní příslušný software. Vznikla tak potřeba vytvořit nějaký jednoduchý otevřený formát, který není úzce svázán s nějakou platformou nebo proprietární technologií. Tím může být právě *XML*, který je založen na jednoduchém textu a je zpracovatelný (v případě potřeby) libovolným textovým editorem. Specifikace *XML* konsorcia *W3C* je zdarma přístupná všem. Každý tak může bez problémů do svých aplikací implementovat podporu *XML*. To je velký rozdíl oproti firemním formátům, k nimž není k dispozici žádná dokumentace a navíc se jedná v porovnání s *XML* o velice složité, často binární, formáty.

XML hned od samého počátku myslel na potřeby i jiných jazyků než je angličtina. Jako znaková sada se implicitně používá *ISO 10646* (také *Unicode*). V *XML* proto můžeme vytvářet dokumenty, které obsahují texty v mnoha jazycích najednou. Současně je přípustné i jiné libovolné kódování (např. *windows-1250*, *ISO 88592*), musí však být v každém dokumentu přesně

určeno. Odpadají tak problémy s konverzí z jednoho kódování do druhého.

Pomocí *XML* značek (tagů) vyznačujeme v dokumentu význam jednotlivých částí textu. Dokumenty tak obsahují více informací, než kdyby se používalo značkovaní zaměřené na prezentaci. *XML* dokumenty jsou informačně bohatší. To lze samozřejmě s výhodou využít v mnoha oblastech. Největší přínos bude samozřejmě pro prohledávání, kdy můžeme určit i jaký význam má mít hledaný text.

XML neobsahuje předdefinované značky (tagy), je třeba definovat vlastní značky, které budeme používat. Tyto značky je možné (nepovinně) definovat v souboru *DTD* (*Document Type Definition*). Potom je možné automaticky kontrolovat, zda vytvářený *XML* dokument odpovídá této definici. Program, který tyto kontroly provádí, se nazývá parser. Při vývoji aplikací můžeme parser použít, a ten za nás detekuje většinu chyb v datech. *DTD* není jediný definiční jazyk pro *XML*. Neobsahuje možnost kontrolovat typy dat. To je vlastnost, která chybí při zpracování dat databázového charakteru. V současné době se pod názvem *XML* schémata pracuje na půdě konsorcia *W3C* na vytvoření jednotného standardu, který tyto kontroly umožní. Další vlastností *XML* je, že v jednom dokumentu můžeme používat najednou nezávisle na sobě několik druhů značkovaní pomocí jmenných prostorů (*namespaces*). To umožňuje kombinovat v jednom dokumentu několik různých definic ve formě *DTD* nebo schémat bez konfliktů v pojmenování elementů.

XML stejně jako *HTML* umožňuje vytváření odkazů v rámci jednoho dokumentu i mezi dokumenty, má však více možností. Je možné vytvářet i vícesměrné odkazy, které spojují více dokumentů dohromady.

3.1.2 Syntaxe XML

XML dokument je text, vždy *Unicode*, v Česku obvykle kódovaný jako *UTF-8*, ale jsou přípustná i jiná kódování. Na rozdíl od např. *HTML*, efektivita *XML* je silně závislá na struktuře, obsahu a integritě. Aby byl dokument považován za správně strukturovaný (*well-formed*), musí mít splňovat následující vlastnosti

- Musí mít právě jeden kořenový (root) element.
- Neprázdné elementy musí být ohraničeny startovací a ukončovací značkou. Prázdné elementy mohou být označeny tagem „prázdný element“.
- Všechny hodnoty atributů musí být uzavřeny v uvozovkách, buďto jednoduchých (') nebo dvojitých ("), ale jednoduchá uvozovka musí být uzavřena jednoduchou a dvojitá dvojitou. Opačný pár uvozovek může být použit uvnitř hodnot.
- Elementy mohou být vnořeny, ale nemohou se překrývat. To znamená, že každý (ne kořenový) element musí být celý obsažen v jiném elementu.
- Je potřeba dbát na jména elementů v *XML*, jelikož rozlišují malá a velká písmena.

3.1.3 Tvorba XML dokumentu

XML nám umožňuje definovat vlastní sadu značek (tagů), které chceme v dokumentu používat. Dále nám tento jazyk nabízí možnost tvorby odkazů v rámci dokumentu i mezi dokumenty

navzájem. Každý *XML* dokument se skládá z elementů, které jsou do sebe navzájem vnořené. Elementy se v textu vyznačují pomocí počátečních a ukončovacích tagů. Nejprve se musí v dokumentu deklarovat znaková sada a její kódování. Znaková sada určuje množství a symboliku znaků pod určitými čísly, které můžeme použít v dokumentu. Kódování znakové sady určuje, jak jsou jednotlivé kódy znaků převedeny na sekvenci bajtů. Dále je možno specifikovat *DTD*, což představuje definici typu dokumentu. *DTD* určuje jaké elementy a atributy , se můžou v dokumentu použít. Navíc je zde definováno, v jakých vzájemných vztazích mohou být jednotlivé elementy použity. *DTD* je nástroj, který hlídá správnou strukturu našeho dokumentu. Pokud chceme využít výhody *DTD* musíme jej definovat pomocí *doctype* v hlavičce dokumentu.

3.1.4 Zpracování XML v Javě

Existují dva nejčastější přístupy ke zpracování *XML* dokumentu

- *DOM* parser (*DOM* je zkratka pro *Document Object Model*) načte celý *XML* dokument do paměti a vytvoří tam stromovou objektovou reprezentaci.
- *SAX* parser (*SAX* je zkratka pro *Simple API for XML*) postupně prochází *XML* dokument a vyvolává události. Je na programátorovi, aby tyto události zpracoval.

Pro účely této diplomové práce byl zvolen parser *DOM*, jelikož jeho souhrn uvedených vlastností je pro účel této práce vhodný a plně dostačující. Parser *DOM* je standardem consorcia *W3C*. Objektům stromové struktury se říká nody nebo-li uzly. Na rozdíl od *SAX* je *DOM* vhodný i pro změnu nebo vytváření nových *XML* dokumentů. Disponuje také možností zápisu a nastavování. Zcela ve stejné filosofii jako u *SAX* je *DOM* odstíněn přes *JAXP*[1]. V *org.w3c.dom* jsou rozhraní jednotlivých nodů. Tyto rozhraní jsou pro další postup nezbytné, jelikož je zde popsáno celkem čtrnáct typů nodů, z nichž běžně používaných je pět.

- *Node* je základní prvek a předek s potomky.
  - *Document* představuje počáteční nod (neplést s kořenovým elementem).
  - *Attr* jsou atributy.
  - *Element* jsou elementy.
  - *Text* jsou hodnoty elementů.

Z těchto nodů je pak vytvořen objektový stromový model. V *java.xml.parsers* jsou třídy zajišťující vlastní parsování *DocumentBuilder* a *DocumentBuilderFactory*, které jsou v této práci použity.

[1] *JAXP* je zkratka pro *Java API for XML Processing*. Představuje rozhraní pro zpracování *XML* dokumentů. Podporuje *DOM*, *SAX* a *XSLT* transformace.

3.2 Popis API NEF pomocí XML

Po stručném úvodu do značkovacího jazyka *XML* se nyní přesuneme k jeho samotnému použití k popisu *API NEF*. Začneme strukturou *XML* souborů. Vzhledem k přehlednosti a orientaci ve výsledných *XML* souborech byl popis *API NEF* rozdělen do čtyř *XML* souborů, které odpovídají jednotlivým typům komponent *NEF*. Struktura vypadá následovně

- `nefAPI-functions.xml` (popis funkcí *NEF*),
- `nefAPI-rv.xml` (popis náhodných veličin *NEF*),
- `nefAPI-systems.xml` (popis systémů *NEF*),
- `nefAPI-estimators.xml` (popis estimátorů *NEF*).

Tyto vytvořené soubory tedy zahrnují kompletní popis *API NEF*. Jsou uložené v adresáři `nefapi`, která je součástí adresářové knihovny `lib`.

3.2.1 Výčet vytvořených značek pro popis API NEF

Tato sekce poskytuje kompletní přehled vytvořených značek za účelem popsat *API NEF*. Tyto značky figurují v každém z vytvořených *XML* souborů.

Přehled vytvořených elementů k popisu *API NEF*		
Element	**Hodnota**	**Popis**
`<nefapi>`	vnořené elementy	kořenový element
`<component>`	vnořené elementy	začátek popisu konkrétní komponenty
`<class>`	vnořené elementy	začátek popisu nové třídy
`<methods>`	vnořené elementy	zastřešuje metody konkrétní třídy
`<constructor>`	vnořené elementy	začátek popisu konstruktoru třídy
`<method>`	vnořené elementy	popis konkrétní metody
`<parameters>`	vnořené elementy	zastřešuje vstupní parametry
`<parameter>`	název vstupního parametru	konkrétní popis vstupního parametru
`<results>`	vnořené elementy	zastřešuje výstupní parametry
`<result>`	název výstupního parametru	konkrétní popis výstupního parametru

Tabulka 3.1: Přehled vytvořených elementů k popisu *API NEF*.

Přehled vytvořených atributů k popisu *API NEF*			
Atribut	**Hodnota (příklad)**	**Vazba na element**	**Popis**
`version`	`"1.2.0"`	`<nefapi>`	verze *API NEF*
`type`	`"functions"`	`<component>`	udává typ komponenty
`type`	`"matrix"`	`<parameter>` nebo `<result>`	udává typ dat vstupního/výstupního parametru
`type`	`"value"`	`<parameter>`	udává, že parametr je hodnotou předchozího parametru
`name`	`"nefFunction"`	`<class>`	jméno třídy
`name`	`"evaluate"`	`<method>`	jméno metody
`extends`	`"nefFunction"`	`<class>`	dědičnost třídy
`required`	`"true"`	`<parameter>` nebo `<result>`	povinnost parametru
`additionalParRequired`	`"true"`	`<parameter>`	určuje nutnost definice dalšího parametru, který specifikuje hodnotu předchozího
`editable`	`"false"`	`<parameter>`	možnost specifikace dalších hodnot uživatelem kromě aktuálních hodnot parametru
`input`	`"1"`	`<parameter>`	interní aplikační informace pro specifikaci vstupních pinů bloku
`output`	`"1"`	`<result>`	interní aplikační informace pro specifikaci výstupních pinů bloku
`description`	`"stateNoise"`	`<parameter>` nebo `<result>`	popisek pro parametr

Tabulka 3.2: Přehled vytvořených atributů k popisu *API NEF*.

3.2.2 Příklady popisu funkcí NEF

Soubor `nefAPI-functions.xml` popisuje kompletně všechny komponenty funkcí vstupně výstupním chováním. Tedy zohledňuje všechny vstupní a výstupní parametry konstruktorů tříd a jejich metod. Uvedeme si několik příkladů, na kterých bude podrobně ukázán návrh *XML* syntaxe.

Popis třídy nefFunction

Prvním z nich bude rodičovská třída *nefFunction*, jejíž konstruktor neobsahuje žádné volitelné ani vstupní parametry. Pouze jeden výstupní parametr konstruktoru. Na ukázku byla vybrána metoda *evaluate*. Bohužel zde není možné uvést úplně všechny metody, jelikož každá třída disponuje mnoha metodami.

Výpis 3.1: Hlavička konstruktoru třídy *nefFunction* v *NEF*.

```
function [obj] = nefFunction()
```

Výpis 3.2: Hlavička metody *evaluate* třídy *nefFunction* v *NEF*.

```
function [val] = evaluate(obj, State, Input, Noise, Time)
```

Výpis 3.3: Popis rodičovské třídy *nefFunction* v `nefAPI-functions.xml`.

```
<xml version="1.0"encoding="UTF-8">
<nefapi version="1.2.0">
 <component type="functions">
  <class name="nefFunction" extends="none">
   <methods>
    <constructor>
     <results>
      <result required="true" type="nefFunction">obj</result>
     </results>
    </constructor>
    <method name="evaluate">
     <parameters>
      <parameter required="true" type="nefFunction">obj</parameter>
      <parameter required="true" type="matrix">State</parameter>
      <parameter required="true" type="matrix">Input</parameter>
      <parameter required="true" type="matrix">Noise</parameter>
      <parameter required="true" type="vector">Time</parameter>
     </parameters>
     <results>
      <result required="true" type="matrix">val</result>
     </results>
    </method>
    <!-- : -->
   </methods>
  </class>
 </component>
</nefapi>
```

Kořenovým elementem je `<nefapi>` s atributem `version` a hodnotou atributu `1.2.0`. Poznamenejme, že verze *API NEF* je aplikací *nefLab* při načtení všech souborů popisující *API NEF* kontrolována. Při rozdílných verzích aplikace zobrazí chybu související s načtením *API NEF* a aplikace se ukončí. Hodnotou kořenového elementu jsou další vnořené elementy. To platí pro všechny další takto hierarchicky vnořené elementy a tato informace již nebude dále v textu znovu zmiňována.

```
<nefapi version="1.2.0">..</nefapi>
```

Poté následuje vnořený element `<component>` s atributem `type`, jehož hodnota udává jaký typ komponent *NEF* je právě popisován. V tomto případě tedy *functions*.

```
<component type="functions">..</component>
```

Další vnořený element `<class>` vyznačuje začátek popisu nové třídy. Obsahuje dva atributy `name` a `extends`. Jak názvy napovídají jejich hodnoty udávají jméno třídy tedy `nefFunction` a dědičnost je u rodičovské třídy `none`.

```
<class name="nefFunction" extends="none">..</class>
```

Element `<methods>` oznamuje začátek fáze deklarace metod, tedy i konstruktoru. Neobsahuje žádný atribut.

```
<methods>..</methods>
```

Konstruktor třídy je v *XML* vyznačen elementem `<constructor>`, který je vnořen do elementu `<methods>`. Konstruktor třídy *nefFunction* má pouze jeden výstupní parametr `obj`. Atribut `required="true"` udává, že parametr je nutný a atribut `type="nefFunction"` udává jakého typu bude výsledný objekt. Toto vše je v *XML* popsáno takto

```
<constructor>
 <results>
  <result required="true" type="nefFunction">obj</result>
 </results>
</constructor>
```

Metoda *evaluate* je pak popsána následujícím způsobem. Atribut `name="evaluate"` elementu `<method>` poskytuje jméno metody. Element `<parameters>` vyznačuje začátek vstupních parametrů metody. Element `<parameter>` pak představuje konkrétní parametr. Hodnota elementu je název parametru. A podobně jako u konstruktoru `required="true"` udává, že parametr je nutný a atribut `type` udává jakého typu bude výsledný objekt.

```
<method name="evaluate">
 <parameters>
  <parameter required="true" type="nefFunction">obj</parameter>
  <parameter required="true" type="matrix">State</parameter>
  <parameter required="true" type="matrix">Input</parameter>
  <parameter required="true" type="matrix">Noise</parameter>
  <parameter required="true" type="vector">Time</parameter>
 </parameters>
 <results>
  <result required="true" type="matrix">val</result>
   </results>
</method>
```

Poznamenejme, že značka $<!--:--->$ vyjadřuje v *XML* komentář. Dvojtečka v komentáři pak reprezentuje pokračování kódu nebo-li že nebyl uveden celý kód *XML*.

Popis třídy nefLinFunction

Dále bude jako příklad uveden popis třídy *nefLinFunction*, kde se objevují ještě poměrně jednoduché volitelné parametry v konstruktoru třídy. Popis metod třídy je syntakticky podobný popisu metody *evaluate* z minulého příkladu a nebude zde již uveden.

Výpis 3.4: Hlavička konstruktoru třídy *nefLinFunction* v *NEF*.

```
function [obj] = nefLinFunction(f,g,h,varargin)
    p = inputParser;
    p.FunctionName = 'NEFLINEFUNCTION';
    p.addRequired('f',@(x)isnumeric(x) || isempty(x));
    p.addRequired('g',@(x)isnumeric(x) || isempty(x));
    p.addRequired('h',@(x)isnumeric(x) || isempty(x));
    p.addParamValue('check',1,@(x)x==0 || x==1);
    p.parse(f,g,h, varargin{:});
    obj.check = p.Results.check;
  :
```

Výpis 3.5: Popis třídy *nefLinFunction* v `nefAPI-functions.xml`.

```
<!-- : -->
<class name="nefLinFunction" extends="nefFunction">
 <methods>
  <constructor>
   <parameters>
    <parameter required="true" type="matrix">f</parameter>
    <parameter required="true" type="matrix">g</parameter>
    <parameter required="true" type="matrix">h</parameter>
    <parameter required="false">varargin</parameter>
    <parameter required="false" additionalParRequired="true">'check'</parameter>
    <parameter type="value" editable="false">1%0</parameter>
   </parameters>
   <results>
    <result required="true" type="nefLinFunction" output="1">obj</result>
   </results>
  </constructor>
  <!-- : -->
 </methods>
</class>
```

Element `<class>` tedy vyznačuje začátek popisu nové třídy. Opět obsahuje dva atributy `name` a `extends`. První hodnota atributu udává jméno třídy tedy `nefLinFunction` a druhá hodnota udává jakou třídu rozšiřuje nebo-li jeho rodičovskou třídu tedy `nefFunction`.

```
<class name="nefLinFunction" extends="nefFunction">..</class>
```

Výraznější změnou oproti příkladu *nefFunction* je popis volitelných parametrů konstruktoru třídy. Element `<parameter>` s hodnotou `varargin` popisuje matlabovský zápis volitelných parametrů, kdy se na místo tohoto parametru vloží právě volitelné parametry jsou-li požadovány. Popis volitelných parametrů bude podrobněji rozebrán v dalším příkladu pro třídu *nefHandleFunction*.

```
<parameter required="false">varargin</parameter>
```

Další element `<parametr>` s hodnotou `'check'` představuje jméno volitelného parametru. Obsahuje atribut `required="false"`, který říká že není povinný a atribut `additionalParRequired="true"`, který říká že vyžaduje další parametr, který bude jeho hodnotou.

```
<parameter required="false" additionalParRequired="true">'check'</parameter>
```

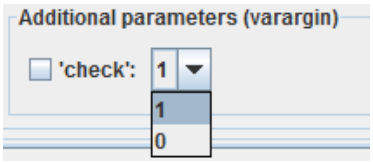
Tuto hodnotu představuje další element `<parametr>` s hodnotou `1%0`. Znak `%` je aplikací při parsování rozpoznán a rozdělí případný řetezec hodnoty na dvě části. Atribut `type="value"` nám říká, že parametr je hodnotou předchozího parametru. A atribut `editable="false"` představuje pro aplikaci informaci zda-li může uživatel vybírat pouze z těchto hodnot či přidat své vlastní.

```
<parameter type="value" editable="false">1%0</parameter>
```

Obrázek 3.1 ilustruje výsledek této syntaxe v aplikaci *nefLab*.

Ukázka popisu třídy nefHandleFunction

Výpis 3.6 představuje jak vypadá v *NEF* hlavička konstruktoru třídy *nefHandleFunction*. Konstruktor obsahuje dva povinné vstupní parametry *fun* a *sizes*. Povinné parametry jsou specifikovány příkazem matlabovského parseru `p.addRequired`. Argumenty tohoto příkazu udávají jakých hodnot může tento parametr nabývat. Tedy, že například parametr *fun* bude anonymní funkce a tato funkce bude obsahovat čtyři vstupní argumenty. Dále konstruktor nabízí možnost



Obrázek 3.1: *Checkbox* a jeho hodnoty načtené z popisu *API NEF*.

specifikovat volitelné parametry. Tyto parametry jsou specifikovány příkazem `p.addParamValue`. Volitelné parametry jsou speciální v tom, že se skládají z názvu parametru a hodnoty parametru. Například volitelný parametr s názvem *'Diff1State'* může nabývat hodnot, které budou opět anonymní funkce se čtyřmi vstupními argumenty. Pro názornost bude ještě uveden příklad vytvoření objektu *nefHandleFunction* v programovém prostředí *Matlab*.

```
f = nefHandleFunction(@(x,u,w,k) [x(1)^2*x(2)+w(1);x(2)+w(2)],[2 0 2 0],...
'diff1Noise',@(x,u,v,k) eye(2),'diff1State',@(x,u,w,k) [2*x(1)*x(2) x(1)^2;0 1]);
```

Výpis 3.6: Hlavička konstruktoru třídy *nefHandleFunction* v *NEF*.

```
function [obj] = nefHandleFunction(fun,sizes,varargin)
 p = inputParser;
 p.FunctionName = 'NEFHANDLEFUNCTION';
 p.addRequired('fun',@(x) isa(x, 'function_handle') && nargin(x) == 4);
 p.addRequired('sizes',@(x) isvector(x) && length(x) == 4 &&
 all(x≥0) && all(mod(x,1)==0));
 p.addParamValue('Diff1State',[],@(x) isa(x, 'function_handle') && nargin(x)==4);
 p.addParamValue('Diff2State',[],@(x) isa(x, 'function_handle') && nargin(x)==4);
 p.addParamValue('Diff1Noise',[],@(x) isa(x, 'function_handle') && nargin(x)==4);
 p.addParamValue('Diff2Noise',[],@(x) isa(x, 'function_handle') && nargin(x)==4);
 p.addParamValue('isAdditive',0,@(x) isnumeric(x));
 p.addParamValue('isLinear',0,@(x) isnumeric(x));
 p.addParamValue('check',1,@(x)x==0 || x==1);
 p.parse(fun,sizes, varargin{:});
 obj.check = p.Results.check;
 :
```

Výpis 3.7 pak demonstruje popis konstruktoru třídy *nefHandleFunction* pomocí *XML*. Syntaxe *XML* je opět stejná jako v předchozích příkladech. Poukažme ovšem na zápis volitelných parametrů.

Výpis 3.7: Popis třídy *nefHandleFunction* v `nefAPI-functions.xml`.

```
<!-- : -->
<class name="nefHandleFunction" extends="nefFunction">
 <methods>
  <constructor>
   <parameters>
    <parameter required="true" type="function_handle">fun</parameter>
    <parameter required="true" type="vector">sizes</parameter>
    <parameter required="false">varargin</parameter>
    <parameter required="false" additionalParRequired="true">'Diff1State'
    </parameter>
    <parameter type="value" editable="true">@(x,u,w,t) [function_handle]
    </parameter>
    <parameter required="false" additionalParRequired="true">'Diff2State'
    </parameter>
    <parameter type="value" editable="true">@(x,u,w,t) [function_handle]
    </parameter>
    <parameter required="false" additionalParRequired="true">'Diff1Noise'
    </parameter>
    <parameter type="value" editable="true">@(x,u,w,t) [function_handle]
    </parameter>
    <parameter required="false" additionalParRequired="true">'Diff2Noise'
    </parameter>
    <parameter type="value" editable="true">@(x,u,w,t) [function_handle]
    </parameter>
    <parameter required="false" additionalParRequired="true">'isAdditive'
    </parameter>
    <parameter type="value" editable="true">isnumeric</parameter>
    <parameter required="false" additionalParRequired="true">'isLinear'
    </parameter>
    <parameter type="value" editable="true">isnumeric</parameter>
    <parameter required="false" additionalParRequired="true">'check'</parameter>
    <parameter type="value" editable="false">1%0</parameter>
   </parameters>
   <results>
    <result required="true" type="nefHandleFunction" output="1">obj</result>
   </results>
  </constructor>
  <!-- : -->
 </methods>
</class>
```

Element `<parameter>` s hodnotou `varargin`,

```
<parameter required="false">varargin</parameter>
```

specifikuje v syntaxi *XML* začátek popisu volitelných parametrů. A také je použit v aplikaci k sestavení popisu hlavičky konkrétní komponenty *NEF*, tj. v tomto případě

```
function [obj] = nefHandleFunction(fun,sizes,varargin)
```

Po tomto elementu následují elementy volitelných parametrů `<parameter>`, které jsou ve vzájemném vztahu parametr-hodnota. Vazba je specifikována pomocí atributů `additionalParRequired="true"` u elementu, který představuje název parametru. Tedy uvádí informaci, že je potřeba dalšího parametru, který bude obsahovat jeho hodnotu. Parametr, který nese konkrétní hodnotu je pak označen atributem `type="value"`. Dále tento parametr obsahuje atribut `editable` s hodnotou pravda/nepravda. Tento atribut specifikuje v aplikaci povolení pro uživatele specifikovat kromě hodnot popsaných v *XML* další hodnoty. Příkladem mohou být parametry *'isLinear'*, jehož hodnota je `isnumeric`. Jeho hodnota může být jakékoli číslo zadané uživatelem. Tedy je povoleno uživateli zadávat své hodnoty (`editable="true"`). Na rozdíl od toho jsou pak u parametru *'check'* povoleny pouze hodnoty `1` a `0`, ze kterých má uživatel možnost výběru (`editable="false"`).

Tímto to bylo předvedeno jak byla navržena syntaxe *XML* pro popis *API NEF*. Stejným způsobem jsou popsány komponenty náhodných velčin, systémů a estimátorů. Z důvodu obsáhlosti popisu *API NEF* není prostor pro detailnější popis. Pokračujme dále sekcí pro popis experimentu pomocí *XML*.

3.3 Popis experimentu pomocí XML

V této sekci bude představena syntaxe *XML* pro popis experimentu. Opět zde bude uveden kompletní výčet všech značek vytvořených pro popis experimentu. Syntaxe experimentu bude prezentována na příkladu (2.4.1), to zahrnuje ukázku kompletního konfiguračního souboru pro uvedený příklad s podrobným popisem každé řádky.

3.3.1 Výčet vytvořených značek pro popis experimentu

Tato sekce poskytuje kompletní přehled vytvořených značek za účelem popsat experiment. Většina těchto značek figuruje v každém z vytvořených souborů popisující experiment.

Přehled vytvořených elementů k popisu experimentu		
Element	**Hodnota**	**Popis**
`<nefexp>`	vnořené elementy	kořenový element
`<usesAPI>`	číslo verze *API NEF*	verze *API NEF*
`<experiment>`	vnořené elementy	začátek popisu experimentu
`<timeSteps>`	počet časových kroků	časový horizont experimentu
`<construct>`	vnořené elementy	vytvoření objektu
`<object>`	vnořené elementy	popis konkrétního vytvářeného objektu
`<objectInstance>`	vnořené elementy	popis konstruktoru vytvářeného objektu
`<attribute>`	hodnota parametru	popis parametru konkrétního objektu
`<objectMethod>`	vnořené elementy	popis metody vytvářeného objektu
`<systemInitialization>`	vnořené elementy	zastřešuje objekty pro inicializaci systému

Tabulka 3.3: Přehled vytvořených elementů k popisu experimentu.

Přehled vytvořených atributů k popisu experimentu			
Atribut	**Hodnota (příklad)**	**Vazba na element**	**Popis**
name	"Experiment1"	<experiment>	název experimentu
name	"system"	<object>	název objektu
type	"systemInit"	<object>	typ objektu
class	"nefEqSystem"	<objectInstance>	název třídy ke které přísluší objekt
myname	"MySystem1"	<objectInstance>	jméno pro pojmenování grafické komponenty v aplikaci
systemID	"1"	<objectInstance>	identifikátor systému
name	"f"	<attribute>	jméno parametru v *API NEF*
name	"simulate"	<objectMethod>	název metody příslušící objektu
x	"simState"	<objectMethod>	název proměnné do které se uloží posloupnost stavů systému
z	"simMeas"	<objectMethod>	název proměnné do které se uloží posloupnost měření systému
name	"nefEqSystem"	<systemInitialization>	název inicializovaného systému
systemID	"1"	<systemInitialization>	identifikátor inicializovaného systému
zslotID	"1"	<objectInstance>	identifikátor systémového slotu
type	"varargin"	<attribute>	volitelný parametr
val	"estimates"	<objectMethod>	název proměnné do které se uloží hodnoty poskytnuté metodou *estimate*

Tabulka 3.4: Přehled vytvořených atributů k popisu experimentu.

3.3.2 Popis experimentu příkladu (2.4.1) pomocí XML

Výpis 3.8: Popisu experimentu příkladu (2.4.1) pomocí *XML*.

```
<xml version="1.0"encoding="UTF-8"standalone="no">
<nefexp>
  <usesAPI>1.2.0</usesAPI>
  <experiment name="priklad_241">
  <timeSteps>20</timeSteps>
  <construct>
    <object name="system" type="system">
      <objectInstance class="nefEqSystem" myname="MySystem1" systemID="1">
        <attribute name="f">f</attribute>
        <attribute name="h">h</attribute>
        <attribute name="w">w</attribute>
        <attribute name="v">v</attribute>
        <attribute name="x0">x0</attribute>
      </objectInstance>
      <objectMethod name="simulate" x="x" z="z">
```

```
        <attribute name="obj">system</attribute>
        <attribute name="steps">time_steps</attribute>
        <attribute name="Input">[]</attribute>
      </objectMethod>
    </object>
  </construct>
  <systemInitialization name="nefEqSystem" systemID="1">
    <construct>
      <object name="f" type="systemInit">
<objectInstance class="nefHandleFunction" myname="nefHandleFunction" zslotID="1">
          <attribute name="fun">@(x,u,w,k)[x(1)*x(2)+w(1);x(2)+w(2)]</attribute>
          <attribute name="sizes">[2 0 2 0]</attribute>
      <attribute name="'Diff1Noise'" type="varargin">@(x,u,w,k) eye(2)</attribute>
        </objectInstance>
      </object>
    </construct>
    <construct>
      <object name="h" type="systemInit">
       <objectInstance class="nefLinFunction" myname="nefLinFunction" zslotID="2">
          <attribute name="f">[ 1  0 ]</attribute>
          <attribute name="g">[]</attribute>
          <attribute name="h">[ 1 ]</attribute>
        </objectInstance>
      </object>
    </construct>
    <construct>
      <object name="w" type="systemInit">
        <objectInstance class="nefGaussianRV" myname="nefGaussianRV" zslotID="3">
          <attribute name="m">[ 0 ; 0 ]</attribute>
          <attribute name="v">[ 0.05  0 ; 0  0.05 ]</attribute>
        </objectInstance>
      </object>
    </construct>
    <construct>
      <object name="v" type="systemInit">
        <objectInstance class="nefGaussianRV" myname="nefGaussianRV" zslotID="4">
          <attribute name="m">[ 0 ]</attribute>
          <attribute name="v">[ 0.01 ]</attribute>
        </objectInstance>
      </object>
    </construct>
    <construct>
      <object name="x0" type="systemInit">
        <objectInstance class="nefGaussianRV" myname="nefGaussianRV" zslotID="5">
          <attribute name="m">[ 0.9 ; -0.85 ]</attribute>
          <attribute name="v">[ 0.01  0 ; 0  0.01 ]</attribute>
        </objectInstance>
      </object>
    </construct>
  </systemInitialization>
  <construct>
    <object name="UKF" type="estimator">
      <objectInstance class="nefUKF" myname="nefUKF">
        <attribute name="system">system</attribute>
      </objectInstance>
```

```
      <objectMethod name="estimate" val="estimates">
        <attribute name="obj">UKF</attribute>
        <attribute name="Measurement">z</attribute>
        <attribute name="Input">[]</attribute>
      </objectMethod>
    </object>
  </construct>
  </experiment>
</nefexp>
```

Kořenovým elementem souboru je `<nefexp>` (zkratka pro *NEF* experiment), který označuje začátek experimentu. Opět poznamenejme, že elementy jsou vkládány hierarchicky jak je patrno z ukázky 3.6 a nebude to již dále v textu uváděno. Poté následuje element

```
<usesAPI>1.2.0</usesAPI>
```

jehož hodnota udává se kterou verzí *API NEF* byl experiment vytvořen. Pokud by se verze *API NEF* lišila s aktuální verzí, se kterou aplikace pracuje, tak aplikace zobrazí chybu a nenačte požadovaný experiment. Název experimentu pak obstarává element

```
<experiment name="priklad_241">
```

Dále následuje element, který udává časový horizont (počet časových kroků), na kterým bude experiment proveden.

```
<timeSteps>20</timeSteps>
```

Element `<construct>` pak znamená, že se bude v aplikaci vytvářet objekt.

```
<construct>..</construct>
```

Objekt bude mít jméno a bude nějakého typu. Typem je v tomto případě myšleno jaký význam má vytvářený objekt v aplikaci, neplést s typem komponent *NEF*. Tento typ hraje roli při parsování konfiguračního experimentu zpět do aplikace.

```
<object name="system" type="system">
```

Nyní přichází na řadu element, jehož atributy říkají, že objekt bude instancí konkrétní třídy *NEF* `class="nefEqSystem"` a instance bude jména zadané uživatelem `myname="MySystem1"`. Atribut `systemID="1"` je identifikátorem objektu (systému).

```
<objectInstance class="nefEqSystem" myname="nefEqSystem" systemID="1">
```

Další elementy pak představují parametrizaci objektu (systému). Hodnota elementu `<attribute>` je jméno objektu (v tomto případě funkce pro `f` a `h` a náhodné veličiny pro `w`, `v` a `x0`) zadané uživatelem a jeho atribut `name` je jméno parametru v *API NEF*.

```
<attribute name="f">f</attribute>
<attribute name="h">h</attribute>
<attribute name="w">w</attribute>
<attribute name="v">v</attribute>
<attribute name="x0">x0</attribute>
```

Dále přichází na řadu případné metody objektu. V tomto případě metoda *simulate*.

```
<objectMethod name="simulate" x="x" z="z">
<attribute name="obj">system</attribute>
<attribute name="steps">time steps</attribute>
<attribute name="Input">[]</attribute>
</objectMethod>
```

kde atribut `name` již typicky udává jméno metody. Atributy `x` a `y` zachycují jména výstupních parametrů metody z pohledu *API NEF* a jejich hodnoty jsou názvy proměnných do kterých je zaznamenávána posloupnost stavů systému a měření. Další vnořené elementy

```
<attribute name="obj">system</attribute>
<attribute name="steps">time steps</attribute>
<attribute name="Input">[]</attribute>
```

s názvem `<attribute>` představují parametrizaci metody *simulate*. Jejich hodnoty `system`, `time_steps` a `[]` udávají hodnoty jednotlivých parametrů metody *simulate*. Tedy jaký systém se bude simulovat a na jakém časovém horizontu a nakonec jaké bude řízení systému, v tomto případě řízení nebude systému z venčí poskytnuto a bude si ho generovat sám. Tímto končí první fáze vytvoření objektu systému a přejde se k druhé fázi inicializaci jednotlivých parametrů/objektů konstruktoru třídy *nefEqSystem(f, h, w, v, x0)*. Inicializace systému je označena elementem

```
<systemInitialization name="nefEqSystem" systemID="1">
```

kde jeho atributy udávají opět jméno inicializovaného systému a jeho identifikátor. Dále přichází na řadu zmíněný element `<construct>`, `<object>` a `<objectInstance>`. Tyto elementy mají opět za úkol vytvořit objekty odpovídajícího typu, jména, třídy a názvu instance s rozdílem, že zde není již identifikátor systému, ale identifikátor slotu `zslotID="1"` systému, do kterého se má daný objekt vložit.

```
<construct>
<object name="f" type="systemInit">
<objectInstance class="nefHandleFunction" myname="nefHandleFunction" zslotID="1">
<attribute name="fun">@(x,u,w,k)[x(1)*x(2)+w(1);x(2)+w(2)]</attribute>
<attribute name="sizes">[2 0 2 0]</attribute>
<attribute name="'Diff1Noise'" type="varargin">@(x,u,w,k) eye(2)</attribute>
</objectInstance>
</object>
</construct>
```

Hodnota elementu `<attribute>` udává hodnotu parametru a jeho atribut přiřazuje tuto hodnotu parametru konkrétního jména. Konkrétně vstupnímu parametru konstruktoru třídy *nefHandlefunction*, jenž je v popisu *API NEF* uveden pod tímto názvem. Takto jsou postupně vytvořeny všechny objekty, které inicializují systém. Po ukončovacím tagu elementu `</systemInitialization>` se pak v aplikaci patřičně zhotoví instance třídy systému.

Jako poslední je v *XML* popsána konfigurace estimátoru. Opět se zde nacházejí již zmiňované elementy `<construct>`, `<object>`, `<objectInstance>` a `<objectMethod>`.

```
  <construct>
    <object name="UKF" type="estimator">
      <objectInstance class="nefUKF" myname="nefUKF">
        <attribute name="system">system</attribute>
      </objectInstance>
      <objectMethod name="estimate" val="estimates">
        <attribute name="obj">UKF</attribute>
        <attribute name="Measurement">z</attribute>
        <attribute name="Input">[]</attribute>
      </objectMethod>
    </object>
  </construct>
```

Hodnota elementu `<attribute>` v elementu `<objectInstance>` pak udává nad jakým systémem bude estimátor provádět odhad. Jeho atribut `name` udává na jaký parametr se konkrétní hod-

nota váže v popisu *API NEF*. Obdobně pak u metody *estimate*, která je zavedena elementem `<objectMethod>`. Atributy tohoto elementu specifikují jméno metody a její výstup. Tedy, že výsledek odhadu bude uložen v proměnné `estimates` a váže se na výstupní parametr `val` uvedený v *API NEF* jako výstupní parametr této metody. Hodnota prvního elementu `<attribute>` metody *simulate* určuje estimátor tedy `UKF`, hodnota druhého elementu se stejným názvem pak udává měření systému `z` a prázdný vektor `[]` udává externí vstup estimátoru, který v tomto případě zůstává nevyužit.

3.4 Shrnutí

Tato kapitola nastínila na příkladech a ukázkách použitou syntaxi *XML* k popisu *API NEF* a popisu experimentu. Také se zde ke každému popisu nachází tabulky obsahující kompletní seznam vytvořených značek. Popis *API NEF* je členěn pro lepší přehlednost a orientaci do čtyř souborů. Tyto soubory popisují jednotlivé komponenty *NEF* jako jsou funkce (`nefAPI-functions.xml`), náhodné veličiny (`nefAPI-rv.xml`), systémy (`nefAPI-systems.xml`) a estimátory (`nefAPI -estimators.xml`). Popis experimentu byl do detailu rozebrán na příkladu 2.4.1.

S přihlédnutím na obsáhlost popisu *API NEF* a variaci různých popisů experimentů nebyl prostor pro detailnější analýzu vytvořené syntaxe *XML* jak pro popis *API NEF* tak pro popis experimentů. Bylo ovšem uvedeno vše nezbytné pro seznámí čtenáře s použitým přístupem k tomuto problému. Celkově se vytvořené soubory popisující *API NEF* blíží čtyřem tisícům řádků.

Kapitola 4

Implementace aplikace

Tato kapitola se bude věnovat implementaci důležitých dílčích aspektů nutných pro aplikaci *nefLab*. Tato kapitola je cílena především případným zájemcům o tuto aplikaci. Buďto za účelem převzetí a rozšíření konkrétních implementovaných funkcí nebo jednoduše za účelem detailnějšího pochopení implementace této aplikace. Dodejme, že aplikace byla nazvána *nefLab*. Název vychází z účelu této aplikace, jenž je laborování s komponenty *NEF*.

Nejprve zde budou uvedeny třídy, které slouží jako základní stavební kameny aplikace *nefLab*. To představuje krátký výčet klíčových tříd fungujících jako tzv. singleton a jejich významných metod. Poté bude předvedena implementace singletonu, který operuje nad daty, tedy představuje takové srdce celé aplikace z hlediska poskytování dat.

Dále bude kapitola obsahovat implementaci datové struktury, která se dělí na několik částí a má ji na starost právě zmiňovaný datový singleton. To zahrnuje implementaci *DOM* parseru, který načte *XML* soubory popisující *API NEF* do paměti ve stromové struktuře. Dále se z této stromové struktury vytvoří pole řetězců, ze které se již vytvoří datová struktura, kterou reprezentují obecné vzájemně hierarchicky propojené třídy pomocí tzv. kolekcí.

Kapitola také stručně zmíní implementaci knihovny aplikace *nefLab*. V další sekci bude uvedeno pár slov ohledně implementace parseru konfiguračního souboru experimentu. Nakonec bude v rychlosti představena implementace javovské knihovny *Matlabcontrol*. Závěr pak bude patřit krátkému shrnutí informací obsažených v této kapitole.

4.1 Klíčové třídy aplikace

V úvodu kapitoly bylo zmíněno, že klíčové třídy aplikace *nefLab* fungují jako tzv. singleton. Jinými slovy jsou tyto třídy implementovány pomocí návrhového vzoru singleton. Singleton představuje zajištění existence pouze jedné instance dané třídy a poskytnutí globálního přístupu k ní. Takto lze docílit toho, aby v aplikaci byly načítány například data pouze na jednom místě v kódu. Samozřejmě se nemusí jednat vždy o data, ale cokoli co programátor vyžaduje mít na jednom místě a vždy po ruce. Pro práci s těmito například daty pak budou implementovány metody. Tyto metody pak lze volat kdekoli v kódu. V aplikaci *nefLab* bylo použito hned několik typů singletonů. Tyto singletony nejsou implementované úplně stejným způsobem, ale každá z nich má velmi podobný účel a to poskytovat globálně pouze jednu instanci třídy. Jinými slovy slouží jako operační centra pro jednotlivé části aplikace *nefLab*.

Následuje výčet klíčových tříd aplikace *nefLab*. Ke každé třídě je uveden stručný popis jejího účelu, popřípadě popis jejích metod.

- `DataStorage.java`
  - vrací načtené *XML* dokumenty typu `Document`
  - vytváří z těchto dokumentů pole řetězců typu `String[]`
  - z pole řetězců sestavuje datovou strukturu z předem vytvořených obecných tříd `NefComponent.java`, `Method.java`, `Parameter.java` a `Attribut.java`
  - vrací instance třídy `NefComponent.java`
- `ExperimentStorage.java`
  - ukládá všechny instance třídy `Experiment.java` na jediném místě (spadá do datové vrstvy)
  - obsahuje metodu `getExperimentId(Experiment experiment)` - vrací identifikátor experimentu
  - obsahuje metodu `get(int index)` - vrací instanci třídy `Experiment.java` na základě poskytnutého identifikátoru
  - obsahuje metodu `load(File file, int experimentId, int expGUIid)` - načte experiment z konfiguračního souboru experimentu *XML* (kapitola 3.3)
  - obsahuje metodu `save(File file, int experimentId, int expGUIid)` - uloží experiment do konfiguračního souboru experimentu *XML* (kapitola 3.3)
  - obsahuje metodu `playSeq(Experiment experiment, int expGUIid)` - pošle sekvenčně programovému prostředí Matlab vypočítat experiment (po jednotlivých příkazech)
  - obsahuje metodu `playBatch(Experiment experiment, int expGUIid)` - pošle dávkově programovému prostředí Matlab vypočítat experiment (celý *m-skript*)
  - obsahuje metodu `exportToMFile(File file, Experiment experiment,... int expGUIid)` - přeloží/exportuje experiment do matlabovského *m-skriptu*
  - obsahuje metodu `exportToMATFile(File file, Experiment experiment,... int expGUIid)` - uloží/exportuje data experimentu do matlabovského souboru `mat-filu`
- `ExperimentGUIStorage.java`
  - ukládá všechny instance třídy `ExperimentGUI.java` na jediném místě (třída `ExperimentGUI.java` spadá do prezenční vrstvy)
  - obsahuje metody které vracejí jednotlivé uložené instance podle identifikátoru či jejich identifikátory podobně jako třída `ExperimentStorage.java`
  - dále obsahuje velké množství metod pro vytváření grafických komponent, které reprezentují samotné objekty *NEF* (vytváření komponent při načtení konfiguračního souboru experimentu a při `drag and dropu` objektů přetažených z knihovny)

- `Library.java`
  - vytváří grafickou podobu knihovny
  - nastavuje zdroje a cíle pro funkci `drag and drop`
  - implementuje základní funkce s rozbalovacím stromem, který obsahuje komponenty vytvořených z popisu *API NEF*
- `Matlab.java`
  - obsahuje metodu `getMeProxy()` - vytvoří a vrátí nové spojení s programovým prostředím *Matlab* pokud ještě nebylo vytvořeno, pokud bylo vytvořeno tak vrátí jeho spojení (návratová hodnota je typu `MatlabProxy`)
  - obsahuje metodu `createMatlab()` - vytvoří a vrátí nové spojení s programovým prostředím *Matlab* (návratová hodnota je typu `MatlabProxy`)

Nyní bude uvedena ukázka implementace singletonu `DataStorage.java` v aplikaci *nefLab*.

Výpis 4.1: Implementace singletonu v aplikaci *nefLab*.

```
public class DataStorage {
  //Vytvorime staticky objekt ktery ma hodnotu null
  private static DataStorage instance = null;
  :
  //Vytvorime soukromy konstruktor
  private DataStorage() {
    //Konstruktor pro vytvoreni objektu tzv. jedinacek
    //Tato cast kodu se vykona pouze jednou pri prvnim vytvoreni teto instance
    //Zde nacitame data DOM parserem a tvorime z¬nich datovou strukturu
    :
  }
  //Pokud je instance rovna null vytvorime objekt
  public static DataStorage getInstance() {
    if (instance == null) {
      instance = new DataStorage();
    }
    %//vracime objekt
    return instance;
  }
  //Implementace metod pro praci se singletonem
  :
  public NefComponent getComponent(String name) {
    :
  }
  public String getAPIversion() {
    :
  }
  public Document getConfigDocument() {
    return configDocument;
  }
  public Document getFunctionsDocument() {
    return functionsDocument;
  }
  public Document getRvDocument() {
    return rvDocument;
  }
  public Document getSystemsDocument() {
    return systemsDocument;
  }
  public Document getEstimatorsDocument() {
    return estimatorsDocument;
  }
}
```

Dále uvedeme jak lze tento singleton volat v aplikaci *nefLab* například k získání verze *API NEF*.

Výpis 4.2: Volání singletonu v aplikaci *nefLab*.

```
String version = DataStorage.getInstance().getAPIversion();
```

Takto bylo v aplikaci implementováno načítání *XML* dokumentů obsahující popis *API NEF*. Z těchto dokumentů poté byla vytvořena datová struktura, ke které jsme tímto z hlediska kódu získali globální přístup.

4.2 Datová struktura

Tato sekce poskytne podrobný popis k přístupu implementace datové struktury. Implementace datové struktury sestává z několika nezbytných kroků. Prvním krokem je načtení souborů *XML* popisující *API NEF* aplikací do dokumentu typu `Document` využitím *DOM* parseru (3.1.4) a (4.2.1). Jinými slovy nám *DOM* parser vytvoří v paměti stromovou strukturu typu `Document`. Z této stromové struktury vytvoříme pole řetězců postupným procházením stromu a odchytáváním jednotlivých elementů a atributů a jejich hodnot. Výsledné pole řetězců pak již představuje dobře strukturované pole, ze kterého lze naplnit připravenou strukturu vzájemně provázaných tříd `NefComponent.java`, `Method.java`, `Parameter.java` a `Attribut.java` konkrétními daty z *API NEF*. Tyto třídy jsou provázané pomocí dynamických seznamů *ArrayListů*. Jejich struktura bude rozebrána dále v textu.

Následující body ve zkratce shrnují kroky pro vytvoření datové struktury.

- implementace *DOM* parseru
- vytvoření pole řetezců
- struktura tříd popisující komponenty *NEF*
- vytvoření a naplnění struktury tříd konkrétními daty

4.2.1 DOM parser

Parser vytvoří objekty umožňující metodou *DOM* přečíst soubor `nefAPI-functions.xml`. Dá se použít jako verifikátor (*well-formed*). Nevypíše-li chybu, je *XML* soubor v pořádku. Příkazem

```
dbf = DocumentBuilderFactory.newInstance();
```

se vytvoří tzv. obálka pro univezální parser. Dále pak příkaz

```
builder = dbf.newDocumentBuilder();
```

vytvoří vlastní parser. Není zřejmé, jaký skutečný parser se použije. Příkazem

```
dbf.setIgnoringComments(true);
```

pak požadujeme, aby parser ignoroval komentáře.

Takto v kódu zavoláme *DOM* parser.

Výpis 4.3: Volání *DOM parseru* v aplikaci *nefLab*.

```
functionsFilePath = "lib/nefapi/nefAPI-functions.xml";
ROOT_ELEMENT = "nefapi";
XmlBase functionsXmlBase = new XmlBase(functionsFilePath, ROOT_ELEMENT);
functionsDocument = functionsXmlBase.getDocument();
```

Následuje ukázka celé implementace *DOM* parseru v aplikaci *nefLab*.

Výpis 4.4: Implementace *DOM* parseru v aplikaci *nefLab*.

```
public class XmlBase {
  private Document document;
  private DocumentBuilderFactory dbf;
  private DocumentBuilder builder;
  DOMImplementation domImplementation;
  private String rootElement;

  public XmlBase(String fileName, String rootElement) {
    this.rootElement = rootElement;
    dbf = DocumentBuilderFactory.newInstance();
    try {
      builder = dbf.newDocumentBuilder();
    } catch (ParserConfigurationException e) {
      e.printStackTrace();
    }
    dbf.setIgnoringComments(true);
    domImplementation = builder.getDOMImplementation();

    File xmlFile = new File(fileName);
    if (xmlFile.exists()) {
      try {
        document = builder.parse(xmlFile);
      } catch (SAXException e) {
        e.printStackTrace();
      } catch (IOException e) {
        e.printStackTrace();
      }
    } else {
      document = createDocument();
    }
  }
  protected Document createDocument() {
    Document tempDoc = domImplementation.createDocument(null, rootElement, null);
    tempDoc.createElement(rootElement);
    return tempDoc;
  }
  public Document getDocument() {
    return document;
  }
}
```

4.2.2 Pole řetězců

Ze stromové struktury poskytnuté *DOM* parserem nyní vytvoříme pole řetězců. V singletonu `DataStorage.java` zavoláme metodu `getArrayFromDoc(functionsDocument)` třídy `ArrayFromDoc.java`.

Výpis 4.5: Vytvoření pole řetězců ze stromové struktury poskytnuté parserem *DOM*.

```
functionsArray = ArrayFromDoc.getArrayFromDoc(functionsDocument);
```

Výpis 4.6: Vytvoření pole řetězců ze stromové struktury poskytnuté parserem *DOM*.

```
public class ArrayFromDoc {
  public static String[] getArrayFromDoc(Document doc) {
    odstranMezeryALF(doc);

    ArrayList<String> functions = new ArrayList<String>();
    NodeList nl = doc.getElementsByTagName("*");
    for (int i = 0; i < nl.getLength(); i++) {
      Element e = (Element) nl.item(i);
      functions.add(("Element: " + e.getNodeName()));
      NodeList nodeList = e.getChildNodes();
      for (int l = 0; l < nodeList.getLength(); l++) {
        Node child = nodeList.item(l);
        if (child.getNodeType() == Node.TEXT_NODE) {
    functions.add(("Element value: " + child.getTextContent().trim().toString()));
        }
      }
      for (int j = 0; j < e.getAttributes().getLength(); j++) {
        String t = e.getAttributes().item(j).toString();
        functions.add("Attribut: " + t);
      }
    }
    String e1 = "END OF";
    String e2 = "DOC";
    functions.add(e1);
    functions.add(e2);
    String[] s≠ (String[]) functions.toArray(new String[0]);
    return s;
  }
  public static void odstranMezeryALF(Document doc) {
    Node n = doc.getDocumentElement();
    NodeIterator ni = ((DocumentTraversal) doc).createNodeIterator(
            doc.getDocumentElement(),
            NodeFilter.SHOW_TEXT,
            new PrazdnyText(), true);
    while ((n = ni.nextNode()) != null) {
      Node rodic = n.getParentNode();
      rodic.removeChild(n);
    }
  }
  public static class PrazdnyText implements NodeFilter {
    @Override
    public short acceptNode(Node n) {
      if (n.getNodeValue().trim().length() == 0) {
        return NodeFilter.FILTER_ACCEPT;
      } else {
        return NodeFilter.FILTER_SKIP;
      }
    }
  }
}
```

Poznamenejme, že bylo potřeba odstranit tzv. odřádkovací nody. Tyto nody mohou jednoduše rozhodit práci algoritmu, který vytváří ze stromové struktury pole řetězců, při průchodu *DOM*

dokumentu. K odstranění se použila metoda `odstranMezeryALF(Document doc)`, která po zavolání všechny odřádkovací nody z *DOM* odstraní. Tato metoda byla převzata z [6]. Následuje krátká ukázka části vytvořeného pole řetězců.

Výpis 4.7: Část pole řetězců.

```
Element: nefapi
Attribut: version="1.2.0"
Element: component
Attribut: type="functions"
Element: class
Attribut: extends="none"
Attribut: name="nefFunction"
Element: methods
Element: constructor
:
```

4.2.3 Struktura tříd popisující komponenty NEF

Datovou strukturu v aplikaci `nefLab` můžeme přirovnat k tzv. stromové datové struktuře. Počet potomků ve svých vnitřních uzlech je dán popisem *API NEF*, respektive počtem tříd *NEF*, jejich metod, parametrů a atributů. Vnitřní uzly pomyslného stromu tedy reprezentují obecně komponenty, metody (včetně konstruktorů) a parametry. Koncové uzly nebo-li listy obecně představují atributy. Vrcholem nebo-li kořenem datové struktury je dynamický seznam *ArrayList*, do kterého jsou na základě popisu *API NEF* přidány objekty, jež jsou instance třídy `NefComponent.java`. Tato třída obecně popisuje třídu *NEF*, neboli komponentu *NEF* (např. *nefLinFunction* nebo *nefEqSystem*). Obsahuje data jako název třídy komponenty *NEF*, typ komponenty a další dynamický seznam, který obsahuje objekty jenž jsou instancemi třídy `Method.java`. Každý tento objekt, který je instancí třídy `Method.java`, představuje příslušné metody třídy *NEF*. Data třídy `Method.java` představují název metody a opět několik dynamických seznamů, které obsahují objekty, jež jsou instance třídy `Parameter.java`. Tyto objekty reprezentují vlastní parametry metody. Třída `Parameter.java` obsahuje název parametru a seznam atributů. Objekty v seznamu jsou objekty instance třídy `Attribut.java`. Třída `Attribut.java` obsahuje již pouze název atributu. Tímto jsme dosáhli tzv. listu v pomyslné stromové struktuře.

Struktura tříd je tedy vytvořena pomocí dynamických seznamů *ArrayListů* a tříd `NefComponent.java`, `Method.java`, `Parameter.java` a `Attribut.java`. Třídy jsou strukturovány tak, aby popisovaly komponentu *NEF*.

Pro názornost výše uvedeného textu bude vhodné uvést ukázku každé třídy. To poskytne čtenáři dostatečný příklad jak jsou mezi sebou jednotlivé třídy propojeny.

Výpis 4.8: Ukázka třídy `NefComponent.java`.

```
public class NefComponent {
  private String className;
  private ArrayList<Method> methods;
  private String type;
  public boolean methodsCorrected = false;

  public String getClassName() {return className;}
  public ArrayList<Method> getMethods() {return methods;}
  public String getType() {return type;}

  public void setType(String type) {this.type = type;}
  public void setclassName(String className) {this.className = className;}

  public NefComponent(String className, ArrayList<Method> methods) {
    this.className = className;
    this.methods = methods;
  }
}
```

Třída `Method.java` již obsahuje spoustu metod a nelze nebo není je vhodné zde všechny uvést.

Výpis 4.9: Ukázka třídy `Method.java`.

```
public class Method {
  private String methodName;
  private ArrayList<Parameter> parametersPlusResults;

  private List<Parameter> parameters;
  private ArrayList<Parameter> vararginParameters;
  private ArrayList<Parameter> results;

  public Method(String methodName, ArrayList<Parameter> parametersPlusResults) {
    this.methodName = methodName;
    this.parametersPlusResults = parametersPlusResults;
    build();
  }
```

Ukázka třídy `Parameter.java` opět pouze zobrazuje konstruktor třídy s vynecháním všech ostatních metod, které pro nás, ale nejsou momentálně důležité.

Výpis 4.10: Ukázka třídy `Parameter.java`.

```
public class Parameter {
  private String parameterName;
  private ArrayList<Attribut> attributes;
  private Object source;
  private RowSpinner rowSpinner;
  private ColSpinner colSpinner;
  private boolean useAsVarargin = false;
  private String valueForVarargin;

  public Parameter(String parameterName, ArrayList<Attribut> attributes) {
    this.parameterName = parameterName;
    this.attributes = attributes;
  }
```

A nakonec je uvedena ukázka třídy `Attribut.java`.

Výpis 4.11: Ukázka třídy `Attribut.java`.

```
public class Attribut {
  private String attributName;

  public String getAttributName() {return attributName;}

  public Attribut(String attributName) {
    this.attributName = attributName;
  }
}
```

4.2.4 Vytvoření a naplnění struktury tříd konkrétními daty

Nyní již je vše připraveno k vytvoření a naplnění struktury tříd konkrétními daty z *API NEF*. Algoritmus postupně prochází celé pole řetězců a vytváří objekty tříd `NefComponent.java`, `Method.java`, `Parameter.java` a `Attribut.java` do stromové datové struktury popsané v (4.2.3). Algoritmus je implementován v metodě

```
private static ArrayList<NefComponent> CreateStructure(String[] array) {
```

náležící třídě `Structure.java`. Vstupním parametrem této metody je pole řetězců a výstupem je dynamický seznam *ArrayList*, který obsahuje objekty typu *NefComponent*. Nasleduje ukázka kódu algoritmu pro vytváření datové struktury.

Výpis 4.12: Ukázka kódu algoritmu pro vytváření datové struktury.

```
  private static ArrayList<NefComponent> CreateStructure(String[] array) {
   :
    for (int w = 0; w < arrayLength; w++) {
      if (array[w].equals("Element: class")) {
        if (array[w].equals("Element: class") && writeml == true) {
          component = new NefComponent(className, ml);
          cl.add(component);
          methods = new Method(metodName, pl);
          ml.add(methods);
          ml = new ArrayList<Method>();
          pl = new ArrayList<Parameter>();
        }
        writeal = false;
        writeml = true;
        className = array[w + 2].subSequence(10, array[w + 2].length()).toString();
      }
      if (array[w].equals("Element: constructor")) {
        writeal = false;
        metodName = array[w].subSequence(9, array[w].length()).toString();
        writepl = true;
      }
      if (array[w].equals("Element: method")) {
        writeal = false;
        if (array[w].equals("Element: method") && writepl == true) {
          methods = new Method(metodName, pl);
          ml.add(methods);
```

```
            pl = new ArrayList<Parameter>();
          }
          metodName = array[w + 1].subSequence(10, array[w + 1].length()).toString();
        }
        if (array[w].equals("Element: parameters")) {
          div = array[w].subSequence(9, array[w].length()).toString();
          parameters = new Parameter(div, new ArrayList<Attribut>());
          pl.add(parameters);
        }
        if (array[w].equals("Element: results")) {
          div = array[w].subSequence(9, array[w].length()).toString();
          parameters = new Parameter(div, new ArrayList<Attribut>());
          pl.add(parameters);
        }
        if (array[w].startsWith("Element value: ")) {
          parameterName = array[w].subSequence(15, array[w].length()).toString();
          writeal = true;
        }
        if (array[w].startsWith("Attribut: ") && writeal == true) {
          sq = array[w].subSequence(10, array[w].length());
          at[w] = new Attribut(sq.toString());
          al.add(at[w]);
          if (!array[w + 1].startsWith("Attribut: ")) {
            parameters = new Parameter(parameterName, al);
            al = new ArrayList<Attribut>();
            pl.add(parameters);
          }
        }
        if (array[w].equals("END OF")) {
          methods = new Method(metodName, pl);
          ml.add(methods);
          component = new NefComponent(className, ml);
          cl.add(component);
        }
      }
      return cl;
    }
```

4.3 Implementace knihovny a knihovních bloků

S připravenou datovou strukturou můžeme přejít k implementaci knihovny a jejích bloků. Knihovna aplikace *nefLab* sestává především z rozbalovacího stromu (*JTree*), který slouží jako navigátor v knihovně, a panelu (*JPanel*), který zobrazuje knihovní bloky. Pro případný náhled viz. kapitola 5, obrázek 5.1.

Implementaci rozbalovacího stromu zajišťuje třída `TreeBuilder.java`. Třída obsahuje statickou metodu `buildTree()`, která v první řadě získá dokumenty parseru *DOM* (4.2.1) ze singletonu `DataStorage.java` (4.1). Zopakujme, že tyto dokumenty popisují *API NEF*. Postupným průchodem těchto dokumentů (jedná se též o strom) je pak vytvořena struktura rozbalovacího stromu. Jeho struktura je tvořena šesti základními uzly a jedním kořenovým. Kořen zobrazuje verzi *API NEF*. A uzly dělí knihovnu na šest skupin, konkrétně generátory signálu, funkce,

náhodné veličiny, systémy, estimátory a výsledky. Listy těchto uzlů představují názvy jednotlivých komponent *NEF* či bloků, které byly vytvořeny externě (ne na základě popisu *NEF*).

To se týká bloků generátorů signálu a výsledků. Tyto bloky nejsou *NEF* poskytovány a byly implementovány navíc. Z generátorů signálu to jsou bloky *Matrix*, *Sine*, *Square*, *Sawtooth*, *Step*, *Ramp* a *Read data from file* a dva bloky *nefFunction box* a *nefRV box*. Poslední dva zmiňované jsou sloty pro *funkce* a *náhodné veličiny*, které uživatel může použít jako zdroj signálu. Ze skupiny výsledků to jsou potom *pdfLinePlot*, *pdfPunEstPlot*, *pdfTimePlot* a *pdfGaussianPlot*.

Vraťme se dále k rozbalovacímu stromu, kterému byl implementován MouseListener. Tento listener umožňuje uživateli procházet strom pomocí myši. Na stejnou událost pak reaguje i panel, který zobrazuje odpovídající komponenty vybrané v rozbalovacím stromu. Základem grafické podoby komponenty je obyčejné java tlačítko nebo-li *JButton*. O grafických reprezentacích komponent nebo-li blocích se čtenář dozví více v 5.2. Tomuto bloku je pak při zobrazení zároveň implementována funkce *drag and drop*. Která poskytuje možnost přetáhnutí bloku na pracovní plochu estimačního experimentu. Kromě přetáhnutí zde má možnost uživatel také kliknout na blok. Tento klik pak otevře parametrizační okno bloku (komponenty).

Tímto to byla stručně popsána implementace knihovny a knihovních bloků. Tento úkol je ve zdrojovém kódu řešen pomocí tříd, které spadají do balíků (packages) `gui.buttons`, `gui.library` a `gui.parametrization`.

4.4 Parser konfiguračního souboru experimentu

K načtení konfiguračního souboru experimentu je zde opět využito *DOM* parseru (4.2.1). Tento parser pak předá vytvořenou stromovou strukturu třídě `LoadExperiment.java`. Tato třída postupně prochází stromovou strukturu a pomocí nespočet metod vytváří zpětně všechny objekty *NEF*, tím je docíleno zpětného vytvoření experimentu na základě konfiguračního souboru *XML*.

Ukládání konfiguračního souboru experimentu je pak implementováno v třídě `SaveExperiment.java`. Tato třída postupně vytváří již dobře známý dokument typu `Document` (stromovou strukturu), aby ji pak mohla zapsat do souboru *XML*. Tento dokument je vytvořen postupným průchodem experimentu tedy třídy `Experiment.java` a jejími objekty třídy `NefComponentInstance.java` představující jednotlivé komponenty estimačního experimentu. Tyto objekty jsou ve třídě `Experiment.java` drženy pomocí dynamických seznamů *ArrayList*.

Znovu je třeba uvést, že tento problém je dosti komplexní a není možno mu věnovat tolik prostoru. V případě zájmu je možno nahlédnout do přiloženého zdrojového kódu.

4.5 Knihovna Matlabcontrol

Matlabcontrol je javovské *API*, které umožňuje volání programového prostředí *Matlab* z *Javy*. Můžeme využívat matlabovských funkcí jako eval (provedení matlabovského výrazu v textovém řetězci), feval (vyhodnocení funkce) nebo například nastavovat a získávat proměnné. Interakce může probíhat buďto přímo z *Matlabu* nebo z venčí. Nová verze umožňuje mnoho funkcí jako je opětovné připojení k předešlé relaci *Matlabu*, spuštění *Matlabu* na pozadí nebo lepší zpracování navrácených dat.

Výpis 4.13: Ukázka volání programového prostředí *Matlab* z aplikace *nefLab*.

```
Matlab.getMeProxy().eval("clc");
```

Základní postup pro použití *matlabcontrolu* je vytvořit tzv. factory a poté vytvořit tzv. proxy, viz. výpis 4.13. Proxy je pak použito k samotné komunikaci s *Matlabem*, viz. výpis 4.14.

Výpis 4.14: Ukázka třídy `Matlab.java` v aplikaci *neflab*.

```
public class Matlab {

  private static MatlabProxy proxy;

  private static void createMatlab() throws MatlabConnectionException,
   MatlabInvocationException {
    boolean hidden = true;
    MatlabProxyFactoryOptions options = new MatlabProxyFactoryOptions.
    Builder().setHidden(hidden).setUsePreviouslyControlledSession(true).build();
    MatlabProxyFactory factory = new MatlabProxyFactory(options);
    proxy = factory.getProxy();
  }

  public static MatlabProxy getMeProxy() {
    if ( proxy==null || !proxy.isConnected()){
      try {
        createMatlab();
      } catch (MatlabConnectionException ex) {
        Logger.getLogger(Matlab.class.getName()).log(Level.SEVERE, null, ex);
      } catch (MatlabInvocationException ex) {
        Logger.getLogger(Matlab.class.getName()).log(Level.SEVERE, null, ex);
      }
    }
    return proxy;
  }
}
```

Pro více informací o tomto *API* [9].

4.6 Shrnutí

Tato kapitola se zabývala implementací prvotních dílčích úkolů, které bylo třeba vyřešit, aby mohla vzniknout aplikace *nefLab*. Byly zde uvedeny informace k singletonům, velice oblíbeným návrhovým vzorům používaných nejen v *Javě*. Dále byla do detailů rozebrána tvorba datové struktury z popisu *API NEF*. Kapitola také dále stručně popsala implementaci knihovny *nefLab*, implementaci parseru konfiguračního souboru experimentu a nakonec implementaci javovské knihovny *Matlabcontrol*. Poznamenejme ovšem, že tyto úkoly představují pouze špičku ledovce z hlediska naprogramování celé aplikace *nefLab*.

Kapitola 5

Realizace aplikace

Po kapitole, která důkladně popsala implementaci toho nejdůležitějšího v aplikaci *neflab* se nyní přesuneme k její realizaci. Především se zde čtenář seznámí s vytvářením estimačních experimentů na dvou příkladech, které byly uvedeny v kapitole druhé, konkrétně (2.4.1) a (2.4.2). Tímto bude zajištěno porovnání vytváření estimačních experimentů v *NEF*, kterému tato aplikace slouží jako podpora, s porovnáním vytváření estimačních experimentů v samotné aplikaci *nefLab*. Kromě samotného vytvoření estimačních experimentů bude popsána také jeho správa, což zahrnuje především uložení a načtení estimačních experimentů. Ke každému příkladu bude také přiložen k nahlédnutí odpovídající výsledný matlabovský skript, který je samozřejmě možno samostatně v programovém prostředí *Matlab* spustit. Toto vše přinese čtenáři krátkou průpravu k užívání aplikace. Závěr kapitoly bude patřit několika větám, které shrnou informace získané v této kapitole.

5.1 Spuštění aplikace nefLab

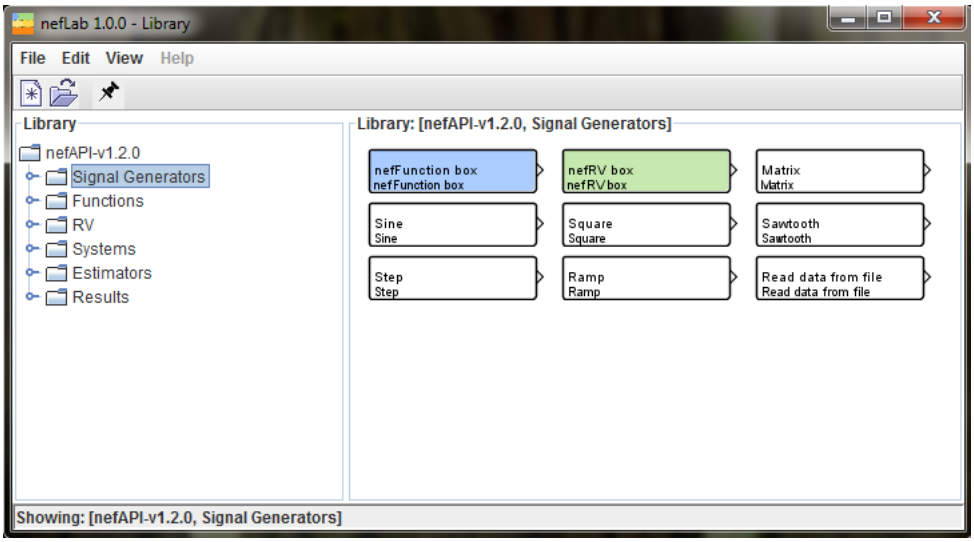
V příkazové řádce operačního systému se přesuneme do složky obsahující soubor `nefLab-v1.0.0.jar`. Nutno poznamenat, že složka spolu s tímto souborem musí také obsahovat složku `lib`, která obsahuje veškeré knihovní soubory nutné k běhu aplikace. Pokud je vše splněno, aplikace se pak spustí příkazem

```
java -jar "nefLab-v1.0.0.jar"
```

Tímto se úspěšně spustila aplikace *nefLab* a můžeme přejít k další sekci kapitoly.

5.2 Popis knihovní části aplikace nefLab

Po spuštění aplikace se zobrazí aplikační okno s názvem *nefLab - Library*. V levé části se nachází rozbalovací strom, jehož vrcholem je verze *nefAPI*, se kterou aplikace pracuje. Jeho další uzly pak rozdělují knihovní rozbalovací strom do šesti hlavních skupin, konkrétně to jsou generátory signálů, funkce, náhodné veličiny, systémy, estimátory a výsledky. Každá tato skupina, jak názvy napovídají, má v estimačním experimentu svůj účel a také obsahuje komponenty, které tento účel naplňují. V pravé části se pak nachází panel, který reaguje na události rozbalovacího stromu a zobrazuje seznam knihovních bloků pro příslušnou vybranou skupinu. Jediným kliknutím na tento blok se pak otevře instance pro jeho vlastní parametrizaci ovšem s nepovoleným zápisem, jelikož blok ještě nebyl přetáhnut do estimačního experimentu. Tato funkce je spíše informativního charakteru a slouží uživateli především k prvotnímu nahlédnutí k čemu a jak konkrétní komponenta slouží. Obrázek 5.1 ilustruje knihovnu aplikace *nefLab* s vybranými knihovními bloky skupiny generátorů signálu.



Obrázek 5.1: Knihovna aplikace *nefLab* zobrazující komponenty generátorů signálu.

Skupina generátorů signálu

Generátory signálu se skládají z knihovních bloků uvedených v tabulce 5.1.

Seznam bloků skupiny generátorů signálu a jejich popis	
Název bloku	**Popis**
Matrix	Blok pro řádkový vektor nebo matici. Reprezentuje pevně zadanou sekvenci skalární nebo vektorové veličiny.
Sine	Blok sinusového signálu.
Square	Blok obdélníkového pulsu.
Sawtooth	Blok pilové funkce.
Step	Blok skokové funkce.
Ramp	Blok rampové funkce.
Read data from file	Blok pro načtení dat ze souboru.
nefFunction box	*Drag and drop* slot pro funkce.
nefRV box	*Drag and drop* slot pro náhodné veličiny.

Tabulka 5.1: Knihovní bloky skupiny generátorů signálu.

Každý blok v této skupině byl vytvořen dodatečně tedy ne na základě popisu *API NEF*. Tyto bloky nejsou *NEF* poskytovány a byly implementovány navíc. Dva poslední bloky, slouží jako *drag and drop* sloty pro další typ knihovních bloků a to funkcí a náhodných veličin použitých jako zdroj signálu. Obrázek 5.1 je pak ilustrací zobrazení generátorů signálu v knihovně *nefLab*.

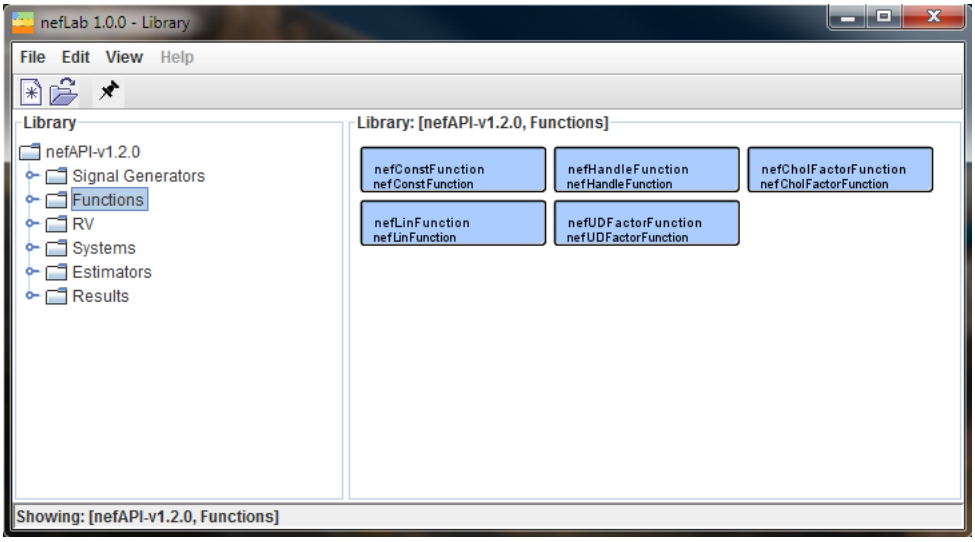
Skupina funkcí

Skupinu funkcí reprezentují knihovní bloky, které jsou uvedeny v tabulce 5.2.

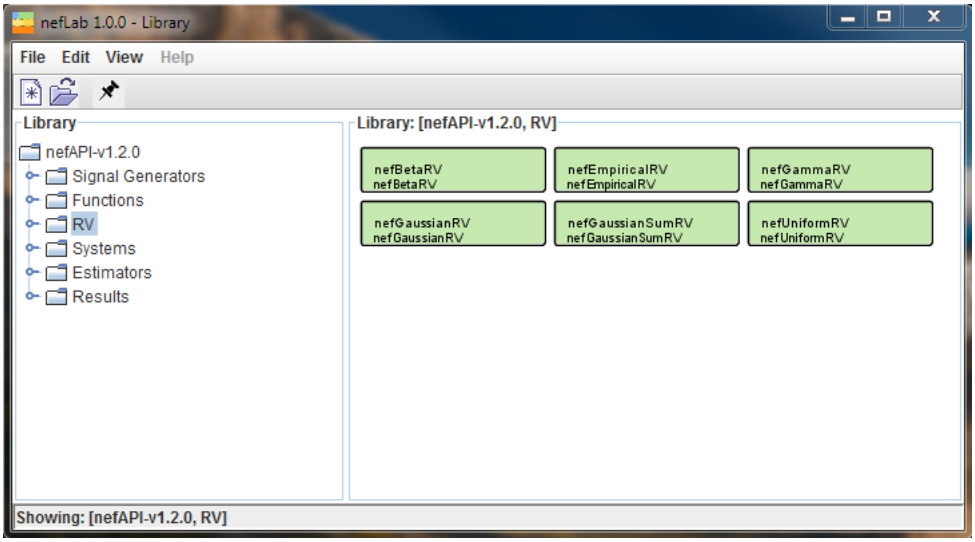
Seznam bloků skupiny funkcí a jejich popis	
Název bloku	**Popis**
nefConstFunction	Blok konstantní funkce.
nefHandleFunction	Blok anonymní (handle) funkce.
nefCholFactorFunction	Blok funkce pro odmocninový rozklad.
nefLinFunction	Blok lineární funkce.
nefUDFactorFunction	Blok funkce pro UD rozklad.

Tabulka 5.2: Knihovní bloky skupiny funkcí.

Poznamenejme, že tyto bloky nemohou být vloženy samostatně do estimačního experimentu (nemají vstupní a výstupní piny). Většinou jsou vkládány do bloku systému jako jeho popis nebo do bloku *nefFunction box* jako například generátor vstupního signálu pro systém. Obrázek 5.2 je pak ilustrací zobrazení funkcí v knihovně *nefLab*.



Obrázek 5.2: Knihovna aplikace *nefLab* zobrazující komponenty funkcí.



Obrázek 5.3: Knihovna aplikace *nefLab* zobrazující komponenty náhodných veličin.

Skupina náhodných veličin - RV

Skupina náhodných veličin nabízí následující komponenty uvedené v tabulce 5.3.

Seznam bloků skupiny náhodných veličin a jejich popis	
Název bloku	**Popis**
nefBetaRV	Beta rozdělení je spojité rozdělení pravděpodobnosti definované na intervalu $[0, 1]$ a je parametrizované dvěma parametry, typicky označovány α a β.
nefEmpiricalRV	Empirické rozdělení je součtová distribuční funkce, která poskytuje pravděpodobnost $F(x) = Pr\{X <= x\}$.
nefGammaRV	Gamma rozdělení je dvou-parametrové spojité rozdělení pravděpodobnosti. Disponuje scale parametrem θ a shape parametrem k.
nefGaussianRV	Normální rozdělení (Gaussovské rozdělení) je spojité rozdělení pravděpodobnosti, které popisuje data, které se shlukují kolem střední hodnoty nebo průměrné hodnoty.
nefGaussianSumRV	Jedná se o vážený součet více Gaussovských rozdělení pravděpodobnosti, přičemž součet jednotlivých vah je roven jedné.
nefUniformRV	Rovnoměrné rozdělení, které můžeme charakterizovat tím, že všechny hodnoty konečné množiny možných hodnot jsou stejně pravděpodobné.

Tabulka 5.3: Knihovní bloky skupiny náhodných veličin.

Poznámka: Jednotlivé třídy popisující hustoty pravděpodobnosti uvedené v tabulce 5.3 jsou podtřídami rodičovské třídy nefRV.

Podobně jako bloky skupiny funkcí nemohou být vkládány samostatně do estimačního experimentu (též nemají vstupní a výstupní piny). Slouží jako popis pro systémy nebo jako zdroj předgenerovaných vzorků stavových šumů či měření systémů. Obrázek 5.3 ilustruje zobrazení náhodných veličin v knihovně *nefLab*.

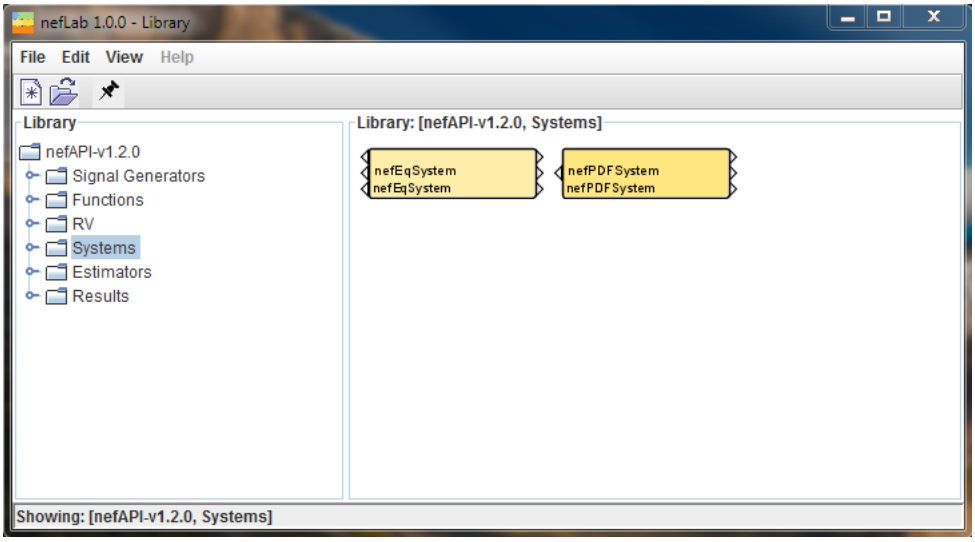
Skupina systémů

Skupina systémů nabízí následující komponenty uvedené v tabulce 5.4.

Seznam bloků skupiny systémů a jejich popis	
Název bloku	**Popis**
nefEqSystem	Blok pro strukturální popis systému (stochastické rovnice).
nefPDFSystem	Blok pro pravděpodobnostní popis systému (hustoty pravděpodobnosti).

Tabulka 5.4: Knihovní bloky skupiny systémů.

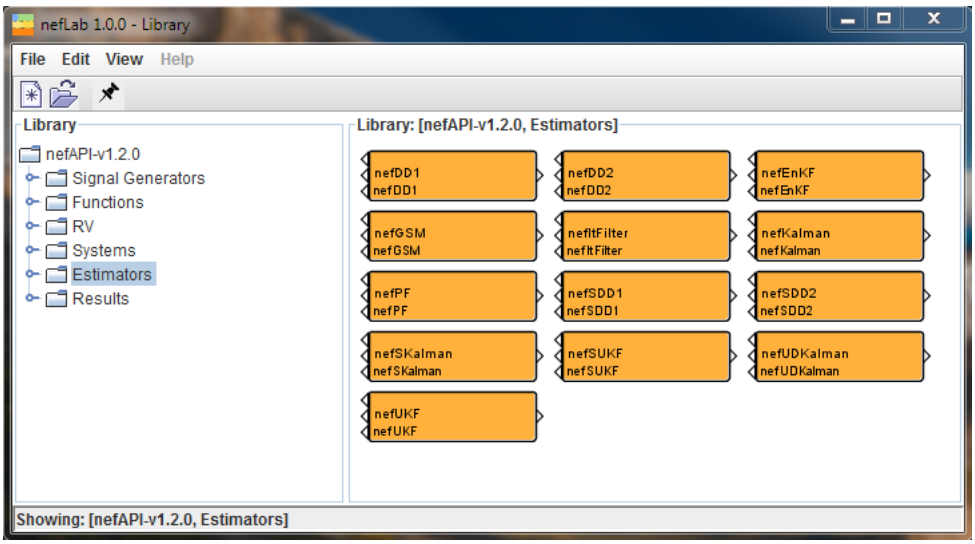
Je vhodné uvést, že ke každému systému se váže příslušná výkonná metoda *simulate*, která simuluje jeho vlastní trajektorii. Obrázek 5.4 ilustruje zobrazení systémů v knihovně *nefLab*.



Obrázek 5.4: Knihovna aplikace *nefLab* zobrazující komponenty systémů.

Skupina estimátorů

Skupina estimátorů nabízí následující komponenty uvedené v tabulce 5.5. Každý estimátor k sobě také váže výkonnou metodu *estimate*, která představuje jeho vlastní estimaci. Obrázek 5.5 ilustruje zobrazení estimátorů v knihovně *nefLab*.



Obrázek 5.5: Knihovna aplikace *nefLab* zobrazující komponenty estimátorů.

Seznam bloků skupiny estimátorů a jejich popis	
Název bloku	**Popis**
nefDD1	Divided Difference Estimator 1st Order.
nefDD2	Divided Difference Estimator 2nd Order.
nefEnKF	Extended Kalman Filter.
nefGSM	Gaussian Sum Filter.
nefItfilter	Iterated Kalman Filter.
nefKalman	(extended) Kalman Filter and Rauch-Tung-Striebel Smoother.
nefPF	Particle Filter.
nefSDD1	Square-Root Divided Difference Estimator 1st Order.
nefSDD2	Square-Root Divided Difference Estimator 2nd Order.
nefSKalman	Square-Root (Extended) Kalman Filter and Rauch-Tung-Striebel smoother - based on triangularization.
nefSUKF	Square-Root Unscented Kalman Estimator.
nefUDKalman	Square-Root (Extended) Kalman Filter - based on UD factorization.
nefUKF	Unscented Kalman Estimator.

Tabulka 5.5: Knihovní bloky skupiny estimátorů.

Skupina výsledků

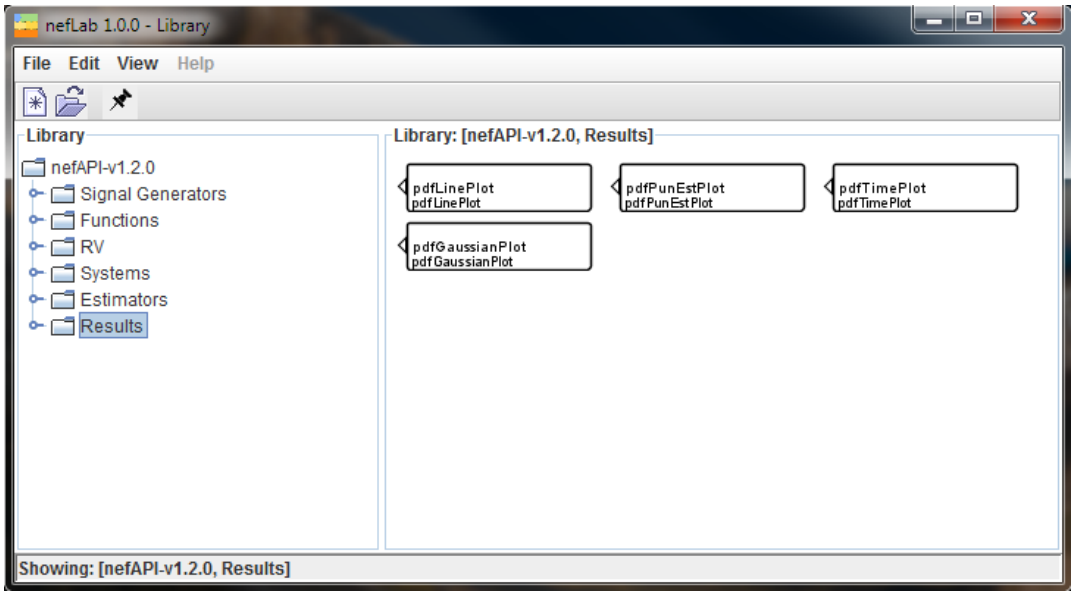
Skupinu výsledků reprezentují knihovní bloky, které jsou uvedeny v tabulce 5.6.

Seznam bloků skupiny výsledků a jejich popis	
Název bloku	**Popis**
pdfLinePlot	Blok vykreslí ze vstupních dat liniový graf. Jako vstupní data lze použít pouze reálná (prostá) data.
pdfPunEstPlot	Blok vykreslí ze vstupních dat graf bodových odhadů. Vstupní data jsou reprezentovány podtřídou třídy nefRV.
pdfTimePlot	Blok vykreslí ze vstupních dat graf časových posloupností hustot. Vstupní data jsou reprezentovány podtřídou třídy nefRV
pdfGaussianPlot	Blok vykreslí ze vstupních dat typu nefGaussianRV tzv. „klobouk“ hustoty normálního (Gaussovo) rozdělení pravděpodobnosti v požadovaném čase. Vstupní data jsou reprezentovány podtřídou nefGaussianRV třídy nefRV.

Tabulka 5.6: Knihovní bloky skupiny výsledků.

Obdobně jako u skupiny generátorů signálu, každý blok v této skupině byl vytvořen dodatečně tedy ne na základě popisu *API NEF*. Tyto bloky nejsou *NEF* poskytovány a byly implementovány navíc. Byly implementovány za účelem vizualizace výsledků estimačních experimentů, jelikož *NEF* sám o sobě poskytuje pouze hrubá data. Bloky byly vytvořeny pomocí sady vizualizačních metod, jejímž vytvořením se zabývala bakalářské práce [5]. Tato sada tedy byla převzata a využita v této diplomové práci k vizualizaci výsledků experimentů. Obrázek 5.6

ilustruje jejich zobrazení v knihovně *nefLab*.

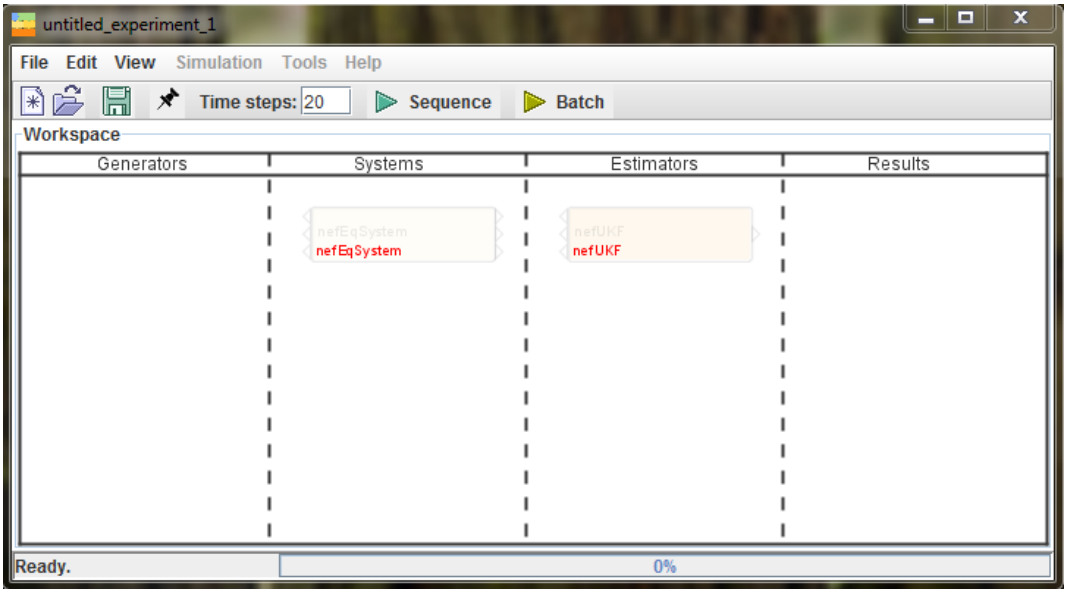


Obrázek 5.6: Knihovna aplikace *nefLab* zobrazující komponenty výsledků.

5.3 Vytvoření estimačních experimentů v aplikaci nefLab

5.3.1 Odhad stavu nelineárního Gaussovského systému užitím UKF

Uvažujme tedy stejný příklad (2.4.1) z kapitoly druhé. Systém je opět dán vztahy (2.4.1)-(2.4.5). Na rozdíl od příkladu (2.4.1), kde se v první řadě vytvářely stochastické funkce, zde začneme vytvořením nového experimentu buďto z menu nebo z panelu nástrojů. Jelikož je systém popsán stochastickými rovnicemi, je nutno využít blok *nefEqSystem*. Najdeme si tedy tento blok v knihovně systémů a přetáhneme ho na pracovní plochu nového estimačního experimentu. Blok bude po přetažení zprůhledněn a jméno příslušné třídy bude začervenalé, viz. obrázek 5.7. Takto je v aplikaci *nefLab* indikován každý neparametrizovaný blok. Po správné parametrizaci a přidání do experimentu se již blok vykreslí tak jak je uveden v knihovně.

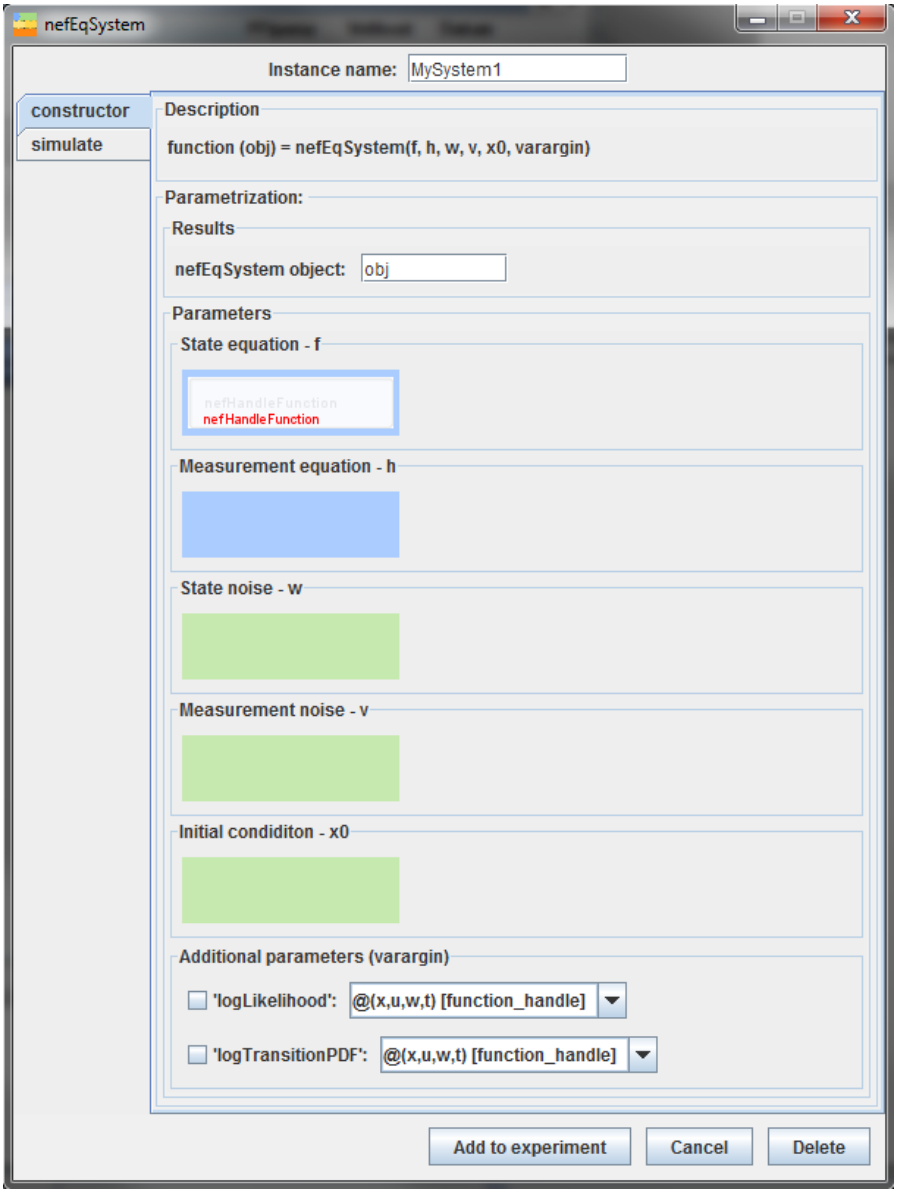


Obrázek 5.7: Nový estimační experiment s neparametrizovanými bloky.

Nyní již přejdeme k popisu systému pomocí stochastických funkcí. Vícerozměrná nelineární funkce reprezentující stavovou rovnici, která je v příkladu (2.4.1) specifikována zavedením příkazu

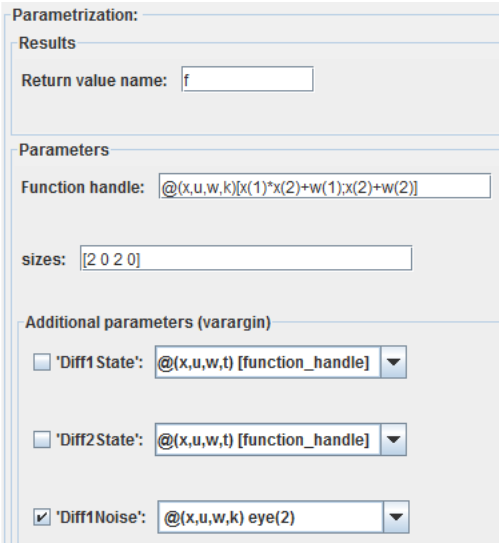
```
fFun = @(x,u,w,k)[x(1)1*x(2)+w(1);x(2)+w(2)]
f = nefHandleFunction(fFun, [2 0 2 0], 'diff1Noise', @(x,u,w,k) eye(2));
```

bude v aplikaci nefLab vytvořena přetažením bloku *nefHandleFunction* z knihovny funkcí do bloku *nefEqSystem* v okně experimentu na místo stavového popisu (*State equation*). Obrázek 5.8 ilustruje aktuální stav vytváření estimačního experimentu.



Obrázek 5.8: Parametrizace bloku *nefEqSystem*.

Nyní kliknutím na blok *nefHandleFunction* otevřeme parametrizační okno. Do textového pole u parametru *Function handle* zadáme anonymní funkci ve tvaru `@(x,u,w,k)[x(1)*x(2)+w(1);x(2)+w(2)]`, která reprezentuje stavovou rovnici. Dále parametrizujeme parametr *sizes* zadáním dimenze stavu, tj. `[2 0 2 0]`. Dále zaškrtneme volitelný parametr *'diff1Noise'*, který reprezentuje první derivaci nelineární funkce se zřetelem na šum w_k a jako jeho hodnotu uvedeme `@(x,u,w,k) [1 0;0 1]` nebo pokud si chceme usnadnit práci, můžeme využít matlabovské funkce `@(x,u,w,k) eye(2)`, příkazy jsou z pohledu *Matlabu* ekvivalentní. Nezapomene dát vytvářenému objektu jméno tedy `f` a parametrizace bloku je u konce. Dále je možno definovat vlastní název instance. V tomto případě ho ale ponecháme na výchozí hodnotě. Stiskem tlačítka *Add to system* přidáme objekt *nefHandleFunction* jako popis stavu systému *nefEqSystem* a blok se vykreslí jako parametrizovaný.

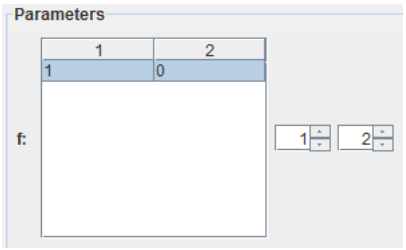


Obrázek 5.9: Parametrizace bloku *nefHandleFunction*.

Rovnice měření je lineární proto využijeme bloku *nefLinFunction*. Najdeme si tento blok v knihovně a přetáhneme ho do systému na místo pro rovnici měření (*Measurement equation*). Příkaz k tvorbě rovnice měření, který je v příkladu 2.4.1 uveden takto

```
H = [1 0];
h = nefLinFunction(H, [], 1);
```

nahradíme parametrizací bloku *nefLinFunction* zadáním příslušných hodnot do připravených matic. Pouze pomocí tzv. *spinnerů* nastavíme dimenzi matic a dosadíme hodnoty vektorů. Matici `g` necháme prázdnou. Tím se tedy popsalo, že rovnice nemá žádné řízení a nebude přítomný šum. Opět nezapomene dát vytvářenému objektu jméno tedy `h` a parametrizace bloku je u konce. Stiskem tlačítka *Add to system* přidáme objekt *nefLinFunction* jako popis měření systému *nefEqSystem* a blok se vykreslí jako parametrizovaný.



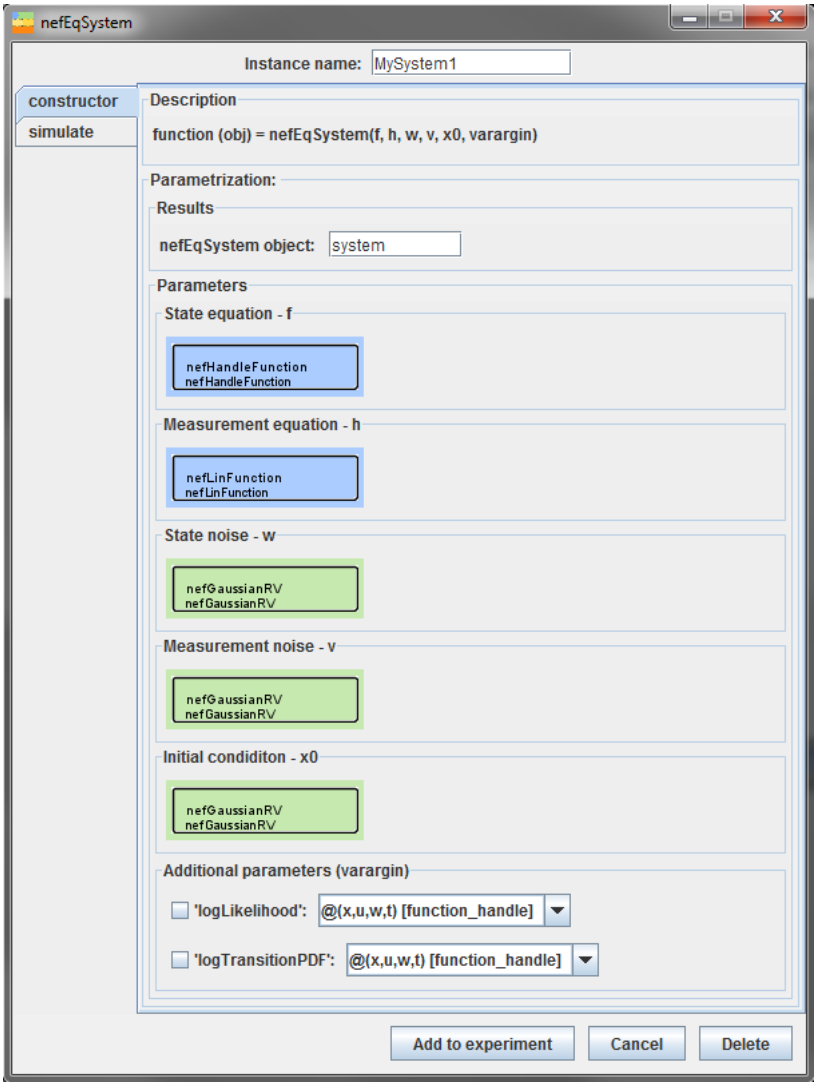
Obrázek 5.10: Parametrizace bloku *nefLinFunction*.

Stejně jako v příkladu 2.4.1 je pak dalším krokem popis náhodné veličiny w_k, v_k a x_0. Každá z nich bude uvažována jako gaussovská a proto budou využity bloky *nefGaussianRV*. Tento blok je parametrizován pomocí předem připravených matic. Stačí opět nastavit příslušný počet dimenzí a dosadit hodnoty. Po dosazení hodnot nezapomene dát vytvářeným objektům jména tedy `w` , `v` a `x0`. Zopakujme příkazy z příkladu 2.4.1.

```
w = nefGaussianRV([0 0]', eye(2)*0.05);
v≠ nefGaussianRV(0, 0.01);
x0 = nefGaussianRV([0.9; −0.85], 1e−1*eye(2));
```

Dále pojmenujeme vytvářený objekt systému *system*. **Zdůrazněme, že názvy objektů slouží ke vzájemnému grafickému provázání bloků na pracovní ploše experimentu a ke korektnímu předávaní dat v samotném estimačním experimentu, který je vykonán v programovém prostředí *Matlab***. Po tomto kroku je blok (objekt) *nefEqSystem* plně parametrizován a přejdeme k definici jeho výkonné metody *simulate*, která tento blok reprezentuje. Obrázek 5.11 ilustruje aktuální stav parametrizace bloku systému *nefEqSystem*. Pro porovnání s příkladem 2.4.1 je uveden příkaz, který koresponduje s obrázkem 5.11.

```
system = nefEqSystem(f, h, w, v, x0);
```

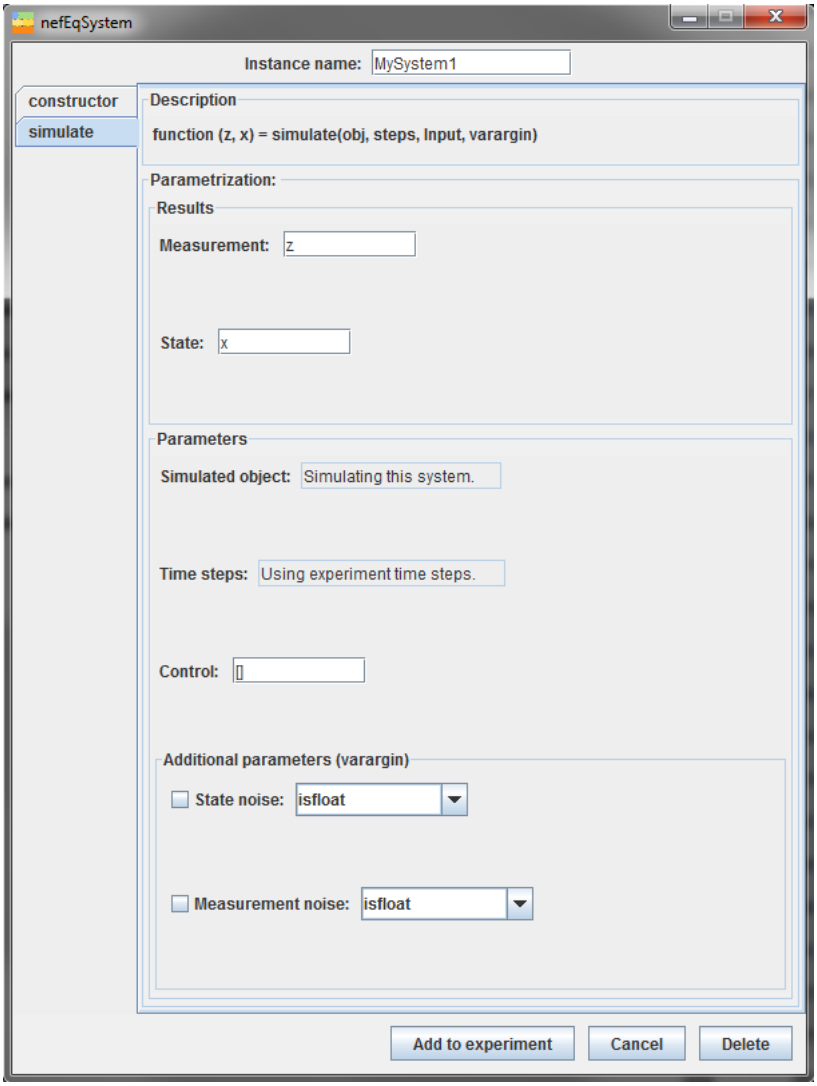


Obrázek 5.11: Parametrizace bloku *nefEqSystem*.

Pro definici výkonné metody se v systému přepneme kliknutím na záložku s názvem *simulate*.

Systém bude simulován na časovém horizontu, který můžeme nastavit v panelu nástrojů (*Time steps*) v okně experimentu, viz. obrázek 5.7. Parametr řízení vyplníme prázdným vektorem `[]`, tzn. že popisujeme autonomní systém, který nemá vstup a tedy na vstup zadáváme prázdný vektor. Volitelné parametry zde nebudou využity. Výstupní parametry *Measurement* nastavíme na hodnotu `z` a *State* nastavíme na `x`. Do těchto proměnných jsou pak uloženy posloupnosti měření systému a stavu. Stisknutím tlačítka *Add to experiment* pak dokončíme parametrizaci bloku systému *nefEqSystem*. Následující příkaz z 2.4.1 odpovídá obrázku 5.12.

```
nSteps = 20;
[z,x] = simulate(system, nSteps, []);
```

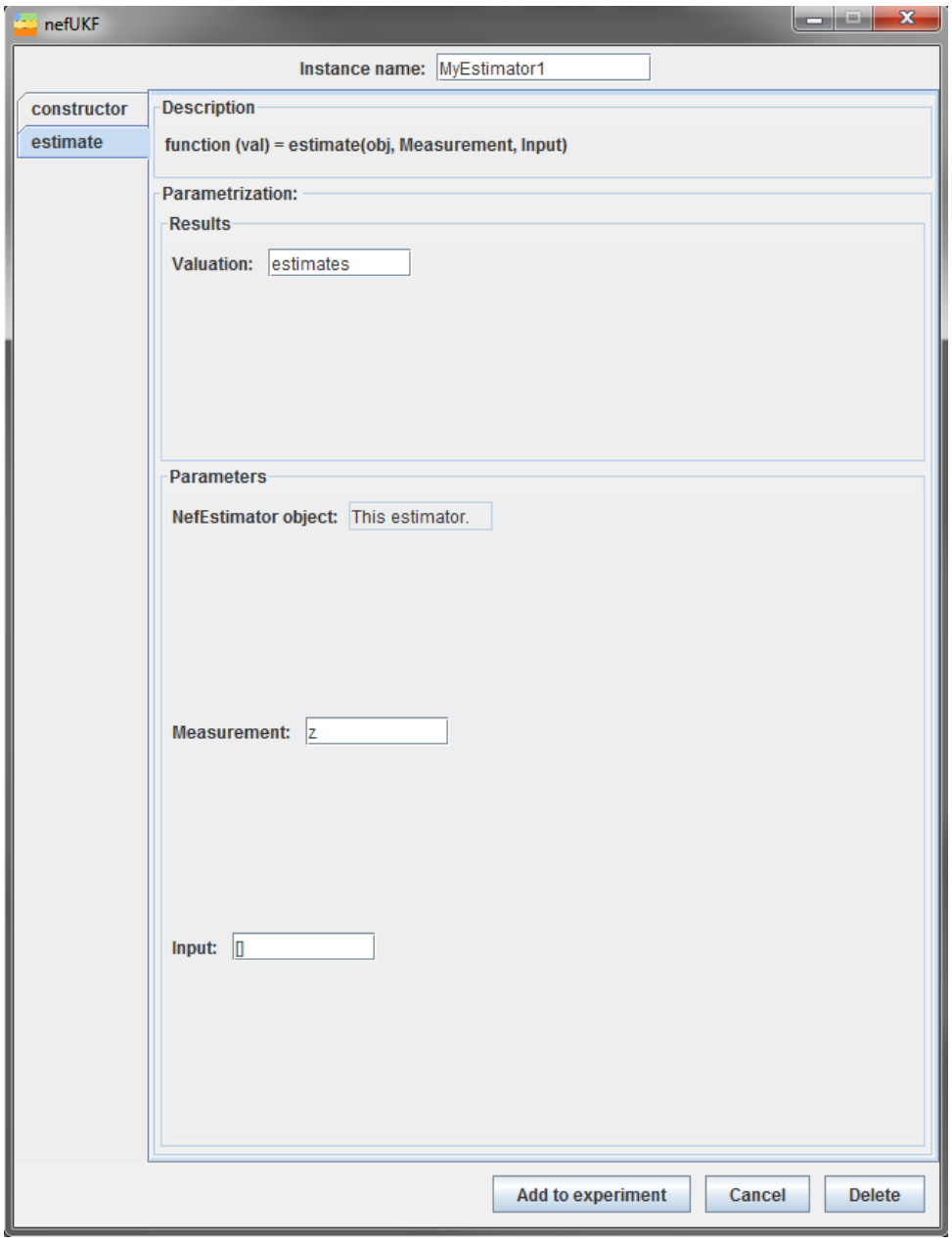


Obrázek 5.12: Parametrizace bloku *nefEqSystem* - metoda *simulate*.

Konečně lze přejít k samotnému odhadu stavu systému. Stejně jako v příkladu 2.4.1 byl vybrán unscentovaný Kalmanův filtr. V knihovně najdeme skupinu estimátorů a přetáhneme z ní blok *nefUKF* do experimentu. Otevřeme si parametrizační okno tohoto bloku a vstupní parametr *system* vyplníme názvem systému, který chceme odhadovat tedy hodnota `system`. Volitelné

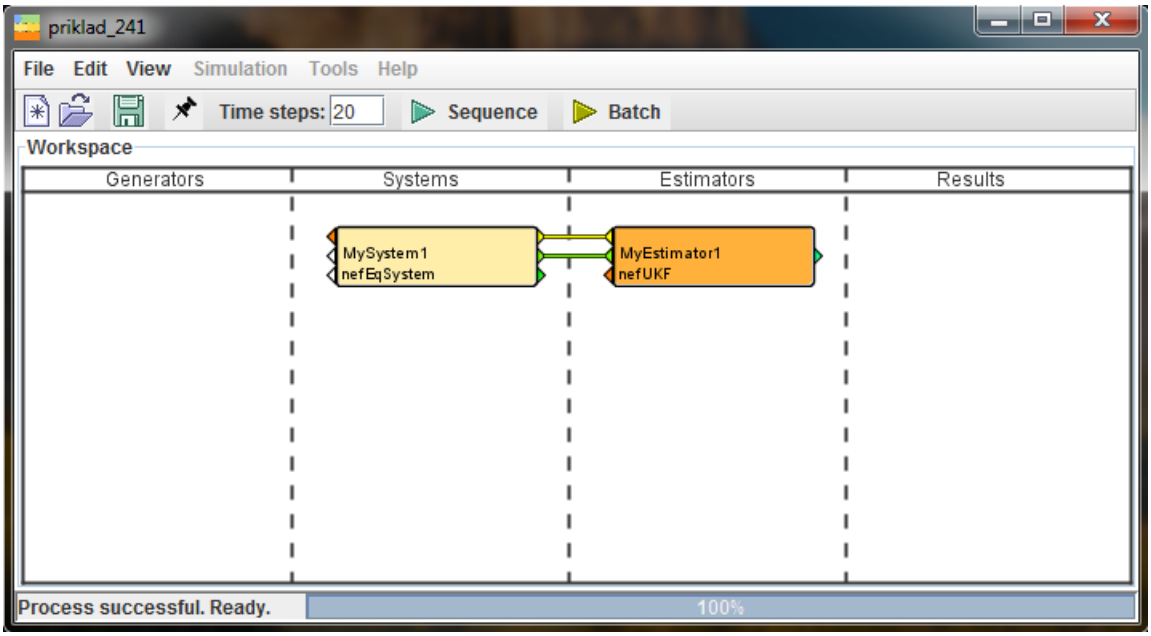
parametry necháme bez povšimnutí a vyplníme ještě hodnotu výstupního parametru hodnotou `UKF`, tj. název objektu estimátoru. Dále se přepneme pomocí záložky na metodu *estimate*, kde vyplníme vstupní parametry *Measurement* a *Input* hodnotami `z` (měření systému poskytnuté od metody *simulate*) a prázdným vektorem `[]` (externí vstup estimátoru). Výstupní parametr *Valuation* určuje do jaké proměnné chceme ukládat výsledky filtračního odhadu, tj. `estimates`. Následující příkaz z 2.4.1 odpovídá obrázku 5.13.

```
UKF = nefUKF(system);
estimates = estimate(UKF, z, []);
```



Obrázek 5.13: Parametrizace bloku *nefUKF* - metoda *estimate*.

Tímto jsme kompletně popsali experiment a můžeme přejít k jeho výpočtu. Obrázek 5.14 ilustruje aktuální stav experimentu.



Obrázek 5.14: Experiment s parametrizovanými bloky těsně po provedení výpočtu.

Poznamenejme, že bychom mohli využít bloků výsledků k analyzování výsledku estimačního experimentu. Dále je možno experiment uložit a opět načíst či exportovat do *m-skriptu* nebo exportovat data programového prostředí *Matlab* do *mat-filu*. To vše lze kontrolovat buďto z menu nebo panelu nástrojů okna experimentu.

Výpočet experimentu zahájíme jedním z tlačítek play z panelu nástrojů. Zelené tlačítko *Sequence* pošle experiment programovému prostředí *Matlab* sekvenčně příkaz po příkazu. Tento způsob je vhodný například pro ladění experimentu, kdy můžeme kontrolovat postupně průběh experimentu. Zatímco tmavě žluté tlačítko *Batch* pošle celý experiment k výpočtu programovému prostředí *Matlab* na jednou tedy dávkově. Tento způsob je vhodný použít po odladění experimentu, jelikož již víme, že experiment je v pořádku připraven k výpočtu. Také je tento způsob výhodný v rychlosti výpočtu experimentu, jelikož dávkové zpracování se provede znatelně v kratším čase. Výsledky obou způsobů zpracování experimentu jsou ale samozřejmě totožné. Po stisknutí příslušného tlačítka se spustí na pozadí programové prostředí *Matlab* a aplikace ho vyzve k provedení výpočtu. Pokud již byl předtím aplikací *nefLab* spuštěn, aplikace naváže spojení s předchozím spuštěným procesem a opět ho vyzve k provedení výpočtu.

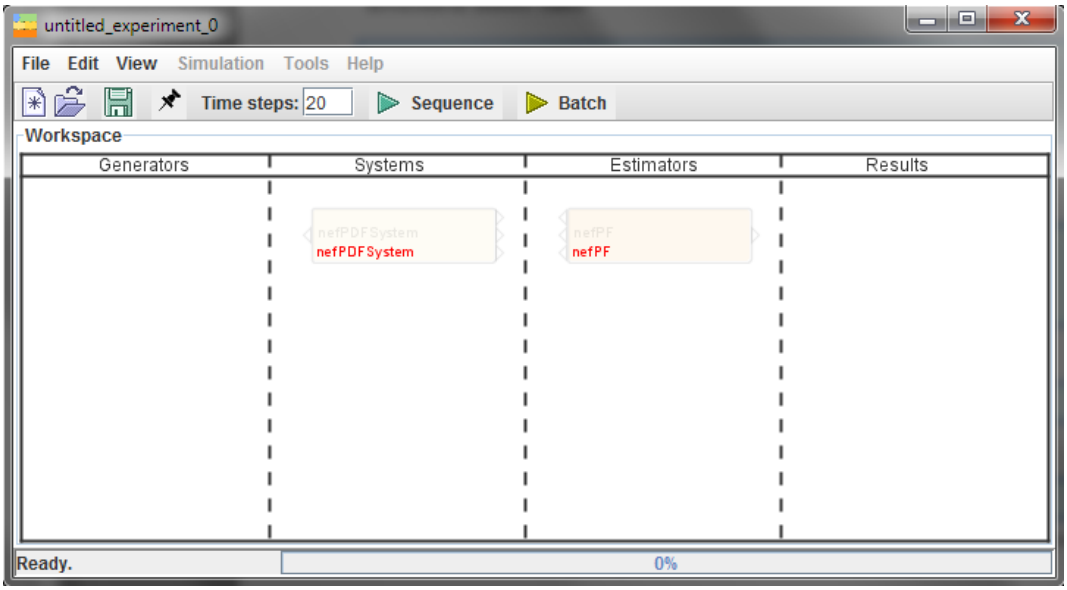
Následuje ukázka vygenerovaného *m-skriptu* příkladu 2.4.1 aplikací *nefLab*.

Výpis 5.1: Vygenerový *m-skript* příkladu 2.4.1 aplikací *nefLab*.

```
clc
clear all
close all
time_steps = 20
f = nefHandleFunction(@(x,u,w,k)[x(1)*x(2)+w(1);x(2)+w(2)], [2 0 2 0],...
'Diff1Noise', @(x,u,w,k) eye(2))
h = nefLinFunction([ 1  0 ], [], [ 1 ])
w = nefGaussianRV([ 0 ; 0 ], [ 0.05  0 ; 0  0.05 ])
v≠ nefGaussianRV([ 0 ], [ 0.01 ])
x0 = nefGaussianRV([ 0.9 ; −0.85 ], [ 0.01  0 ; 0  0.01 ])
system = nefEqSystem(f, h, w, v, x0)
[z, x] = simulate(system, time_steps, [])
UKF = nefUKF(system)
t = cputime;
[estimates] = estimate(UKF, z, [])
estimatestime = cputime−t;
t = [1:time_steps];
for i = 1:time_steps
xest_estimates(:,i) = evalMean(estimates{i});
msem_estimates(:,i) = (xest_estimates(:,i)−x(:,i)).^2;
end
fprintf('Stats : MSEM\t\t time\n');
fprintf('UKF   : %f\t%f\n',mean(mean(msem_estimates)),estimatestime);
```

5.3.2 Filtrace nelineárního gaussovského systému

Nyní přejděme k realizaci příkladu (2.4.2) filtrace nelineárního gaussovského systému v aplikaci *nefLab*. Zopakujme, že tento příklad představuje problém sledování trajektorie objektu. Dynamika stavu je popsána přechodovou podmíněnou hustotou pravděpodobnosti (2.4.6). A odpovídající podmíněná hustota pravděpodobnosti měření je dána vztahem (2.4.7). Z těchto vztahů je zřejmé, že je potřeba využít blok pro stochastický popis systému, tj. *nefPDFSystem*. Nejdříve, ale začneme vytvořením nového experimentu. To je možno buďto z menu nebo z panelů nástrojů aplikace *nefLab*. V knihovně si najdeme příslušný blok a přetáhneme ho z panelu na pracovní plochu experimentu. Blok bude po přetažení zprůhledněn a jméno třídy (*nefPDFSystem*) začervenalé, viz. obrázek 5.15. Takto je v aplikaci *nefLab* indikován každý neparametrizovaný blok. Po správné parametrizaci se již blok vykreslí tak jak je uveden v knihovně.



Obrázek 5.15: Nový estimační experiment s neparametrizovanými bloky.

Citujme nyní větu z příkladu (2.4.2). Přechodová hustota pravděpodobnosti a hustota pravděpodobnosti měření jsou obě podmíněnými hustotami pravděpodobnosti, které je možné vyjádřit tak, že jejich střední hodnota je dána přesně danou funkcí a jejich rozptyl pomocí kovarianční matice takto

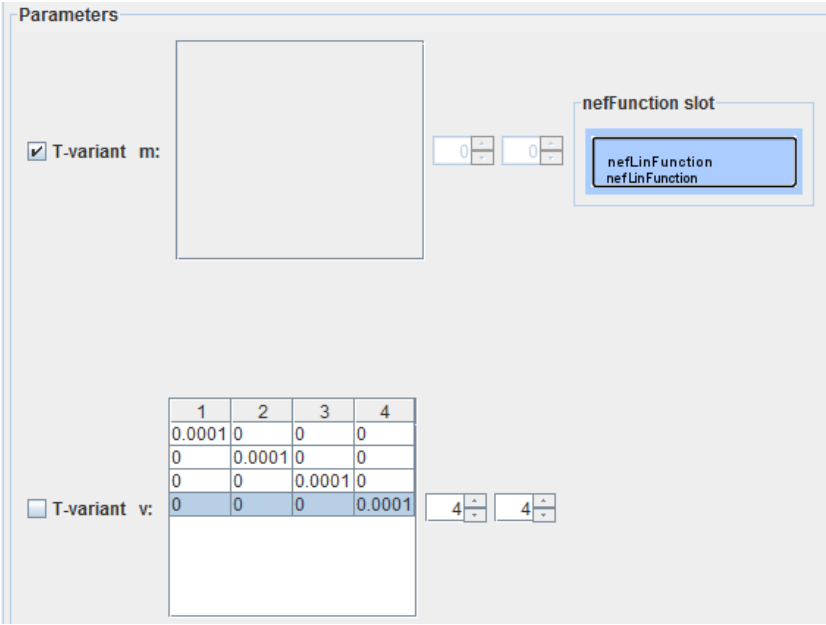
```
F = [1 1 0 0; 0 1 0 0; 0 0 1 1; 0 0 0 1];
xMean = nefLinFunction(F, [], []);
xVariance = 0.0001*eye(4);
xPdf = nefGaussianRV(xMean, xVariance);
```

Otevřeme si parametrizační okno bloku *nefPDFSystem*, vyhledáme si v knihovně blok *nefGaussianRv* a na místo pro přechodovou hustotu pravděpodobnosti (*Transition pdf*) vložíme z knihovny tento blok. Otevřeme si jeho parametrizační okno. Zde uvidíme již známé komponenty pro plnění matic a u každé z nich také *CheckBox* s názvem *T-Variant*. Jelikož chceme střední hodnotu přechodové hustoty pravděpodobnosti vyjádřit funkcí zaškrtneme *CheckBox* u první matice. Tím se zobrazí modrý slot pro vkládání funkcí vedle matice, která se touto akcí zablokuje. V knihovně

si najdeme blok *nefLinFunction* a vložíme ho do modrého slotu.

Rozklikneme parametrizační okno naposled vloženého bloku *nefLinFunction*. A parametrizujeme ho již známým způsobem z příkladu (5.2.1). Tedy pomocí *spinnerů* nastavíme dimenze příslušné matice a doplníme hodnoty do matic. První matice (f) bude matice čtyři na čtyři a další dvě (g a h) pak zůstanou prázdné. Přesně jako v naposled uvedeném příkazu. Pojmenujeme objekt *nefLinFunction* `xMean`. Stiskneme tlačítko *Add to RV* a parametrizace střední hodnoty je u konce.

Zbývá parametrizovat kovarianční matici představující rozptyl v bloku *nefGaussianRV*. Nastavíme matici dimenze čtyři a zadáme hodnoty `0.0001` na její hlavní diagonálu a zbytek doplníme nulami. Pojmenujeme vytvářený objekt `xPDF` a stiskneme tlačítko *Add to sytem*. Parametrizace přechodové hustoty pravděpodobnosti je u konce. Aktuální stav parametrizace bloku *nefGaussianRV* je ilustrována na obrázku 5.16.



Obrázek 5.16: Parametrizace bloku *nefGaussianRV*.

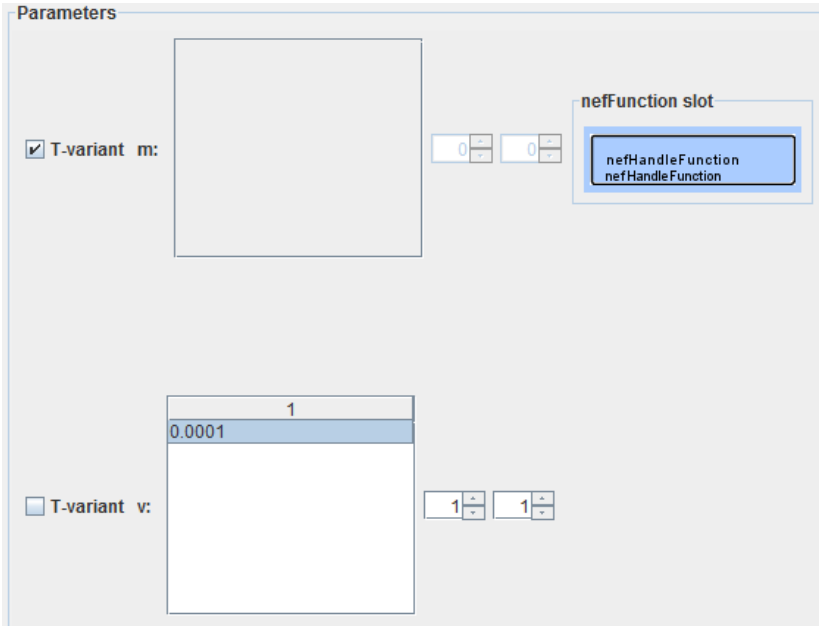
Obdobně tedy i pro hustotu pravděpodobnosti měření vyjádříme její střední hodnotu přesně danou funkcí a její rozptyl pomocí kovarianční matice následovně

```
mFun = @(x,u,v,t) atan(x(3)/x(1));
zMean = nefHandleFunction(mFun, [4 0 0 0]);
zVariance = 0.0001;
zPdf = nefGaussianRV(zMean, zVariance);
```

Přetáhneme z knihovny blok *nefGaussianRV* na místo pro hustotu pravděpodobnosti měření (*Measurement pdf*). Rozklikneme tento blok a parametrizujeme. Zaškrtneme *CheckBox* s názvem *T-Variant* u první matice. Najdeme si v knihovně tentokrát blok *nefHandleFunction* a vložíme ho do nově zobrazeného modrého slotu pro funkce. Rozklikneme pro parametrizační okno. Obdobným postupem jako v příkladu (5.2.1) parametrizujeme tento blok, tj. u parametru *Function handle* uvedenem hodnotu `@(x,u,v,t) atan(x(3)/x(1))`. U parametru *sizes* uvedeme vek-

tor `[4 0 0 0]`. Pojmenujeme objekt `zMean` a stiskneme tlačítko *Add to RV*. Tím jsme úspěšně zavedli střední hodnotu hustoty pravděpodobnosti měření.

Kovarianční matici parametrizujeme nastavením druhé matice na dimenzi jedna. Doplníme hodnotu `0.0001` a pojmenujeme objekt *nefGaussianRV* `zPdf`. Stiskneme tlačítko *Add to system* a parametrizace hustoty pravděpodobnosti měření je u konce. Aktuální stav parametrizace bloku *nefGaussianRV* je ilustrována na obrázku 5.17.



Obrázek 5.17: Parametrizace bloku *nefGaussianRV*.

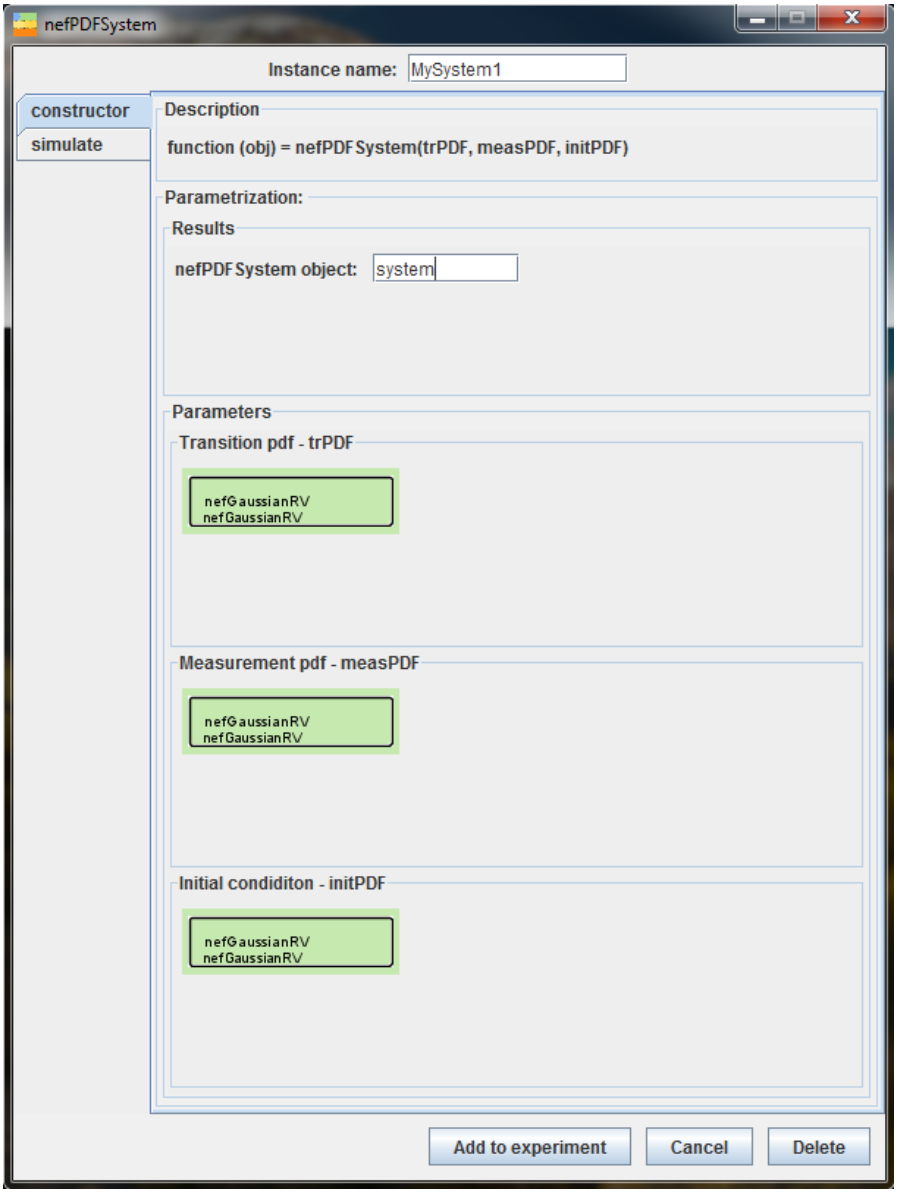
Počáteční podmínka je v příkladu (2.4.2) specifikována jako

```
x0Pdf = nefGaussianRV([−0.05 0.001 2 −0.055]', 0.01*eye(4));
```

To znamená klasickým způsobem najít blok *nefGaussianRV* a přetáhnout ho do bloku *nefPDF-System* na místo pro počáteční podmínku (*Initial condition*). Rozkliknout právě vložený blok a parametrizovat. První matici nastavíme na čtyři řádky a jeden sloupec a doplníme hodnoty `[-0.05 0.001 2 -0.055]`. Druhou matici pak nastavíme na čtyři řádky a čtyři sloupce a opět doplníme hodnoty. Na hlavní diagonále bude hodnota `0.01` a zbytek nula. Pojmenujeme objekt `x0Pdf` a stiskneme tlačítko *Add to system*. Tímto je parametrizace počáteční podmínky u konce.

Zbývá pojmenovat objekt parametrizovaného systému `system`. **Opět zdůrazněme, že názvy objektů slouží ke vzájemnému grafickému provázání bloků na pracovní ploše experimentu a ke korektnímu předávaní dat v samotném estimačním experimentu, který je vykonán v programovém prostředí *Matlab***. Obrázek 5.18 ilustruje aktuální stav parametrizace systému a koresponduje s následujícím příkazem z příkladu (2.4.2).

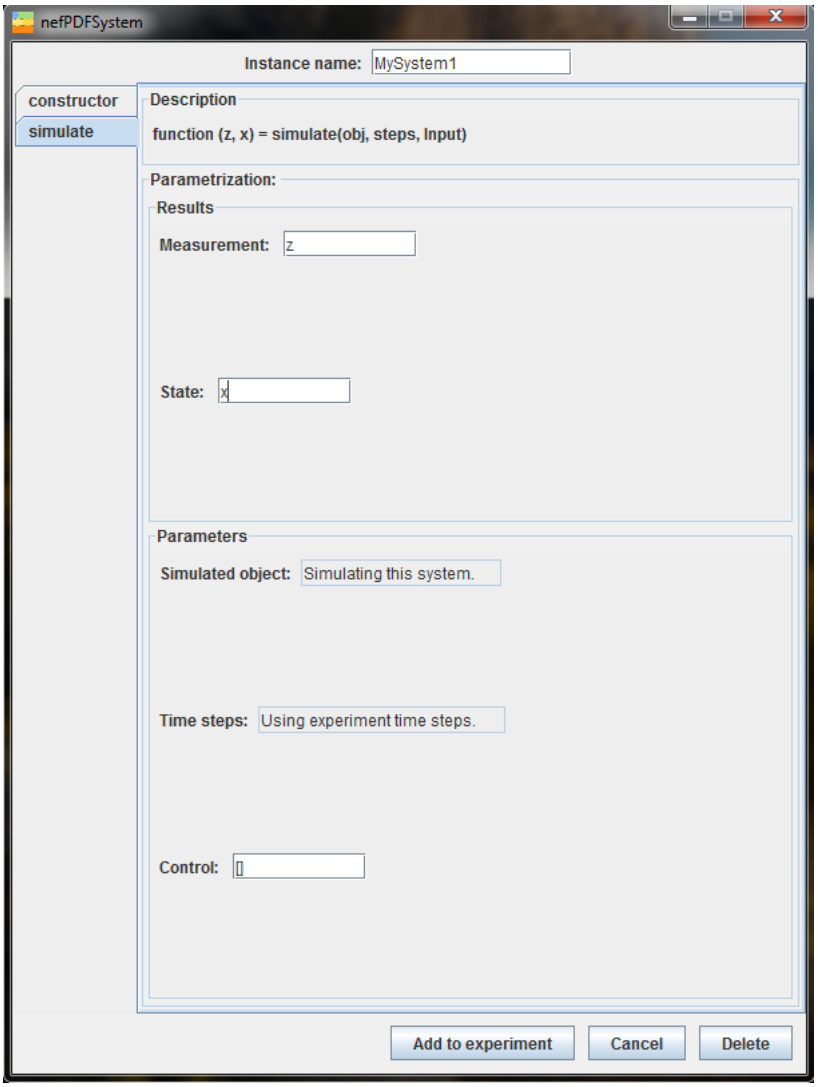
```
system = nefPDFSystem(xPdf, zPdf, x0Pdf);
```



Obrázek 5.18: Parametrizace bloku *nefPDFSystem*.

Můžeme přejít k parametrizaci jeho metody *simulate*. Přepneme se pomocí záložky *simulate* do parametrizačního okna této metody a parametrizujeme. Časový horizont na kterém bude systém simulován nastavíme v panelu nástrojů (*Time steps*) okna experimentu. Vstupní parametr řízení (*Control*) vyplníme prázdným vektorem `[]`, tj. nebude použito externího řízení. Výstupní parametr *Measurement* nastavíme na `z`. Do této proměnné bude ukládána posloupnost měření systému. Další výstupní parametr *State* nastavíme na `x`. Do této proměnné bude ukládána posloupnost stavů systému, kterou můžeme použít pro pozdější porovnání s odhadnutými stavy estimátoru. Nyní dokončíme parametrizaci bloku systému *nefPDFSystem* stisknutím tlačítka *Add to experiment*. Následující příkaz z příkladu (2.4.2) koresponduje s aktuálním stavem experimentu.

```
nSteps = 20;
[z,x] = simulate(system, nSteps, []);
```

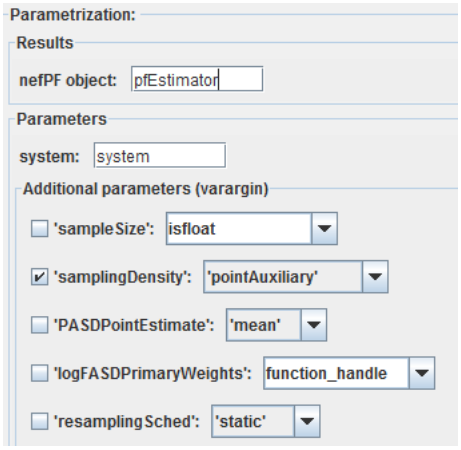


Obrázek 5.19: Parametrizace bloku *nefPDFSystem* - metoda *simulate*.

Pomocí uloženého měření z^k docílíme odhad stavu x_k, užitím globálního ho filtru. Najdeme si tedy v knihovně estimátorů blok *nefPF* a vložíme ho na pracovní plochu experimentu. Klikem na blok zobrazíme parametrizační okno. Zde vyplníme jméno objektu estimátoru `pfEstimator`. Vstupním parametrem `system` určíme nad jakým systémem bude vykonán odhad, tj. `system`. Zaškrtnutím volitelného parametru *'samplingDensity'* a jeho nastavením na hodnotu `'pointAuxiliary'` nastavíme vzorkovací hustotu. Jelikož zde opět není uvedený volitelný parametr *'taskType'* (volba typu úlohy odhadu) je použita jeho výchozí hodnota, tj. *'filtering'*. Rozdíl mezi časovými okamžiky k a l je počátečně nastaven na nulu. Poznamenejme, že nebylo použito dalších parametrů souvisejících s globálním filtrem, tudíž se pro ně použily výchozí hodnoty. Obrázek 5.20 ilustruje aktuální parametrizaci bloku estimátoru *nefPF*, se kterým koresponduje následující

příkaz z příkladu (2.4.2).

```
pfEstimator = nefPF(system, 'samplingDensity', 'pointAuxiliary');}
```

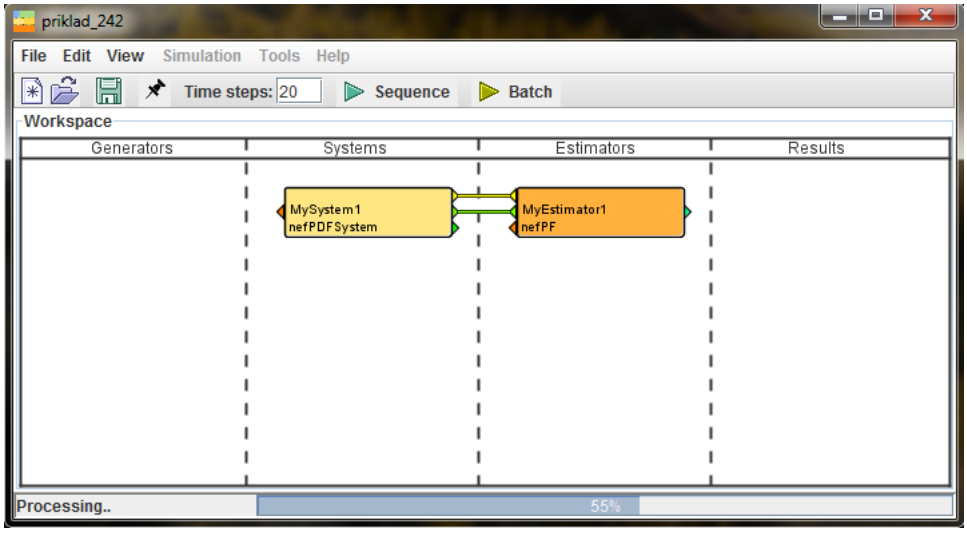


Obrázek 5.20: Parametrizace bloku estimátoru *nefPF*.

K vlastní estimaci přistoupíme opět záložkou *estimate*. Vstupní parametry měření *Measurement* nastavíme na `z` a externí vstup *Input* nastavíme na prázdný vektor `[]`. Výstupní parametr *Valuation* vyplníme hodnotou `estimates`. Do této proměnné jsou uloženy výsledné hodnoty odhadu filtrační hustoty pravděpodobnosti $p(x_k|z^k)$, která je dána empirickou hustotou pravděpodobnosti *nefEmpiricalRV*. Tomuto odpovídá příkaz z příkladu (2.4.2).

```
estimates = estimate(pfEstimator, z, []);
```

Tímto jsme kompletně popsali experiment a můžeme přejít k jeho výpočtu. Obrázek 5.21 ilustruje aktuální stav experimentu.



Obrázek 5.21: Experiment s parametrizovanými bloky během provádění výpočtu.

Poznamenejme, že bychom mohli využít bloků výsledků k analyzování výsledku estimačního experimentu. Dále je možno experiment uložit a opět načíst či exportovat do *m-skriptu* nebo exportovat data programového prostředí *Matlab* do *mat-filu*. To vše lze kontrolovat buď to z menu nebo panelu nástrojů okna experimentu.

Výpočet experimentu zahájíme jedním z tlačítek play z panelu nástrojů. Zelené tlačítko *Sequence* pošle experiment programovému prostředí *Matlab* sekvenčně příkaz po příkazu. Tento způsob je vhodný například pro ladění experimentu, kdy můžeme kontrolovat postupně průběh experimentu. Zatímco tmavě žluté tlačítko *Batch* pošle celý experiment k výpočtu programovému prostředí *Matlab* na jednou tedy dávkově. Tento způsob je vhodný použít po odladění experimentu, jelikož již víme, že experiment je v pořádku připraven k výpočtu. Také je tento způsob výhodný v rychlosti výpočtu experimentu, jelikož dávkové zpracování se provede znatelně v kratším čase. Výsledky obou způsobů zpracování experimentu jsou ale samozřejmě totožné. Po stisknutí příslušného tlačítka se spustí na pozadí programové prostředí *Matlab* a aplikace ho vyzve k provedení výpočtu. Pokud již byl předtím aplikací *nefLab* spuštěn, aplikace naváže spojení s předchozím spuštěným procesem a opět ho vyzve k provedení výpočtu.

Následuje ukázka vygenerovaného *m-skriptu* příkladu 2.4.2 aplikací *nefLab*.

Výpis 5.2: Vygenerovaný *m-skript* příkladu 2.4.2 aplikací *nefLab*.

```
clc
clear all
close all
time_steps = 20
xMean = nefLinFunction([ 1  1  0  0 ; 0  1  0  0 ; 0  0  1  1 ; 0  0  0  1 ],[],[])
xPDF = nefGaussianRV(xMean,[ 0.0001  0  0  0 ; 0  0.0001  0  0 ; 0  0  0.0001  0 ;
0  0  0  0.0001 ])
zMean = nefHandleFunction(@(x,u,v,t) atan(x(3)/x(1)), [4 0 0 0])
zPdf = nefGaussianRV(zMean, [ 0.0001 ])
x0Pdf = nefGaussianRV([ -0.05 ; 0.001 ; 2 ; -0.055 ], [ 0.01  0  0  0 ;
0  0.01  0  0 ; 0  0  0.01  0 ; 0  0  0  0.01 ])
system = nefPDFSystem(xPDF, zPdf, x0Pdf)
[z, x] = simulate(system, time_steps, [])
pfEstimator = nefPF(system, 'samplingDensity', 'pointAuxiliary')
t = cputime;
[estimates] = estimate(pfEstimator, z, [])
estimatestime = cputime-t;
t = [1:time_steps];
for i = 1:time_steps
xest_estimates(:,i) = evalMean(estimates{i});
msem_estimates(:,i) = (xest_estimates(:,i)-x(:,i)).^2;
end
fprintf('Stats : MSEM\t\t time\n');
fprintf('pfEstimator   : %f\t%f\n',mean(mean(msem_estimates)),estimatestime);
```

5.4 Shrnutí

Tato kapitola podrobně představila na dvou poměrně ještě jednoduchých příkladech uvedených v druhé kapitole, jak se v aplikace vytváří estimační experimenty. Poznamenejme, že při vytváření experimentů nebyly využity bloky generátorů signálu a výsledků. Tedy aplikace toho nabízí ještě více než bylo pro popis těchto dvou příkladů nutné.

Bylo ukázáno, že práce s aplikací je intuitivní a přirozená. Pro uživatele, který přišel s aplikací do styku poprvé může být problematická orientace v otevřených oknech při vytváření estimačního experimentu. Po chvilce používání aplikace už by toto nemělo představovat výrazný problém.

Kapitola 6

Závěr

Cílem této diplomové práce bylo vytvoření aplikace v programovacím jazyce *Java* sloužící jako podpora pro *toolbox NEF* za účelem tvoření komplexních estimačních experimentů s následnou vizualizací poskytnutých výsledků. K tomuto hlavnímu cíli se vázaly další dílčí úkoly. Vytvořit dostatečně univerzální nadstavbu popisující *NEF* a tuto nadstavbu poté použít v aplikaci k implementaci dílčích aspektů experimentu. Dále vhodně spravovat experimenty pomocí konfiguračních souborů experimentů.

Diplomová práce se v úvodu zabývala teorií nelineárního odhadu stavu a popisem *toolboxu NEF* a jeho stěžejních komponent. Zejména popis *toolboxu NEF* byl pro čtenáře důležitý z hlediska porozumění dalším cílům práce. Tento bod byl naplněn v kapitole druhé. Tato kapitola důkladně popsala vše důležité z hlediska porozumění *toolboxu NEF*.

Opěrným bodem pro vytvoření aplikace *nefLab* byl právě jeden z kladených požadavků a to vytvořit dostatečně univerzální popis *API toolboxu NEF*. Tento úkol byl zpracován velmi důsledně, jelikož použitá syntaxe myslí i na budoucí vývoj *toolboxu NEF*, tudíž je aplikace schopna dobře reagovat na změny v *NEF*. S tím pak dále souvisí i výsledný matlabovský *m-skript* estimačního experimentu poskytnutý aplikací *nefLab*, který bude stále odpovídat požadavkům *toolboxu NEF*. Neméně důležité pak bylo popsat vytvořený estimační experiment v aplikaci konfiguračním souborem. Pomocí tohoto souboru je pak realizována správa estimačních experimentů. Tento záměr byl naplněn a detailně popsán stejně jako předešlý uvedený úkol v kapitole třetí.

Na tomto základě se pak práce mohla přesunout k samotné implementaci aplikace. Klíčové aspekty z hlediska implementace aplikace jsou rozebrány v kapitole čtvrté. Tato kapitola shrnuje řešení prvotních problémů implementace aplikace *nefLab*. Jednak je tu popsáno jakým způsobem bylo přistoupeno k návrhu datové struktury a tvoření objektů představující jednotlivé komponenty *NEF* z popisu *API NEF*. Dále je tu také předvedeno jak aplikace využívala programového prostředí *Matlab* k výpočtu estimačních experimentů.

Aplikace *nefLab* se snaží jít ve stopách *toolboxu NEF* a uživateli nabízí navržení a jednoduché vykonání komplexních estimačních experimentů i s jejich následnou vizualicí. Zdůrazněme, že aplikace *nefLab* využívá k vizualizaci estimačních experimentů sady vizualizačních metod převzatých z nástroje *nefGUI*, vytvořeného autorem této práce. Účelem tohoto nástroje je komplexní tvorba vizualizací výsledků pro *toolbox NEF*. Podrobně se tímto nástrojem zabývá bakalářská práce [5]. V případě, že by uživatel požadoval pokročilejší vizualizaci výsledků estimačních experimentů, aplikace *nefLab* je schopna předat výsledná hrubá data pomocí matlabovského souboru *mat-file* tomuto podpůrnému nástroji, který uživateli poskytuje širší spektrum možností k vizualizaci

výsledků estimačních experimentů.

Samotná realizace aplikace je pak velice důkladně předvedena na dvou příkladech v kapitole páté. Tato kapitola reprezentuje i stručnou dokumentaci k aplikaci *nefLab*. Kapitola názorně ukáže, že se podařilo vytvořit aplikaci, která poskytuje uživateli jednoduchou a intuitivní možnost tvorby estimačních experimentů. K tvorbě estimačních experimentů je zapotřebí alespoň základních znalostí programového prostředí *Matlab*. Naopak k práci není potřeba porozumět detailně *toolboxu NEF*. Tedy bylo naplněno cíle zprostit uživatele od potřeby znalosti implementace jednotlivých metod nelineárního odhadování.

Aplikaci *nefLab* by se samozřejmě mohlo dostat mnoha vylepšení. Jelikož je to vlastně první takovéto rozhraní cílené pro podporu *toolboxu NEF*, které dokáže využívat jeho komponent pomocí *API NEF*, tak je tu mnoho námětů ke zlepšení. Nastiňme si závěrem práce některé z nich. Prvním vylepšením by mohlo být vytvoření *DTD* pro *XML* soubory popisující *API NEF*, který by zajisté byl kvůli obsáhlosti těchto souborů přínosem. Dalším přínosem by pak mohla být důkladnější kontrola vstupních dat, tj. zejména kontrola dimenzí matic a všeobecně všech dat, které může uživatel zadávat aplikaci. Dále by určitě bylo vhodné důsledně refaktorizovat zdrojový kód s cílem vylepšit, pročistit a zpřehlednit kód k usnadnění případného dalšího vývoje. Dále se zde nabízí možnost využívat například metod k evaluaci kvality, které nabízejí komponenty *NEF* a jsou v popisu *API NEF* zahrnuty, ale aplikace je nevyužívá. Tím ovšem aplikace neztrácí na funkčnosti, jelikož základní funkce *toolboxu NEF* jsou podporovány. Určitě by se také dala vylepšit funkce bloků pro vizualizaci výsledků estimačních experimentů na úroveň již zmíněného nástroje *nefGUI* pomocí parametrizace volitelných parametrů jednotlivých vykreslovacích metod. Dále už jen stručně, například implementace popisu jednotlivých bloků v jejich parametrizačních oknech pomocí *HTML*, implementace vhodnějšího nástroje pro zápis jednotlivých elementů matice či rozmanitější barvy spojů bloků na pracovní ploše experimentu a tak dále.

Tato práce pro mne měla veliký přínos vzhledem k velmi důslednému porozumění programovacího jazyka *Java*, ve kterém se tato práce vytvářela. Dále jsem se dopodrobna seznámil s *toolboxem NEF* a některými metodami nelineárního odhadování. Věřím, že tyto nové poznatky získané během práce na této diplomové práci budu moci zužitkovat v budoucí praxi.

Literatura

[1] prof. Ing. Miroslav Šimandl, CSc.: *Identifikace systémů a filtrace*, ZČU v Plzni, 2001

[2] prof. Ing. Miroslav Šimandl, CSc.: *Odhad stavu stochastických systémů*, ZČU v Plzni, 2010

[3] Ondřej Straka, Miroslav Flídr, Jindřich Duník a Miroslav Šimandl: *A software framework and tool for nonlinear state estimation*, ZČU v Plzni

[4] Tomáš Gotzy: *Implementace metod nelineárního odhadu a vytvoření programového prostředku pro návrh a realizaci estimačních experimentů* [diplomová práce], ZČU v Plzni, 2009

[5] Tomáš Proněk: *Nástroj pro podporu tvorby vizualizací výsledků pro toolbox NEF* [bakalářská práce], ZČU v Plzni, 2009

[6] Doc. Ing. Pavel Herout, Ph.D.: *Přednášky KIV/JXT*, ZČU v Plzni, 2008

[7] *NEF Tools for Nonlinear Estimation* © 2012 IDM Research Group [online] http://www.nft.kky.zcu.cz/nef

[8] *MATLAB Documentation* © 1984-2012 The MathWorks, Inc. [online] http://www.mathworks.com/help/techdoc/ref/function_handle.html

[9] *matlabcontrol* A Java API to interact with MATLAB [online] http://code.google.com/p/matlabcontrol/

Příloha A

CD-ROM

Na CD-ROM nalezneme dva archivy. První archiv `nef-1.2.0.zip` obsahuje *toolbox NEF* a druhý archiv `nefLab.rar` obsahuje aplikaci *nefLab* včetně elektronické verze této diplomové práce (adresář `doc`) a také vizualizační podpůrný nástroj *nefGUI* ve stejnojmenném adresáři.