České vysoké učení technické v Praze
Fakulta elektrotechnická

Katedra kybernetiky

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Student: Michal K r e č

Studijní program: Kybernetika a robotika (bakalářský)

Obor: Robotika

Název tématu: Plánování trajektorie pro bezpilotní letoun za účelem sledování pozemních objektů pomocí inerciálně stabilizované kamerové platformy

Pokyny pro vypracování:

Cílem práce je navrhnout algoritmus pro naplánování trajektorie bezpilotního letounu s ohledem na splnění úkolu vizuálního sledování statických pozemních objektů. Výstupem algoritmu nebudou přímo pokyny pro autopilot nýbrž referenční trajektorie letounu, která bude vykreslena do mapy a schválena na začátku mise lidským operátorem. Kromě samotné trajektorie letounu však budou výstupem algoritmu i požadované úhly či úhlové rychlosti pro motorizovaný systém inerciální stabilizace kamer.

Dílčími cíli práce jsou tyto:

1. Nastudujte problematiku inerciální stabilizace leteckých kamerových systému i konkrétních řešení vyvinutých a realizovaných ve skupině vedoucího této práce.
2. Formulujte rigorózně výše popsanou úlohu jako optimalizační úlohu s omezením a prozkoumejte možné přístupy k řešení (dynamické programování, prediktivní řízení, ...).
3. Pro jeden nebo dva z vyjmenovaných přístupů předveďte jeho funkčnost formou simulace a diskutujte výsledky i otevřené problémy.

Seznam odborné literatury:

Hurák, Z., and M. Řezáč: Image-based pointing and tracking for inertially stabilized airborne camera platform. IEEE Transactions on Control Systems Technology, In Press.

Vedoucí bakalářské práce: Ing. Zdeněk Hurák, Ph.D.

Platnost zadání: do konce zimního semestru 2012/2013

prof. Ing. Vladimír Mařík, DrSc.
vedoucí katedry

L.S.

prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 9. 1. 2012

Prohlášení autora práce

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze dne 28.05.2012 .....

Podpis autora práce

Abstrakt

Tato bakalářská práce zkoumá možnosti využití metod optimálního řízení pro plánování tratě bezpilotního letounu s ohledem na sledování pozemních cílů podvěšeným kamerovým systémem. Hlavní zkoumanou metodou je dynamické programovaní doplněné o myšlenky používané v metodách prohledávání stavového prostoru.

Abstract

This undergraduate thesis explores the possibility of use of methods of optimal control in path planning for UAV with repect to tracking ground targets by onboard camera system. Proposed method is dynamic programming enriched by ideas used in state-space search methods

Obsah

Použité symboly

x, y - polohové souřadnice letadla, příp. hmotného bodu

φ - úhel náklonu letadla

ψ - směrový úhel letadla, nulový směr směřuje po ose x

ψ_k - azimut kamerového systému vůči letadlu, nulový azimut směřuje ve směru letu

θ_k - elevace kamerového systému vůči letadlu, nulová elevace je v rovině letu

T - vzorkovací perioda

v - velikost vektoru rychlosti

h - výška letu nad zemí

g - tíhové zrychlení

τ - převrácená hodnota časové konstanty systému autopilota

Seznam příloh na CD

- Obrázky
  - Obrázek 1 – srov_modelu2.png
  - Obrázek 2 – srov_modelu_phi.png
  - Obrázek 3 – srov_modelu_psi.png
  - Obrázek 4 – spat_prubeh.png
  - Obrázek 5 – zamer.png – 2D, zamer.fig – 3D
  - Obrázek 6 – trajektorie_simple.png
  - Obrázek 7 – traj2.png
- Zdrojové kódy
  - testovací kód z kap. 4.1.1 – pokus.m,diskrLQTrackerpok1.m
  - java aplikace pro testovací systém – složka dyn_prog_simple (NetBeans project)
  - java aplikace pro letadlo – složka pokus_dyn_prog (NetBeans project)
- Simulink model letadla, použitý pro porovnání v kap. 2.2 – model1.mdl
- Text této práce – bp_2012_michal_krec.pdf

1 Úvod

1.1 Motivace

Výzkumná skupina Advanced Algorithms for Control and Communications působící pod vedením Ing. Zdeňka Huráka, Ph.D. na katedře řídící techniky FEL ČVUT realizuje již několik let ve spolupráci s VTÚLaPVO, Center for Machine Perception FEL ČVUT firmou ESSA vývoj inerciálně stabilizované kamerové platformy určené pro letecký průzkum z bezpilotních i pilotovaných letadel
a helikoptér. Systém je v současné době plně funkční a je zaváděn do výroby [1]. Nicméně v rámci dalšího rozvoje vyvstala myšlenka propojit úlohy sledování zadaného cíle a plánování tratě nosného letounu, které jsou v současné době řešeny samostatně. Účelem této práce je tedy prozkoumat možné způsoby, jak naplánovat trajektorii nosného letounu s ohledem na požadavky sledování cílů podvěšeným kamerovým systémem se zaměřením na metody optimálního řízení.

1.2 Cíl práce

Cílem práce je navrhnout algoritmus pro naplánování tratě bezpilotního letounu s ohledem na splnění úkolu vizuálního sledování statických pozemních objektů pomocí metod numerické optimalizace, konkrétněji navrhnout optimální regulátor, z jehož použití vyplyne optimální trať. Výstupem algoritmu nebudou ovšem přímo pokyny regulátoru pro autopilot, nýbrž referenční trať letounu, která bude vykreslena do mapy a schválena na začátku mise lidským operátorem. Kromě samotné tratě letounu však budou výstupem algoritmu i požadované úhly pro motorizovaný systém inerciální stabilizace.

2 Potřebné modely systémů

Při návrhu každého regulátoru, který má řídit nějaký systém, je většinou potřeba nejprve popsat daný systém matematickým modelem. V této práci uvažuji dva dynamické systémy. Jeden představovaný hmotným bodem, který se pohybuje pouze v jednom rozměru a působí na něj síla. A druhý představující bezpilotní letoun nesoucí inerciálně stabilizovanou platformu s kamerou. První systém nesouvisí přímo s řešenou úlohou. Uvažuji ho kvůli ověřování různých myšlenek, principů a algoritmů, které je pomocí jeho modelu možné vyzkoušet velmi snadno, bez složitých teoretických výpočtů a také jeho simulace je méně výpočetně náročná a snadněji se implementuje. Druhý model je očividně nutný k vyřešení zadané úlohy. První systém je popsán přesně, druhý s určitými zjednodušeními, viz dále. Oba systémy modeluji soustavou diferenciálních rovnic, v prvním případě lineárních a ve druhém nelineárních. Oba modely je také třeba diskretizovat, protože zvolená metoda řízení to vyžaduje. Diskretizaci jsem provedl pomocí Taylorova rozvoje způsobem uvedeným v [4], tedy pokud je systém popsán rovnicí $\dot{x}(t) = f(x, u, t)$, pak tuto rovnici můžeme nahradit následující řadou:

$$x(t+T) = x(t) + \frac{\mathsf{d}x(t)}{\mathsf{d}t}\,T + \frac{\mathsf{d}^2x(t)}{\mathsf{d}t^2}\frac{T^2}{2} + \dots$$

Protože jde o nekonečnou řadu, je samozřejmě nutné členy od určitého řádu výše zanedbat, podrobnosti popíšu u jednotlivých modelů.

2.1 Hmotný bod

Tento model popisuje systém řídící se rovnicí $m\ddot{x}(t) = F(t)$. Pro jednoduchost předpokládám jednotkovou hmotnost, takže platí $\ddot{x}(t) = F(t)$ Vstupem je síla, tedy $u(t) = F(t)$ Takže systém lze popsat následujícími rovnicemi

$$\dot{v}(t) = u(t) \tag{1a}$$

$$\dot{x}(t) = v(t). \tag{1b}$$

Rovnice popisující diskretizovaný model následující tvar

$$v(t+T) = v(t) + Tu(t) \tag{2a}$$

$$x(t+T) = x(t) + Tv(t) + \frac{T^2}{2}u(t). \tag{2b}$$

Při diskretizaci jsem přidával členy Taylorova rozvoje, dokud jsem nenarazil na potřebu vypočítat $\frac{\mathrm{d}u(t)}{\mathrm{d}t}$, což v obecném případě nelze, neboť průběh $u(t)$ není předem znám. Členy vyšších řádů se při výpočtu často numericky odhadují, ale vzhledem k tomu, že tento model je zde pouze pro svou jednoduchost, by to bylo kontraproduktivní

2.2 Letadlo s kamerovým systémem

Systém, pro který mám plánovat trajektorii, se skládá z bezpilotního letadla, na kterém je připevněn inerciálně stabilizovaný kamerový systém se dvěma navzájem kolmými osami. Pro účely této práce dynamiku kamerového systému zanedbávám, použitý regulátor je schopen sledovat manévry letadla dostatečně přesně a okamžiky zamíření na nový cíl, kde zanedbání neodpovídá realitě, nehrají pro rozhodování o trajektorii dostatečně významnou roli, která by ospravedlnila zesložitění výpočtů uvažováním dynamiky kamerového systému.

Rovnice popisující dynamiku letadla jsem převzal z [2], je v nich zahrnut autopilot, který obstarává plnění našich požadavků. Pro účely této práce počítám se třemi zjednodušeními: Za prvé, že se letadlo pohybuje konstantní rychlostí a za druhé, že se letadlo pohybuje v konstantní výšce a za třetí, že autopilot zajistí tzv. ustálenou (koordinovanou) zatáčku, tedy že příčné zrychlení na palubě letadla je ve všech okamžicích nulové. Další zjednodušení, nebo dokonce úplné zanedbání dynamiky letadla není možné, protože potřebujeme poměrně přesně znát úhel náklonu letadla, který významným způsobem ovlivňuje prostorový úhel, ve kterém je kamera schopna sledovat cíl. Přesnější model dynamiky naopak není potřeba, protože ve skutečnosti je výstupem trať, kterou bude sledovat stávající autopilot a případné odchylky skutečného náklonu jsou kompenzovány nenulovým zorným úhlem kamery. V mém řešení je totiž požadováno umístění optické osy kamery na cíl, zatímco v skutečnosti nevadí, kdy se cíl krátkodobě přesune na okraj zorného pole.

Rovnice popisující mnou uvažované letadlo jsou následující

$$\dot{x}(t) = v\cos\psi(t) \tag{3a}$$
$$\dot{y}(t) = v\sin\psi(t) \tag{3b}$$
$$\dot{\phi}(t) = \tau(u_1(t) - \phi(t)) \tag{3c}$$
$$\dot{\psi}(t) = \frac{g}{v}\tan\phi(t). \tag{3d}$$

Vstup u_1 tohoto modelu představuje požadovaný úhel náklonu předaný autopilotovi, který ho splní po určité době dané dynamikou letadla. Dále ještě uvažuji dva další vstupy, které představují azimut (u_2) a elevaci (u_3) kamery. Takže při zanedbání dynamiky kamery jsou souřadnice bodu, na který kamera míří, dány následujícími rovnicemi

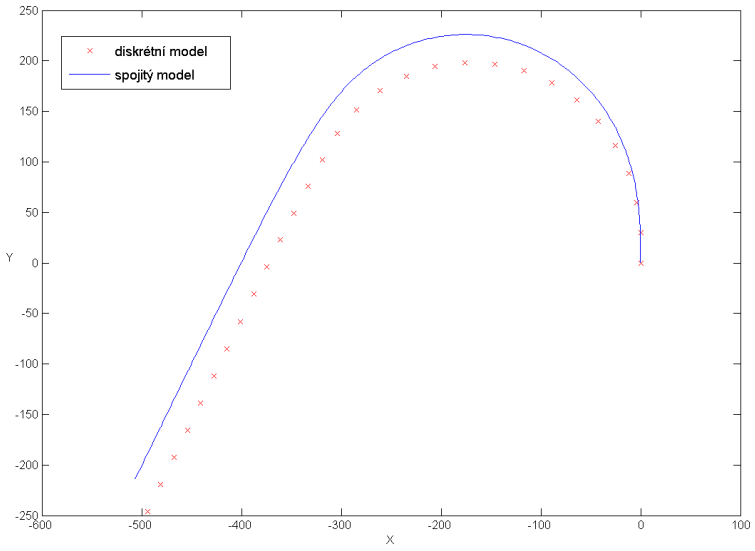
$$x_k = x + h\frac{\cos u_3 \cdot (\cos u_2 \cos\psi - \cos\phi \sin\psi \sin u_2) - \sin\phi \sin\psi \sin u_3}{\cos u_3 \sin\phi \sin u_2 - \cos\phi \sin u_3} \tag{4a}$$
$$y_k = y + h\frac{\cos u_3 \cdot (\cos u_2 \sin\psi + \cos\phi \cos\psi \sin u_2) + \sin\phi \cos\psi \sin u_3}{\cos u_3 \sin\phi \sin u_2 - \cos\phi \sin u_3}. \tag{4b}$$

Model je opět potřeba diskretizovat. Po diskretizaci vypadají rovnice modelu následovně

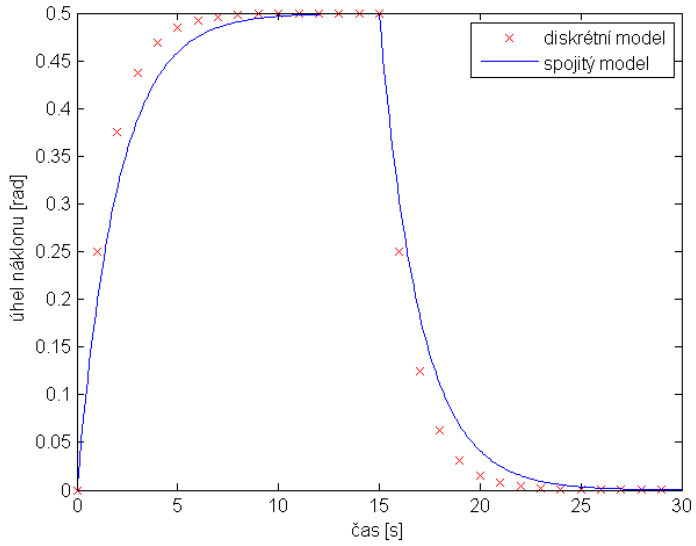
$$x(t+T) = x(t) + Tv\cos(\psi(t)) - \frac{T^2}{2}\, g\sin(\psi(t))\tan(\phi(t)) \tag{5a}$$
$$y(t+T) = y(t) + Tv\sin(\phi(t)) + \frac{T^2}{2}\, g\cos(\psi(t))\tan(\phi(t)) \tag{5b}$$
$$\phi(t+T) = \phi(t) + T\tau(u_1(t) - \phi(t)) \tag{5c}$$
$$\psi(t+T) = \psi(t) + Tg\frac{\tan(\phi(t))}{v} + \frac{T^2}{2} \cdot \frac{g\tau(u_1(t) - \phi(t))}{v\cos^2(\phi(t))}. \tag{5d}$$

U vztahů (5c) a (5d) znemožňuje přidání dalších členů Taylorova rozvoje neznalost $\frac{\mathrm{d}u(t)}{\mathrm{d}t}$, u (5a) a (5b) mi to vzhledem k použité diskretizaci nabývaných hodnot na celé metry, později násobky 5 m přišlo kontraproduktivní (zbytečné výpočty navíc).

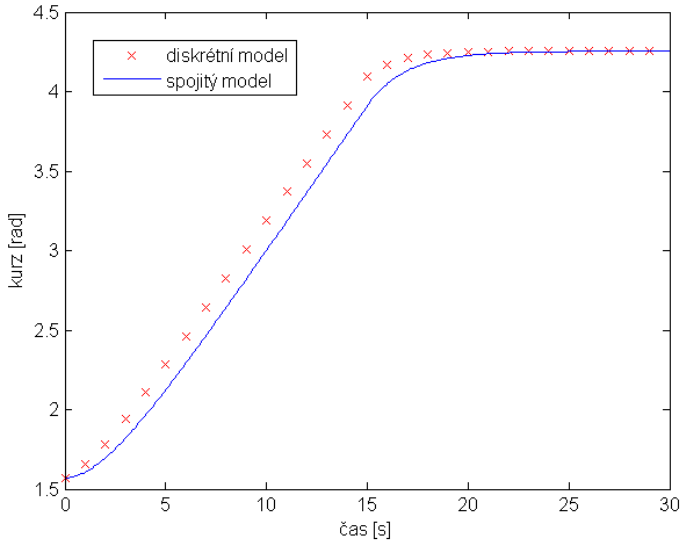
Na obrázcích 1, 2 a 3 je srovnání spojitého modelu modelovaného pomocí nástroje Simulink v MATLABu a diskrétního modelu modelovaného pomocí samotného MATLABu.



Obrázek 1: Porovnání modelů - poloha letadla



Obrázek 2: Porovnání modelů - úhel náklonu



Obrázek 3: Porovnání modelů - směrový úhel

3 Obecná úloha optimálního řízení

Všechny metody optimálního řízení mají společné použití tzv. hodnotící funkce J, která hodnotí stavovou trajektorii, pro kterou vždy hledáme minimum přes množinu možných řídících sekvencí. Hodnotící funkce má obvykle v případě diskrétního času tvar:

$$J_i = \Phi(N, x_N) + \sum_{k=i}^{N-1} L^k(x_k, u_k)$$

a v případě spojitého času tvar

$$J(t_0) = \Phi(x(T), T) + \int_{t_0}^{T} L(x(t), u(t), t)\ \mathsf{d}t,$$

kde L je cenová funkce, která je obecně časově proměnná a závisí jak na stavu tak vstupu a Φ je ohodnocení pouze posledního stavu, které by se sice dalo zahrnout do cenové funkce L, ale tradičně se uvádí zvlášť, kvůli možným výpočetním potížím při absenci vstupu v konečný čas [3].

Cílem je pak nalézt takovou sekvenci vstupů, která minimalizuje J, tedy

$$J_i^* = \min_{u_i,\ldots,u_N} \left(\Phi(N, x_N) + \sum_{k=i}^{N-1} L^k(x_k, u_k) \right),$$

při současném dodržení omezujících podmínek, které v tomto případě představuje model daného systému

$$x_{k+1} = f^k(x_k, u_k).$$

Většina metod optimálního řízení postupuje metodou Lagrangeových multiplikátorů, takže zavedou pomocnou funkci

$$J' = \Phi(N, x_N) + \sum_{k=i}^{N-1} \left[L^k(x_k, u_k) + \lambda_{k+1}^T (f^k(x_k, u_k) - x(k+1)) \right].$$

Přičemž platí, že J může nabývat minima, omezeného podmínkami f, ve stacionárních bodech J'. Dále se vyjádří gradient J' a položí se roven nule a ověří se, že u J (opravdu J, ne J') jde o minimum. Z tohoto postupu vyplývá množství parciálních diferenciálních rovnic, v případě nelineárních systémů i značně komplikovaných. To je v našem případě poměrně nepříjemné, neboť

máme nestandardní požadavky na hodnotící funkci, která bude tím pádem složitá, což se odrazí ve složitosti výsledných rovnic. Naštěstí existuje metoda, která postupuje jiným způsobem a tím se těmto nepříjemnostem vyhne - dynamické programování. Tuto metodu jsem nakonec zvolil abych eliminoval uvedené problémy.

Metody optimálního řízení se dělí podle několika kritérií. Jedním z těchto dělení je na metody s pevným konečným časem (t.j. je dáno, jak dlouho má regulace trvat) a metody s volným konečný časem. Z tohoto hlediska je zřejmé, že nás zajímají metody s volným konečným časem, neboť nemáme žádnou možnost, jak přesně zjistit potřebnou dobu letu. Dalším hlediskem je rozdělení na metody pracující ve spojitém čase a metody pracující v diskrétním čase. Z důvodů popsaných níže jsem se rozhodl věnovat jen metodám pracujícím v diskrétním čase. Také metody rozlišujeme podle toho, jestli se řeší úloha regulace nebo sledování. Při sledování je cílem, aby stavy sledovaly zadanou trajektorii, zatímco cílem regulace je přivést systém do nulového stavu. Regulace je tedy speciálním případem sledování, ale představuje natolik významnou část aplikací a navíc zjednodušení matematických vztahů, že se uvádí samostatně.

V běžných případech optimálního řízení je součástí cenové funkce L nějakým způsobem váhovaný vstup, tedy část, která souvisí s vynaloženou energií. Někdy bývá přítomná i konstantní složka, která koresponduje s požadavkem na minimální čas. A také častá složka je vzdálenost stavu od referenční tratě (nebo nulového stavu v případech striktní regulace). Co tedy požadujeme v našem případě?

Mějme zadáno následující: Počáteční a cílový stav, seznam významných bodů, které chceme sledovat, pro každý z nich minimální vzdálenost, na kterou se smíme přiblížit, maximální vzdálenost, ze které je sledování platné a minimální čas, po který je nutno je sledovat.

Jedna možnost je zahrnout do hodnotící funkce vzdálenost od aktuálního cíle, ale v určitý moment je nutné cíl změnit, když už ne na nový cíl, tak alespoň na bod, ve kterém chceme trajektorii ukončit. Což by v případě spojitého času znamenalo, že hodnotící funkce bude v čase nespojitá. To je

důvod, proč jsem se rozhodl zavrhnout metody pracující ve spojitém čase. Zde také narážíme na problém - jak poznáme, kdy můžeme změnit cíl. Dále je nutné nějak obsáhnout, jestli je kamera schopná se do požadovaného úhlu natočit. Zde vyvstává další problém - jak toto vyjádřit matematickou funkcí?

4 Aplikace optimálního řízení na zadanou úlohu

4.1 Kvadratický sledovací regulátor[1]

Jendou z možností řešení je použít regulátor, který sleduje zadanou trajektorii, podle principu popsaného v [3, kap. 4]. S tím, že reference pro směrový úhel a úhel náklonu budou mít nulovou váhu, takže regulátor je nebude brát v potaz, a do polohových stavů půjde jako reference vždy zadaný cíl. Do hodnotící funkce se buď použije jen vzdálenost od cíle, jako u klasického řízení, nebo vzdálenost od kružnice určitého průměru okolo zadaného cíle a dále konstantní složka pro minimalizaci času a zatím nespecifikovaná složka hodnotící dosažitelnost potřebných úhlů kamerového systému. Než jsem se pustil do počítání parciálních diferenciálních rovnic, které možná nemusí být řešitelné, chtěl jsem si ověřit, jak tento regulátor snáší skokové změny hodnotící funkce, které by se nejspíše objevily při počítání dosažitelnosti úhlů kamery.

4.1.1 Testování myšlenky

V MATLABu jsem napsal skript, který simuluje testovací systém (2) řízený lineárním kvadratickým sledovacím regulátorem, popsaným v [3, kap. 4, odd. 4]:

Pro model

$$x_{k+1} = Ax_k + Bu_k$$
$$y_k = Cx_k,$$

a hodnotící funkci

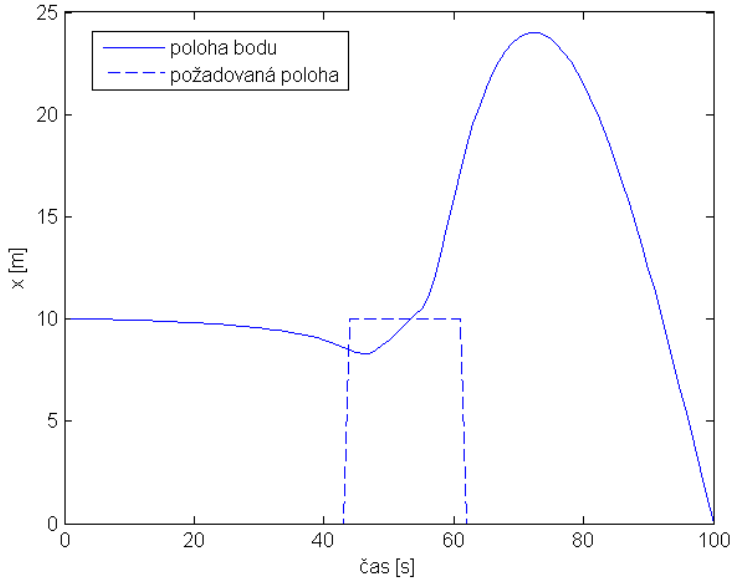
$$J_i = \frac{1}{2}(y_N - r_N)^T P(y_N - R_N) + \frac{1}{2}\sum_{k=i}^{N-1}\left[(y_k - r_k)^T Q(y_k - r_k) + u_k^T R u_k\right],$$

[1] Používaný anglický termín zní tracker, v češtině odpovídající jednoslovný název neexistuje

je optimální řídící sekvence určena následujícími vztahy

$$K_k = (B^T S_{k+1} B + R)^{-1} B^T S_{k+1} A, \qquad S_N = C^T P C$$
$$S_k = A^T S_{k+1} (A - B K_k) + C^T Q C$$
$$\nu_k = (A - B K_k)^T \nu_{k+1} + C^T Q r_k, \qquad \nu_N = C^T P r_N$$
$$K_k^\nu = (B^T S_{k+1} B + R)^{-1} B^T$$
$$u_k = -K_k x_k + K_k^\nu \nu_{k+1}.$$

Na tomto jednoduchém příkladu jsem vyzkoušel případ, kdy systém začíná ve stavu x_0, má skončit ve stavu x_N a v čase $t, 0 < t < N$ chceme aby se nacházel v bodě x_r. Regulátor jsem nastavil tak, že v určitém časovém období je do hodnotící funkce zahrnuta vzdálenost $x - x_r$ ve všech ostatních časech jen kvadrát vstupu. Skript je přiložen v příloze A. Po prozkoumání simulovaných průběhů jsem dospěl k názoru, že tyto změny v hodnotícím kritériu podstatným způsobem narušují fungování regulátoru viz obrázek 4. Skript použitý k tomuto testu je přiložen v příloze A Proto jsem se rozhodl tento směr dále nerozvíjet. A také proto, že stále není jasné, jakou podobu by vlastně měla mít hodnotící funkce, která by byla analyticky vyjádřitelná.



Obrázek 4: Simulace hmotného bodu řízeného LQ trackerem

4.2 Dynamické programování

Dynamické programování je principiálně velmi jednoduchý způsob, jak vyřešit problém řízení, který je velmi dobře schopen poradit si i s nelineárními nebo časově proměnnými systémy. Také se velmi snadno zavádějí dodatečná omezení, ať už na řídící zásah, nebo přípustné hodnoty stavů. Řídící politika je u dynamického programování vyjádřena jako stavová zpětná vazba ve formě look-up tabulky pro jednotlivé stavy v každý okamžik. Dynamické programování je založeno na *Bellmanově principu optimality*:

> Optimální strategie má tu vlastnost, že bez ohledu na to, jaká byla předchozí rozhodnutí (t.j. řídící zásahy), následující rozhodnutí musí tvořit optimální strategii vzhledem ke stavu plynoucímu z předchozích rozhodnutí [3].

Myšlenka dynamického programování je následující - mějme model popsaný rovnicí

$$x_{k+1} = f^k(x_k, u_k),$$

ohodnocený následující cenovou funkcí

$$J_i(x_i) = \Phi(N, x_N) + \sum_{k=i}^{N-1} L^k(x_k, u_k).$$

Předpokládejme, že známe optimální cenu $J^*_{k+1}(x_{k+1})$ od časového okamžiku $k+1$ do konečného okamžiku N pro všechny možné stavy x_{k+1} Pak optimální cena v čase k je dána následujícím vztahem

$$J^*_k(x_k) = \min_{u_k}(L^k(x_k, u_k) + J^*_{k+1}(x_k + 1)),$$

a optimální řídící vstup u^*_k je takové u_k pro které nastává výše zmíněné minimum [3, kap. 6].

Všimněte si, že v případě konečné množiny povolených vstupů nepotřebujeme ke zjištění minima vyjadřovat gradient, ale stačí když pro každý vstup vypočteme hodnotu J a pak vybereme tu nejmenší. Omezením vstupů na malou konečnou množinu sice ztrácíme část volnosti, ale myslím, že v tomto případě mnoho jiných řešení nezbývá.

4.2.1 Popis algoritmu

Jako řešení jsem zvolil metodu dynamického programování s mírnou úpravou (viz dále). Důvody, které mne k tomu vedly, jsou následující: Tento způsob neklade žádná omezení na vlastnosti hodnotící funkce, takže nevadí nespojitosti způsobené změnami aktuálního cíle ani případné vyjádření s logickými podmínkami. Dále použití nelineárního modelu nekomplikuje potřebné výpočty. A v neposlední řadě se velmi snadno implementují omezení nepřípustných stavů a na rozdíl od většiny ostatních přístupů platí, že čím více omezení tím lépe, což je užitečné nejen k odstranění nežádoucích letových režimů (např. náklon více než 90°), ale také k přesnému vymezení operačního prostoru, což je vzhledem k předpokládanému vojenskému využití užitečná vlastnost.

Algoritmus dynamického programování spočívá v systematickém prohledávání a ohodnocování všech přípustných stavů od konečného okamžiku až po počáteční. Pro zvolenou diskretizaci stavových proměnných například na celé metry a stupně a povolený rozsah náklonu $\pm$ 60° a operační prostor 1x1 km to vychází na $120 \cdot 360 \cdot 1000 \cdot 1000 \doteq 4,32 \times 10^{10}$ stavů na jeden časový okamžik. Dá se namítat, že takováto diskretizace je zbytečná jemná, ale stejně dobře se dá namítat, že operační prostor je příliš malý a navíc toto je pouze pro *jeden* časový okamžik. V době superpočítačů a cloud computingu jsem opatrný s výroky, že se to upočítat nedá, ale je to v daném případě za hranicí praktičnosti. Navíc se ještě větším problémem ukázalo množství paměti, potřebné pro uložení takového množství informací. Samozřejmě limitující je velikost operační paměti, protože ukládání na pevný disk by neúměrně zvýšilo časovou náročnost.

Nicméně mě napadlo, že v množině přípustných stavů je mnoho, které "nedávají smysl". Například pokud vím, že chci trajektorii končit například v bodu [0,0], tak po n časových kroků před koncem nemá smysl uvažovat body, které jsou od bodu [0,0] vzdáleny o více než $n \cdot v \cdot Ts$. Podobně se dá uvažovat i o směru a náklonu. Nakonec mě tato skutečnost přivedla na myšlenku neprohledávat celý prostor, ale "platné" stavy postupně generovat jako v mnohých aplikacích prohledávání stavového prostoru. Takže vždy v čase n vygeneruji platné stavy v čase n-1 ze znalosti stavů v čase n a přípustných hodnot vstupů a tyto ohodnotím a rozhodnu o optimálním vstupu. V obvyklé

úloze řízení by něco takového použít nešlo, protože pokud by se systém vlivem poruch dostal mimo optimální trajektorii, tak by regulátor nemusel mít daný stav uložen v tabulce. Nicméně výstupem má být v mém případě jen trať, kterou pak bude sledovat stávající autopilot, takže chybějící informace nevadí. K vygenerování předchozích stavů potřebuji znát dynamiku systému "pozadu", tu odvodím z rovnic (5) tak, že nahradím $t \to t - T$ a vyjádřím $x(t-T) = f(x(t), u(t-T))$. Tedy "pokud jsem použil tento vstup a skončil jsem v tomto stavu, z jakého stavu jsem vyšel?"

Pro testovací systém s hmotným bodem vypadá úprava následovně - z rovnic (2) získám

$$v(t-T) = v(t) - Tu(t-T) \tag{8a}$$

$$x(t-T) = x(t) - Tv(t-T) - \frac{T^2}{2}u(t-T). \tag{8b}$$

Ve druhé rovnici můžeme ponechat $v(t-T)$, protože v okamžiku výpočtu už známe jeho hodnotu z předchozího vztahu.

Pro model letadla je pak postup následující - nejprve upravím rovnici (5c) na

$$\begin{aligned}
\phi(t) &= \phi(t-T) + T\tau(u_1(t-T) - \phi(t-T)) \\
\phi(t) &= T\tau u_1(t) + \phi(t-T)(1 - T\tau) \\
\phi(t-T) &= \frac{\phi(t) - T\tau u_1(t-T)}{1 - T\tau}.
\end{aligned} \tag{9a}$$

Dále upravím rovnici (5d)

$$\begin{aligned}
\psi(t) &= \psi(t-T) + Tg\frac{\tan(\phi(t-T))}{v} + \frac{T^2 g\tau(u_1(t-T) - \phi(t-T))}{2v\cos^2(\phi(t-T))} \\
\psi(t-T) &= \psi(t) - Tg\frac{\tan(\phi(t-T))}{v} - \frac{T^2 g\tau(u_1(t-T) - \phi(t-T))}{2v\cos^2(\phi(t-T))}.
\end{aligned} \tag{9b}$$

Vzhledem k tomu, že v tomto okamžiku již znám hodnotu $\phi(t-T)$ mohu rovnici použít v tomto tvaru a nemusím ji dále upravovat. Nakonec upravím rovnice (5a) a (5b) na rovnice

$$x(t) = x(t-T) + Tv\cos(\psi(t-T)) - \frac{T^2}{2}g\sin(\psi(t-T))\tan(\phi(t-T))$$
$$x(t-T) = x(t) - Tv\cos(\psi(t-T)) + \frac{T^2}{2}G\sin(\psi(t-T))\tan(\phi(t-T)) \tag{9c}$$

$$y(t) = y(t-T) + Tv\sin(\psi(t-T)) + \frac{T^2}{2}g\cos(\psi(t-T))\tan(\phi(t-T))$$
$$y(t-T) = y(t) - Tv\sin(\psi(t-T)) - \frac{T^2}{2}g\cos(\psi(t-T))\tan(\phi(t-T)), \tag{9d}$$

které opět mohou zůstat v tomto tvaru neboť z předchozích 2 kroků znám $\phi(t-T)$ i $\psi(t-T)$.

Také je třeba v každém kroku spočítat požadované úhly azimutu a elevace kamery. Nejprve spočítám souřadnice pomocného bodu $C = [x_c, y_c, 0]$, kde první osa kamery (azimutální) protíná povrch

$$x_c = x - h\tan\phi\sin\psi \tag{10a}$$
$$y_c = y + h\tan\phi\cos\psi. \tag{10b}$$

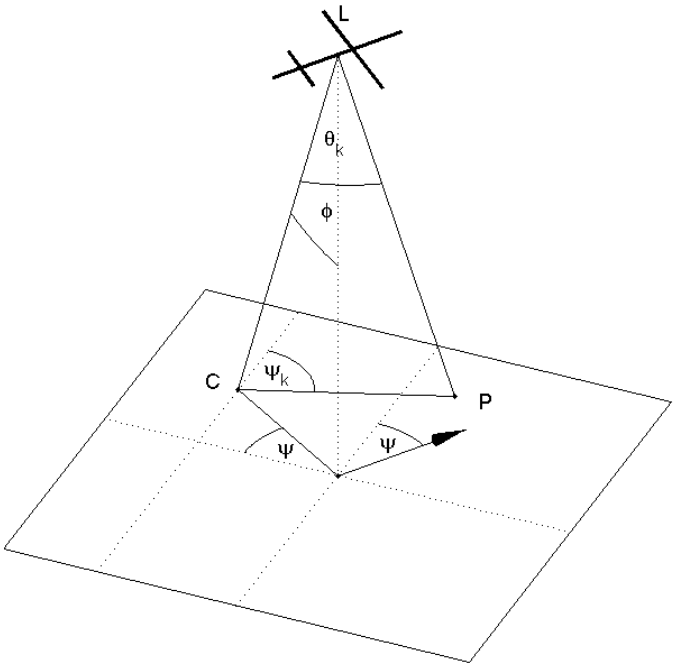
Pokud označíme souřadnice bodu, na který chceme kameru zamířit $P = [x_p, y_p, 0]$, pak azimut kamery je dán následujícím vztahem

$$\psi_k = \mathrm{atan2}(y_p - y_c, x_p - x_c) - \psi. \tag{11}$$

Dále pokud označíme $L = [x, y, h]$ polohu letadla, pak elevace kamery je dána jako úhel mezi vektory $\mathbf{u} = (C - L)$ a $\mathbf{v} = (P - L) - \frac{\pi}{2}$, tedy vztahem

$$\theta_k = \arccos\left(\frac{\mathbf{u}\cdot\mathbf{v}}{|\mathbf{u}||\mathbf{v}|}\right) - \frac{\pi}{2}. \tag{12}$$

Tyto vztahy platí v každý okamžik letu a je možné je případně využít pro samostatné navádění kamery na cíl během letu, nebo pro vypočítání požadovaných úhlů s jemnější diskretizací poté, co bude známa optimální trať.



Obrázek 5: Zaměřování kamery na cíl

Samotné prohledávání probíhá takto:
Mějme množinu všech přípustných ohodnocených stavů v čase t. Na každý z nich aplikujeme zpětnou dynamiku a postupně dosadíme všechny povolené hodnoty vstupu. Nově vygenerované vztahy vkládáme do množiny stavů v čase t-1, pokud už tato množina obsahuje daný stav, tak v ní ponecháme ten s menší cenou. Vzhledem k tomu, že u každého stavu je uložena informace o použité hodnotě vstupu je tím zároveň rozhodnuto o řídící strategii. Také je třeba zkontrolovat, jestli daný stav (resp. trať vycházející z tohoto stavu) už nemá splněné všechny cíle a zároveň není počáteční. V případě, že jsou splněny obě tyto podmínky, tak se provádění algoritmu přeruší po dokončení výpočtu pro aktuální časový okamžik. Dokončení výpočtu je nutné, abychom měli zajištěno, že máme skutečně optimální řešení. Tady se ovšem skrývá jisté nebezpečí - teoreticky je možné, že bude v daném stavu vybrán jiný vstup, vedoucí k trajektorii, která ještě nemá všechny cíle splněné. Je tedy nutné zvolit hodnotící funkci tak, aby tento případ nebyl téměř možný.

Prohledávání je zahájeno v čase 0 s množinou ohodnocených přípustných cílových stavů.

U každého stavu je kromě ceny a optimálního vstupu uložena informace o tom, jaký je stávající cíl, a kolik času je ještě požadováno ke sledování daných cílů. Vzhledem k tomu, že se pohybujeme v čase nazpět, je tuto informaci možno interpretovat jako "Kolik z požadovaných sledovacích časů nesplním, pokud budu z tohoto stavu pokračovat po optimální trajektorii?", je tedy klíčová pro rozhodování a na konci prohledávání chceme, aby součet těchto časů byl nulový.

Mnou zvolená cenová funkce $L(x_k, u_k)$ se skládá z několika komponent. V první řadě je to součet zatím nesplněných sledovacích časů násobený váhovací konstantou. Dále konstantní složka, která se přičítá jen tehdy, pokud je sledovaný cíl mimo dosah, ať už z hlediska vzdálenosti, nebo úhlů (neplatí u testovacího modelu). A nakonec vzdálenost od předcházejícího cíle, případně počátečního stavu. Druhá a třetí složka, jsou jen pomocné, bez nich by mělo příliš mnoho stavů stejnou cenu, což u zvolené implementace, která se vyznačuje jistou náhodností v pořadí vyhodnocování stavů, vedlo k tomu, že trať byla "klikatá". Tyto pomocné složky definují pomocnou strategii výběru, která stabilně mírně zvýhodňuje stavy s podobnými parametry, což vede k hladší trajektorii. U testovacího modelu, byl vyhlazující složkou cenové funkce kvadrát vstupu. Vzhledem ke způsobu ukončení výpočtu, je nalezená trať vždy nejkratší možná (z hlediska času), přičemž sice nemusí minimalizovat cenovou funkci, ale to v daném případě nevadí, protože požadavek na minimální čas je v tomto případě logický. Navíc při zvoleném tvaru cenové funkce je to většinou i její minimum.

Popsaná metoda odpovídá prohledávání stavového prostoru do šířky, kde jedna úroveň grafu představuje jeden časový okamžik. Tento způsob má tu nepříjemnou vlastnost, že množství stavů narůstá exponenciálně s časem až do okamžiku, kdy narazí na výše zmíněný plný rozsah platných stavů, nebo na limit dostupné paměti. Tento problém se dá řešit několika způsoby. Za prvé dostatečně malým počtem povolených stavů. Za druhé dostatečně nízkým faktorem větvení, spolu s dostatečně malým požadovaným počtem kroků. Za třetí použitím jiné metody, než prohledávání do šířky, která nebude potře-

bovat prohledat tolik stavů. Za čtvrté odřezáváním neperspektivních stavů. První způsob je sice velice jednoduchý, ale naprosto eliminuje myšlenku tohoto přístupu a vrací se k původnímu dynamickému programování. Druhý způsob je velmi omezující z hlediska možného použití. Třetí způsob vypadá lákavě, použít nějaký algoritmus z rodiny best-first search, tedy neprohledávat stavy postupně, ale vždy vybrat nejnadějnější. Tento způsob jsme vyzkoušel, ale narazil jsem na 2 problémy. Za prvé je poměrně obtížná volba dobré heuristiky pro určení, který stav je skutečně nejnadějnější. A za druhé, a to je ten hlavní problém, výpočetní náročnost řazení stavů, za účelem výběru nejlepšího, převážila užitek a výsledky byly nevyhovující z hlediska doby výpočtu. Navíc díky nedostatečně dobré heuristice nebyla paměťová úspora tak výrazná. Čtvrtý způsob spočívá v tom, že si při výpočtu zapamatujeme nejnižší cenu v daný moment, a při expandování stavů v dalším kroku přeskočíme ty stavy, které mají cenu M-krát větší než nejnižší. Tento způsob sice může teoreticky zahodit i stav, který by vedl k optimální trajektorii, ale při daném tvaru cenové funkce je to málo pravděpodobné. Testováním jsem zjistil, že pro mojí cenovou funkci je bezpečná hodnota M až tak nízká jako 1,5 – 2 a úspora poměrně výrazná, proto jsem toto vylepšení implementoval.

4.2.2 Implementační detaily

Řešení jsem se rozhodl implementovat jako program v jazyku Java. Hlavní důvod byl ten, že Java poskytuje podporu pro velmi komplexní práci s datovými strukturami, hlavně indexování pomocí obecného objektu, například stavového vektoru, a to pomocí tzv. kolekcí. Konkrétní kolekce kterou využívám je Map, která ukládá vždy dvojici klíč-hodnota, přičemž klíče musí být unikátní. V mém případě je klíčem stavový vektor a hodnotou datová struktura která ukládá informaci o celkové ceně z tohoto stavu do cíle, o čase, který nebyl splněn (t.j. rozdíl požadovaného času pro jednotlivé cíle, a času který bude splněn v optimální trajektorii vycházející z tohoto stavu), aktuálním cíli a optimálním vstupu v daný stav a moment. Konkrétní implementace kterou jsem použil je HashMap, která zaručuje konstantní čas pro operace put a get, tedy vložení nové hodnoty a přečtení hodnoty příslušící k zadanému klíči, pokud je pro klíče definovaná hashovací funkce korektním

způsobem [5].

Množinu stavů pro jeden časový okamžik ukládám jako hodnotu do nadřazené HashMap indexovanou časem, ale od konečného stavu dozadu - množina všech přípustných cílových stavů má tedy index 0, předcházející index $-T$, atd. Po skončení prohledávání pak tyto informace využiji k simulaci řízeného systému, podle ze které získám informaci o optimální trajektorii, kterou potom vrátím jako výstup.

Funkce zajišťující natočení kamery využívá vztahů (11) a (12) a k nastavení správných vstupů využívá výstupní parametr inputs, zatímco návratová hodnota je logická proměnná značící jestli je kamera schopna se na cíl natočit nebo ne - dá se takto přímo využít v podmínce plnění sledování.

Při vytváření nového stavu je nejprve ověřeno, že je přípustný, poté je zjištěno, jestli je schopen sledování a pokud ano je od zbývajícího sledovacího času konkrétního cíle odečten T a pokud je nový zbývající čas ≤ 0 dojde k posunu na další cíl v pořadí. Vzhledem k diskretizaci, a pevné rychlosti se nedá předpokládat, že se nějaká trať trefí přesně do počátečního stavu, je zde proto podmínka aby vzdálenost od počátečního stavu byla $\leq \frac{vT}{2}$ a odchylka směru max 60°. Pokud ke splnění těchto podmínek dojde tak se algoritmus po dokončení aktuálního časového kroku ukončí. Tímto způsobem se sice nemusí najít řešení minimalizující danou hodnotící funkci, ale je to řešení minimalizující čas při splnění zadaných podmínek, což je podstatné.

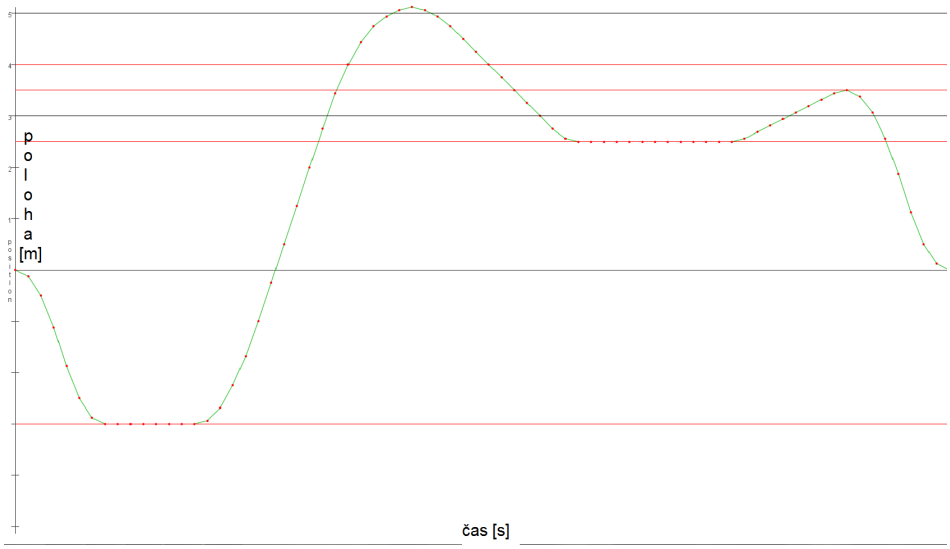
Zdrojový kód obou variant programu je přiložen v přílohách B a C.

4.2.3 Průběh testování programu

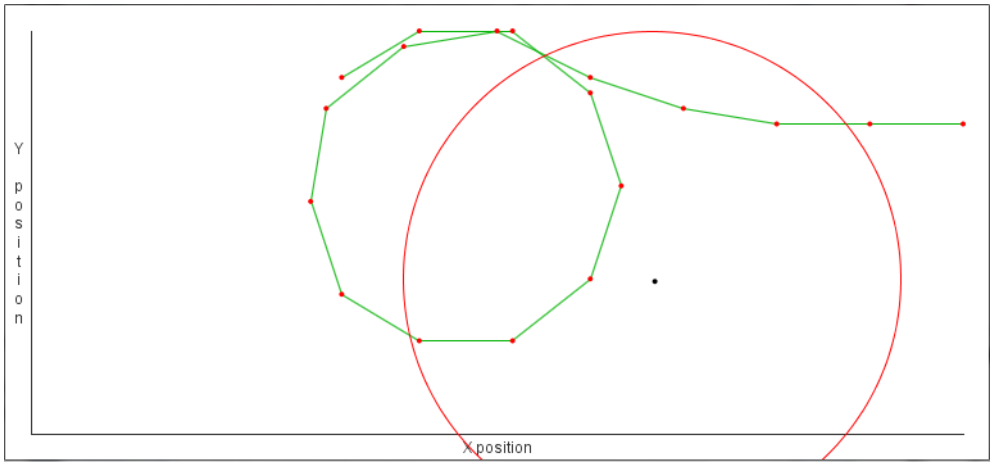
Program jsem nejdříve vyzkoušel na testovacím modelu hmotného bodu, s následujícími omezeními: Přípustné hodnoty vstupu jsou $\pm 1; \pm 0,5a0$ a přípustné hodnoty stavu jsou $|x| < 15$. Zadání je: počáteční bod - [x,v] = [0,0], koncový bod - [0,0], sledovat cíl x = -3 v rozmezí ±0 po dobu 4s, sledovat cíl x = 5 v rozmezí ±1 po dobu 6s, sledovat cíl x = 3 v rozmezí $\pm 0,5$ po dobu 15s. $T = 0,5s$. Výsledek je na obrázku 6.

Pro model letadla je použita diskretizace hodnot stavů na násobky 5 m u stavů x, y a na násobky 5° u stavů ϕ, ψ. Přípustné hodnoty vstupu u_1 jsou $\pm 60°, \pm 45°, \pm 30°$ a 0°. Hodnoty konstant jsou $\tau = 0.5, g = 10, v = 30, h =$

200 a zadání je - počáteční stav - $[x, y, \phi, \psi] = [100, 100, 0, 90]$, koncový stav - $[300, 100, 0, 0]$ a sledovat cíl na souřadnicích [200,50] z max vzdálenosti 80 po dobu 6s. Výsledek je na obrázku 7.



Obrázek 6: Výstup z programu pro testovací model, červené linky zadané hranice pro jednotlivé cíle



Obrázek 7: Výstup z programu pro model letadla, znázorněna poloha letadla při pohledu shor, černý bod představuje zadaný cíl a červená kružnice hranici uznaného sledování

5 Závěr

5.1 Dosažené výsledky

V práci jsem formuloval zadaný problém jako úlohu optimálního řízení, formuloval jsem požadavky na použitou hodnotící funkci a na základě těchto požadavků jsem porovnal několik různých metod řešení v teoretické rovině. Pro dvě z těchto metod, a to konkrétně kvadratický tracking a dynamické programování, jsem realizoval jednoduchý test vhodnosti. Na základě tohoto testu jsem kvadratický tracking jsem poté zavrhl jako neperspektivní a dále se věnoval dynamickému programování, které jsem dovedl do stadia naprogramování a odzkoušení fungujícího algoritmu, vhodného k dalšímu rozvoji.

5.2 Otevřené problémy a náměty na další výzkum

Nebyla zodpovězena otázka jak by měla vypadat hodnotící funkce, popsatelná pomocí elementárních funkcí. Takže námětem dalšího výzkumu může být návrh takovéto funkce, která by umožnila rozvoj i jiných metod než dynamického programování. Během analýzy a programování vyvstaly dva možné problémy a to exponenciální výpočetní a paměťová náročnost použitého algoritmu. Během testování vyšlo najevo, že závažnější je paměťový problém, neboť program je schopen na běžném notebooku zaplnit 2 GB přiděleného prostoru během dvou minut. Takže dalším možným směrem rozvoje je vylepšení algoritmu z tohoto hlediska. Stávající algoritmus také navštěvuje cíle v zadaném pořadí, i když by se dalo optimalizovat i přes pořadí cílů. Tuto úpravu jsem neimplementoval, protože už takto je procházeno příliš mnoho stavů. Další možností je využití stávajícího algoritmu pro předpočítání typických manévrů, které pak budou využity jednodušším plánovacím algoritmem, který je bude jen vhodně skládat za sebe.

6 Použitá literatura

[1] Hurák, Z. and Řezáč, M : Image-Based Pointing and Tracking for Inertially Stabilized Airborne Camera Platform, IEEE Transactions on Control Systems Technology, V tisku

[2] Beard, R. W. and McLain T. W. : Small Unmanned Aircraft: Theory and Practice, Princeton University Press, 2012

[3] Lewis, F. L. and Syrmos, V. L. : Optimal control, John Wiley & and Sons, Inc., 1995

[4] Hurák, Z : Introduction to numerical simulation - single step and multistep methods, lecture 12 on Modeling and Simulation of Dynamic Systems (A3B35MSD)

[5] http://docs.oracle.com/javase/tutorial/collections

Přílohy

Příloha A Testování kvadratického sledovacího regulátoru

Skript použitý pro otestování kvadratického trackeru:

```
close all
clear all
A=[1 0.5; 0 1];
B=[0.125;0.5];
C=[1 0];
x0=[10;0];
P=100; %váha finálního stavu

for k=1:100
R{k}=100;          %váha vstupu
end

for k=1:100
r{k}=0;          %inicializace trajektorie
end

for k=1:100
Q{k}=0;          %inicializace váhy trajektorie
end

pom=100*sin(0.1:0.2:pi);      %tvar váhy trajektorie

for k=1:16
Q{44+k}=pom(k);              %naplnění váhy tajektorie
end

for k=1:18
```

```
r{43+k}=10;                %naplnění tajektorie
end

[x,u,K,Kv]=diskrLQTrackerpok1(A,B,C,P,Q,R,r,100,x0);

pomx=cell2mat(x);

plot(pomx(1,:))
hold on
plot(cell2mat(r),'--')
%plot(cell2mat(Q),'g--')
hold off
figure
plot(pomx(2,:),'g')
figure
plot(cell2mat(u),'r')
```

...a funkce, kterou využívá:

```
function [x,u,K,Kv] = diskrLQTrackerpok1(A,B,C,P,Q,R,r,
    N,x0)
%       x=cell(size(x0));
%       u=cell(size(B,2),1);
%       K=cell(size(B,2),size(x0,1));
%       Kv=cell(size(B,2),size(C,2));
%       v=cell(size(C,2),1);

      x=cell(1,N);
      u=cell(1,N);
      K=cell(1,N);
      Kv=cell(1,N);
      v=cell(1,N);

      S=C'*P*C;
      v{N}=C'*P*r{N};
      for k=N-1:-1:1
          K{k}=(B'*S*B + R{k})\B'*S*A;
          Kv{k}=(B'*S*B+R{k})\B';
          S=A'*S*(A-B*K{k})+C'*Q{k}*C;
          v{k}=(A-B*K{k})'*v{k+1}+C'*Q{k}*r{k};
      end



      x{1}=x0;

      for k=1:N-1
          u{k}=-K{k}*x{k}+Kv{k}*v{k};
          x{k+1}=A*x{k}+B*u{k};
      end

end
```

Příloha B Aplikace pro letadlo

Zdrojový kód aplikace simulující systém letadla s kamerou.

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */
package pokus_dyn_prog;

import java.util.ArrayList;
import java.util.Scanner;
import java.util.HashMap;
import java.util.Set;
import javax.swing.JFrame;
//import java.util.HashSet;
//import java.io.*;

/**
 *
 * @author Michal
 */
public class Main {

    //static Scanner sc = new Scanner(System.in);
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

        System.out.println("Zadejte souřadnice
            počátečního bodu:");
        System.out.println("X: ?");
```

```
double x = sc.nextDouble();
System.out.println("Y: ?");
double y = sc.nextDouble();
System.out.println("Phi: ?");
double phi = sc.nextDouble();
System.out.println("Psi: ?");
double psi = sc.nextDouble();

State start = new State(new double[]{x, y, phi,
    psi});

System.out.println("Zadejte souřadnice cílového
   bodu:");
System.out.println("X: ?");
double tx = sc.nextDouble();
System.out.println("Y: ?");
double ty = sc.nextDouble();
System.out.println("Phi: ?");
double tphi = sc.nextDouble();
System.out.println("Psi: ?");
double tpsi = sc.nextDouble();

State goal = new State(new double[]{tx, ty,
   tphi, tpsi});
//new State(new double[]{300, 100, 0, 0})

ArrayList<POI> poi = new ArrayList<POI>();
ArrayList<Double> pomcas = new ArrayList<Double
   >();

System.out.println("Zadejte informace o cíli:")
   ;
System.out.println("X: ?");
double px = sc.nextDouble();
```

```
System.out.println("Y: ?");
double py = sc.nextDouble();
System.out.println("Maximální uznaná vzdálenost
    : ?");
double max = sc.nextDouble();
System.out.println("požadovaná doba sledování")
    ;
double cas = sc.nextDouble();

poi.add(new POI(px, py, 0, max));
pomcas.add(cas);

System.out.println("přidat další cíl? A/N");
while (sc.next().contains("A")) {
     System.out.println("Zadejte informace o
         cíli:");
     System.out.println("X: ?");
     px = sc.nextDouble();
     System.out.println("Y: ?");
     py = sc.nextDouble();
     System.out.println("Maximální uznaná
         vzdálenost: ?");
     max = sc.nextDouble();
     System.out.println("požadovaná doba
         sledování");
     cas = sc.nextDouble();
     poi.add(new POI(px, py, 0, max));
     pomcas.add(cas);
     System.out.println("přidat další cíl? A/N")
         ;
}

poi.add(null);
poi.add(null);
```

```
pomcas.add((double)0);


//State start = new State(new double[]{100,
    100, 0, Math.PI / 2});
//POI[] POIs = new POI[]{new POI(200, 50, 0,
    80), null, null};

POI[] POIs = new POI[poi.size()];
for (int i = 0; i < POIs.length; i++) {
     POIs[i]= poi.get(i);

}

double[] casy = new double[pomcas.size()];
for (int i = 0; i < casy.length; i++) {
     casy[i]=pomcas.get(i);
}
//new double[]{6, 0}
HashMap<Double, HashMap<State, StateInfo>>
    table = solve(start, goal, POIs, casy);
int stavu = 0;
double shift = 0;
int kroku = 0;
for (double k = 0; table.containsKey(k); k -=
    Model.Ts) {
     int pom = table.get(k).keySet().size();
     stavu += pom;
     System.out.println("čas: " + k + "stavů: "
         + pom);
     shift = -k;
     kroku++;
}
System.out.println("stavů celkem: " + stavu);
```

```
double[] positionsX = new double[kroku];
double[] positionsY = new double[kroku];
int krok = 0;
double cost = Double.MAX_VALUE;
State st = null;

for (State s : table.get(-shift).keySet()) {
    double pom = table.get(-shift).get(s).
        getCost();
    if (Math.abs(s.getState(3) - start.getState
        (3)) <= Math.PI / 3 && (s.distanceFrom(
        start) <= Model.v * Model.Ts / 2) && pom
         < cost) {
         st = s;
         cost = pom;
    }
}


System.out.println("Trajektorie:");
double[] inputs = null;
for (double i = 0; i < shift; i += Model.Ts) {
    inputs = table.get(i - shift).get(st).
        getOptInput();
    System.out.println("V␣čase␣" + i + "␣stav:␣
        " + st + "akční␣zásah:␣" + java.util.
        Arrays.toString(inputs));
    positionsX[krok] = st.getState(0);
    positionsY[krok++] = st.getState(1);
    //st = Model.forward(st, inputs);
    st = findClosestState(Model.forward(st,
        inputs), table.get(i - shift + Model.Ts)
        .keySet());
```

```
        }
        System.out.println("V čase " + shift + " stav:
            " + st + "akční zásah: " + java.util.Arrays.
            toString(inputs));
        positionsX[krok] = st.getState(0);
        positionsY[krok++] = st.getState(1);


        JFrame f = new JFrame();
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE
            );
        f.add(new GraphingData(positionsX, positionsY,
            POIs));
        f.setSize(400, 400);
        f.setLocation(200, 200);
        f.setVisible(true);

}

public static HashMap<Double, HashMap<State,
    StateInfo>> solve(State start, final State goal,
     POI[] POIs, final double[] POITime) {
     HashMap<Double, HashMap<State, StateInfo>>
         table = new HashMap<Double, HashMap<State,
         StateInfo>>();
     double k = 0;
     int i = 0;
     boolean check = false;
     table.put(k, new HashMap<State, StateInfo>(1) {

         {
             put(goal, new StateInfo(0, new double
                 []{0, 0, 0}, POITime, 0));
         }
```

```
});
double oldmincost = Double.MAX_VALUE;
mark:
while (true) {
    HashMap<State, StateInfo> prev_states = new
        HashMap<State, StateInfo>();

    double maxcost = Double.MIN_VALUE;
    double mincost = Double.MAX_VALUE;
    for (State state : table.get(k).keySet()) {
                                          //for
       each state at itme k,
        //we create a set of possible
           predecessors
        //check = false;
        StateInfo old = table.get(k).get(state)
           ;
        if (old.getCost() > 2 * oldmincost) {
            continue;
        }
        for (double[] input : Model.
           Permitted_inputs) {
           //by taking into account all
           possible inputs
            State prevSt = Model.backward(state
               , input);
            if (!Model.validState(k - Model.Ts,
                prevSt)) {       //Checking
               whether a newly found state is
               permitted
                continue;
            }
            int p = old.getCurrentPOIindex();
```

```
double[] pt = old.getPOITime().
    clone();
double cost = 0;
for (int j = 0; j < pt.length; j++)
    {       //spočítáme to už tady,
    ať víme, jestli už nemáme
    splněno
     cost += pt[j];
}
if (cost <= 0 && Math.abs(prevSt.
    getState(3) - start.getState(3))
     <= Math.PI / 3 && prevSt.
    distanceFrom(start) <= Model.v *
     Model.Ts / 2) {
     check = true;
}
//cost *= 50;
cost += old.getCost() + Model.cost(
    k - Model.Ts, prevSt, input,
    start, POIs[p], POIs[p + 1], pt,
     p);

if (cost < mincost) {
     mincost = cost;
}
if (cost > maxcost) {
     maxcost = cost;
}
if (pt[p] <= 0 && POIs[p] != null)
    {
     p++;
}
StateInfo si = new StateInfo(cost,
    input.clone(), pt, p);
```

```
            i++;
            if (prev_states.containsKey(prevSt)
               ) {                          //
               check whether there is same
               state already as predecessor of
               some state
                if (cost < prev_states.get(
                   prevSt).getCost()) {
                            //checking which
                   input leads to lesser cost
                    prev_states.remove(prevSt);
                    prev_states.put(prevSt, si)
                       ;
                }
            } else {
                prev_states.put(prevSt, si);
            }

        }

    }
    System.out.println("čas: " + k + "
       minimální cena: " + mincost + "
       maximální cena: " + maxcost + " stavů: "
        + prev_states.size());
    oldmincost = mincost;
    k -= Model.Ts;
    table.put(k, prev_states);
    if (check) {

        System.out.println("Všech stavů:" + i);
        break mark;
```

```
//                  double st_cost = prev_states.get(
    start).getCost();
//                  for (State s : prev_states.keySet())
    {
//                      if (st_cost < prev_states.get(s).
    getCost()) {
//                          break mark;
//                      }
//                  }

            }


        }

        return table;
    }

    public static State findClosestState(State state,
        Set<State> from) {
        int minVzdal = Integer.MAX_VALUE;
        State ret = null;
        for (State state1 : from) {
            short[] st = state.getStates();
            short[] st1 = state1.getStates();
            int pom = 0;
            for (int i = 0; i < st1.length; i++) {
                pom += Math.abs(st1[i] - st[i]);
            }
            if (pom < minVzdal) {
                minVzdal = pom;
                ret = state1;
            }
        }
```

```
            return ret;
        }
    }
```

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */
package pokus_dyn_prog;

import java.util.Arrays;

/**
 *
 * @author Michal
 */
public class State {

    private short[] states;

    public State(double[] states) {
        this.states = new short[4];
        this.states[0] = (short)(5*Math.round(states
            [0]/5));                    //TODO: dát
            pozor jaké zaokrouhlení si můžeme dovolit
        this.states[1] = (short)(5*Math.round(states
            [1]/5));
        this.states[2] = (short)(5*Math.round(states[2]
             * 360 / (5 * 2 * Math.PI)));
        this.states[3] = (short)((5*Math.round(states
            [3] * 360 / (5 * 2 * Math.PI)) + 360)%360);
    }

    public double getState(int k) {
        if (k == 2 || k == 3) {
            return states[k]*2*Math.PI/360;
        } else {
            return states[k];
```

```
    }
}

public short[] getStates() {
    return states;
}

@Override
public boolean equals(Object obj) {
    if (obj instanceof State) {
        State pom = (State) obj;
        return Arrays.equals(pom.states, this.
            states);
    }
    return false;
}

@Override
public int hashCode() {
    int hash = 5;
    hash = 83 * hash + Arrays.hashCode(this.states)
        ;
    return hash;
}

public double distanceFrom(State which) {
    double distance = 0;
    for (int i = 0; i < 2; i++) {
        distance += (states[i] - which.states[i]) *
            (states[i] - which.states[i]);
    }
    return Math.sqrt(distance);
}
```

```
    public double distnceFromPOI(POI which) {
        return Math.sqrt((states[0] - which.getX()) * (
            states[0] - which.getX()) + (states[1] -
            which.getY()) * (states[1] - which.getY()));
    }

    public boolean inRangeOfPOI(POI which) {
        return (which.getMaxD() >= this.distnceFromPOI(
            which));
    }

    @Override
    public String toString() {
        return java.util.Arrays.toString(states);
    }
}
```

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package pokus_dyn_prog;

import java.util.Arrays;

/**
 *
 * @author Michal
 */
public class StateInfo {

    private double cost;
    private double[] optInput;
    private double[] POITime;
    private int currentPOIindex;


    public StateInfo(double cost, double[] optInput,
       double[] POITime, int index) {
        this.cost = cost;
        this.optInput = optInput;
        this.POITime = POITime;
        this.currentPOIindex = index;
    }


    public double getCost() {
        return cost;
    }
```

```
public double[] getOptInput() {
    return optInput;
}

public double[] getPOITime() {
    return POITime;
}

public double getPOITime(int k) {
    return this.POITime[k];
}

public void setCost(double cost) {
    this.cost = cost;
}

public void setOptInput(double[] optInput) {
    this.optInput = optInput;
}

public int getCurrentPOIindex() {
    return currentPOIindex;
}
```

```
@Override
public String toString() {
    String rizeni = "";
    for (double d : optInput) {
        rizeni = rizeni + d + ",";
    }
    return "optimální řízení: ["+rizeni+"]";
```

```
    }



    @Override
    public int hashCode() {
        int hash = 5;
        hash = 83 * hash + Arrays.hashCode(this.
            optInput) + (int)cost + Arrays.hashCode(this
            .POITime);
        return hash;
    }



    @Override
    public boolean equals(Object obj) {
        if(obj instanceof StateInfo) {
            StateInfo pom = (StateInfo) obj;
            return (this.cost == pom.cost && Arrays.
                equals(this.optInput, pom.optInput) &&
                Arrays.equals(this.POITime, pom.POITime)
                );
        }
        return false;
    }



}
```

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package pokus_dyn_prog;

/**
 *
 * @author Michal
 */
public class POI {

    private double x,y,minD,maxD;

    public POI() {
    }



    public POI(double x, double y, double minD, double
       maxD) {
        this.x = x;
        this.y = y;
        this.minD = minD;
        this.maxD = maxD;
    }

    public double getX() {
        return x;
    }

    public double getY() {
        return y;
```

```
}

public double getMaxD() {
    return maxD;
}

public double getMinD() {
    return minD;
}



@Override
public boolean equals(Object obj) {
    if (obj == null) {
        return false;
    }
    if (getClass() != obj.getClass()) {
        return false;
    }
    final POI other = (POI) obj;
    if (this.x != other.x) {
        return false;
    }
    if (this.y != other.y) {
        return false;
    }
    return true;
}

@Override
public int hashCode() {
    int hash = 7;
```

```
        hash = 71 * hash + (int) (Double.
            doubleToLongBits(this.x) ^ (Double.
            doubleToLongBits(this.x) >>> 32));
        hash = 71 * hash + (int) (Double.
            doubleToLongBits(this.y) ^ (Double.
            doubleToLongBits(this.y) >>> 32));
        hash = 71 * hash + (int) (Double.
            doubleToLongBits(this.minD) ^ (Double.
            doubleToLongBits(this.minD) >>> 32));
        hash = 71 * hash + (int) (Double.
            doubleToLongBits(this.maxD) ^ (Double.
            doubleToLongBits(this.maxD) >>> 32));
        return hash;
    }



}
```

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */
package pokus_dyn_prog;

import java.util.LinkedList;

/**
 *
 * @author Michal
 */
public class Model {

    /* stavy jsou:
     *          [0] - x
     *          [1] - y
     *          [2] - phi
     *          [3] - psi
     */
    public static final int No_states = 4;
    public static final int No_inputs = 3;
    public static final double Ts = 1;
    public static final double v = 30;
    public static final double G = 10;
    public static final double Tau = 0.5;
    public static final int h = 200;
    public static final java.util.LinkedList<double[]>
       Permitted_inputs = new LinkedList<double[]>() {
        {
            add(new double[]{-Math.PI / 3, 0, 0});
            add(new double[]{-Math.PI / 4, 0, 0});
            add(new double[]{-Math.PI / 6, 0, 0});
            add(new double[]{0, 0, 0});
```

```
            add(new double[]{Math.PI / 6, 0, 0});
            add(new double[]{Math.PI / 4, 0, 0});
            add(new double[]{Math.PI / 3, 0, 0});
        }
    };

    public static boolean pointCamera(State state, POI
        where, double[] input) {
        double x = state.getState(0);
        double y = state.getState(1);
        double xp = where.getX();
        double yp = where.getY();
        double xc = x - h * Math.tan(state.getState(2))
            * Math.sin(state.getState(3));
        double yc = y + h * Math.tan(state.getState(2))
            * Math.cos(state.getState(3));
        double psi_k = Math.atan2(yp - yc, xp - xc);
        input[1] = psi_k;
        double u1 = xc - x;
        double u2 = yc - y;
        double h2 = h * h;
        double v1 = xp - x;
        double v2 = yp - y;
        double uv = u1 * v1 + u2 * v2 + h2;
        double utv = Math.sqrt(u1 * u1 + u2 * u2 + h2)
            * Math.sqrt(v1 * v1 + v2 * v2 + h2);
        double th_k = Math.acos(uv / utv) - Math.PI /
            2;              //počítá s povoleným
            náklopením letadla max 90°
        if (th_k > 0) {
            input[2] = 0;
            return false;
        }
        input[2] = th_k;
```

```
        return true;
    }

    public static double cost(double t, State state,
        double[] input, State start ,POI currentPoi, POI
        nextPoi, double[] POITime,int p) {
        double cost = 0;
        if (currentPoi != null && (!pointCamera(state,
            currentPoi, input) || !state.inRangeOfPOI(
            currentPoi))) {          //sets correct
            camera angles in input in the process...
            cost += 50;
        } else if(POITime[p] > 0) {
            POITime[p] -= Ts;
        }
        if (nextPoi != null) {
            cost += state.distnceFromPOI(nextPoi);
        } else {
            cost += state.distanceFrom(start);
        }
        return cost;
    }

    public static State forward(State state, double[]
        inputs) {
        State res = new State(new double[]{(state.
            getState(0) + Ts * v * Math.cos(state.
            getState(3)) - 0.5 * Ts * Ts * G * Math.sin(
            state.getState(3)) * Math.tan(state.getState
            (2))),
                    (state.getState(1) + Ts * v * Math.
                        sin(state.getState(3)) + 0.5 *
                        Ts * Ts * G * Math.cos(state.
                        getState(3)) * Math.tan(state.
```

```
                    getState(2))),
                (state.getState(2) + Ts * Tau * (
                    inputs[0] - state.getState(2))),
                (state.getState(3) + Ts * G * Math.
                    tan(state.getState(2)) / v + 0.5
                     * Ts * Ts * G * Tau * (inputs
                    [0] - state.getState(2)) / (v *
                    Math.cos(state.getState(2)) *
                    Math.cos(state.getState(2)))});
    return res;
}

public static State backward(State state, double[]
    inputs) {
    double phi = (state.getState(2) - Ts * Tau *
        inputs[0]) / (1 - Ts * Tau);
    double psi = state.getState(3) - Ts * G * Math.
        tan(phi) / v - 0.5 * Ts * Ts * G * Tau * (
        inputs[0] - phi) / (v * Math.cos(phi) * Math
        .cos(phi));
    State res = new State(new double[]{(state.
        getState(0) - Ts * v * Math.cos(psi) + 0.5 *
         Ts * Ts * G * Math.sin(psi) * Math.tan(phi)
        ),
                (state.getState(1) - Ts * v * Math.
                    sin(psi) - 0.5 * Ts * Ts * G *
                    Math.cos(psi) * Math.tan(phi)),
                phi, psi});
    return res;
}

public static boolean validState(double t, State s)
    {           //TODO: předělat na měnitelný
    operační prostor
```

```
            long x = (long) s.getState(0);
            long y = (long) s.getState(1);
            if(Math.abs(s.getState(2)) >= (Math.PI/2 -
                0.07)){           //maximální povolený náklon
                 je cca 85°
                 return false;
            }
            if(x < -200 || x > 500 || y < -200 || y > 500)
                {
                 return false;
            }
            return true;
        }
    }
```

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */
package pokus_dyn_prog;

import java.awt.*;
import java.awt.font.*;
import java.awt.geom.*;
import javax.swing.*;

public class GraphingData extends JPanel {

    final int PAD = 20;
    double[] dataX;
    double[] dataY;
    POI[] POIs;

    public GraphingData(double[] dataX, double[] dataY,
        POI[] POIs) {
        this.dataX = dataX;
        this.dataY = dataY;
        this.POIs = POIs;
    }

    @Override
    protected void paintComponent(Graphics g) {
        super.paintComponent(g);
        Graphics2D g2 = (Graphics2D) g;
        g2.setRenderingHint(RenderingHints.
            KEY_ANTIALIASING,
                RenderingHints.VALUE_ANTIALIAS_ON);

        this.setBackground(Color.white);
```

```
int w = getWidth();
int h = getHeight();
int maxX = (int) Math.ceil(getMaxX());
int maxY = (int) Math.ceil(getMaxY());
int minX = (int) Math.floor(getMinX());
int minY = (int) Math.floor(getMinY());
double xScale = (double) (w - 2 * PAD) / (maxX
    - minX);
double yScale = (double) (h - 2 * PAD) / (maxY
    - minY);
double X0 = PAD - minX * xScale;
double Y0 = PAD + maxY * yScale;
// Draw ordinate.
g2.draw(new Line2D.Double(X0, PAD, X0, h - PAD)
    );
// Draw abcissa.
g2.draw(new Line2D.Double(PAD, Y0, w - PAD, Y0)
    );
// Draw labels.
Font font = g2.getFont();
FontRenderContext frc = g2.getFontRenderContext
    ();
LineMetrics lm = font.getLineMetrics("0", frc);
float sh = lm.getAscent() + lm.getDescent();
// Ordinate label.
String s = "Y position";
float sy = PAD + ((h - 2 * PAD) - s.length() *
    sh) / 2 + lm.getAscent();
for (int i = 0; i < s.length(); i++) {
    String letter = String.valueOf(s.charAt(i))
        ;
    float sw = (float) font.getStringBounds(
        letter, frc).getWidth();
    float sx = (PAD - sw) / 2;
```

```
            g2.drawString(letter, sx, sy);
            sy += sh;
        }
        // Abcissa label.
        s = "X position";
        sy = h - PAD + (PAD - sh) / 2 + lm.getAscent();
        float sw = (float) font.getStringBounds(s, frc)
            .getWidth();
        float sx = (w - sw) / 2;
        g2.drawString(s, sx, sy);

        // Draw lines.
        g2.setPaint(Color.green.darker());
        for (int i = 0; i < dataX.length - 1; i++) {
            double x1 = X0 + xScale * dataX[i];
            double y1 = Y0 - yScale * dataY[i];
            double x2 = X0 + xScale * dataX[i + 1];
            double y2 = Y0 - yScale * dataY[i + 1];
            g2.draw(new Line2D.Double(x1, y1, x2, y2));

        }
        // Mark data points.
        g2.setPaint(Color.red);
        for (int i = 0; i < dataX.length; i++) {
            double x = X0 + xScale * dataX[i];
            double y = Y0 - yScale * dataY[i];
            g2.fill(new Ellipse2D.Double(x - 2, y - 2,
                4, 4));
        }


//          for (int i = 1; i <= max; i++) {
//              g2.draw(new Line2D.Double(2*PAD/3, h/2 +
    i*scale, 4*PAD/3, h/2 + i*scale));
```

```
//             g2.draw(new Line2D.Double(2*PAD/3, h/2 -
    i*scale, 4*PAD/3, h/2 - i*scale));
//             float nw = (float) font.getStringBounds(
    String.valueOf(i), frc).getWidth();
//             float nx = (PAD - nw) / 2;
//             float ny = (float) (h / 2 - i * scale +
    sh / 2);
//             g2.drawString(String.valueOf(i), nx, ny);
//
//         }

        for (POI P : POIs) {
            if (P != null) {
                g2.setPaint(Color.black);
                g2.fill(new Ellipse2D.Double(X0 +
                    xScale * P.getX(), Y0 - yScale * P.
                    getY(), 4, 4));
                g2.setPaint(Color.red);
                g2.draw(new Ellipse2D.Double(X0 +
                    xScale * P.getX() - xScale * P.
                    getMaxD(), Y0 - yScale * P.getY() -
                    yScale * P.getMaxD(), xScale * P.
                    getMaxD() * 2, yScale * P.getMaxD()
                    * 2));
            }
        }


    }

    private double getMaxX() {
        double max = Double.MIN_VALUE;
        for (int i = 0; i < dataX.length; i++) {
            if (dataX[i] > max) {
```

```
                max = dataX[i];
            }
        }
        if (max < 0) {
            return 0;
        }
        return max;
    }

    private double getMaxY() {
        double max = Double.MIN_VALUE;
        for (int i = 0; i < dataY.length; i++) {
            if (dataY[i] > max) {
                max = dataY[i];
            }
        }
        if (max < 0) {
            return 0;
        }
        return max;
    }

    private double getMinX() {
        double min = Double.MAX_VALUE;
        for (int i = 0; i < dataX.length; i++) {
            if (dataX[i] < min) {
                min = dataX[i];
            }
        }
        if (min > 0) {
            return 0;
        }
        return min;
    }
```

```
    private double getMinY() {
        double min = Double.MAX_VALUE;
        for (int i = 0; i < dataY.length; i++) {
            if (dataY[i] < min) {
                min = dataY[i];
            }
        }
        if (min > 0) {
            return 0;
        }
        return min;
    }
}
```

Příloha C Aplikace pro hmotný bod

Zdrojový kód aplikace simulující testovací systém

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */
package pokus_dyn_prog;

//import java.util.Scanner;
import java.util.HashMap;
import javax.swing.JFrame;
//import java.util.HashSet;
//import java.io.*;

/**
 *
 * @author Michal
 */
public class Main {

    //static Scanner sc = new Scanner(System.in);
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        State start = new State(new double[]{0, 0});
        POI[] POIs = new POI[]{new POI(3, 0, 0, 0.5),
            new POI(5, 0, 0, 1),new POI(-3, 0, 0, 0),
            null};
        HashMap<Double, HashMap<State, StateInfo>>
            table = solve(start, new State(new double
            []{0, 0}), POIs ,new double[]{15,6,4,0});
        int stavu = 0;
```

```
double shift = 0;
int kroku = 0;
for (double k = 0; table.containsKey(k); k -=
    Model1.Ts) {
      int pom = table.get(k).keySet().size();
      stavu += pom;
      System.out.println("čas: " + k + "stavů: "
          + pom);
      shift = -k;
      kroku++;
}
System.out.println("stavů celkem: " + stavu);
double[] positions = new double[kroku];
int krok = 0;
State s = start;
System.out.println("Trajektorie:");
double[] inputs;
for (double i = 0; i <= shift; i += Model1.Ts)
    {
      inputs = table.get(i - shift).get(s).
          getOptInput();
      System.out.println("V čase " + i + " stav:
          " + s + "akční zásah: " + java.util.
          Arrays.toString(inputs));
      positions[krok++]=s.getState(0);
      s = Model1.forward(s, inputs);
}

JFrame f = new JFrame();
f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE
    );
f.add(new GraphingData(positions,POIs));
f.setSize(400,400);
f.setLocation(200,200);
```

```
        f.setVisible(true);

    }

    public static HashMap<Double, HashMap<State,
        StateInfo>> solve(State start, final State goal,
        POI[] POIs, final double[] POITime) {
        HashMap<Double, HashMap<State, StateInfo>>
            table = new HashMap<Double, HashMap<State,
            StateInfo>>();
        double k = 0;
        int i = 0;
        table.put(k, new HashMap<State, StateInfo>(1) {
            {
                put(goal, new StateInfo(0, new double
                    []{0,0},POITime,0));
            }
        });
        mark:
        while (true) {
            HashMap<State, StateInfo> prev_states = new
                HashMap<State, StateInfo>();
            for (State state : table.get(k).keySet()) {
                                                //for
                each state at itme k,
                //we create a set of possible
                    predecessors
                StateInfo old = table.get(k).get(state)
                    ;
                for (double[] input : Model1.
                    Permitted_inputs) {
                    //by taking into account all
                    possible inputs
```

```
State prevSt = Model1.backward(
    state, input);
int p = old.getCurrentPOIindex();
double[] pt = old.getPOITime().
    clone();
double cost = old.getCost() +
    Model1.cost(k - Model1.Ts,
    prevSt, input, POIs[p], old.
    getPOITime(),goal);
if (Model1.validState(k - Model1.Ts
    , prevSt)) {       //Checking
    whether a newly found state is
    permitted
     if(pt[p] > 0) {
          if(prevSt.inRangeOfPOI(POIs
              [p])) {
               pt[p] -= Model1.Ts;
          }
     } else if(POIs[p] != null){
          p++;
     }
     StateInfo si = new StateInfo(
        cost, input, pt,p);
     i++;
     if (prev_states.containsKey(
        prevSt)) {
                                //
        check whether there is same
        state already as predecessor
         of some state
         if (cost < prev_states.get(
            prevSt).getCost()) {
                     //checking
            which input leads to
```

```
                                lesser cost
                                 prev_states.remove(
                                    prevSt);
                                 prev_states.put(prevSt,
                                     si);
                            }
                        } else {
                            prev_states.put(prevSt, si)
                                ;
                        }
                    }
                }

            }
            k -= Model1.Ts;
            table.put(k, prev_states);
            if (prev_states.containsKey(start)) {
                double sum = 0;
                for (double d : prev_states.get(start).
                    getPOITime()) {
                    sum += d;
                }
                if(sum <= 0){
                    System.out.println("Všech stavů:" +
                        i);
                    break mark;
                }
//                  double st_cost = prev_states.get(
    start).getCost();
//                  for (State s : prev_states.keySet())
    {
//                      if (st_cost < prev_states.get(s).
    getCost()) {
//                          break mark;
```

```
//                          }
//                      }

                }


            }

            return table;
        }
}
```

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package pokus_dyn_prog;

import java.util.Arrays;

/**
 *
 * @author Michal
 */
public class State {
    private double[] states;

    public State(double[] states) {
        this.states = states;
    }

    public double getState(int k) {
        return states[k];
    }

    public double[] getStates() {
        return states;
    }

    @Override
    public boolean equals(Object obj) {
        if(obj instanceof State) {
            State pom = (State) obj;
            return Arrays.equals(pom.states, this.
                states);
```

```
    }
    return false;
}

@Override
public int hashCode() {
    int hash = 5;
    hash = 83 * hash + Arrays.hashCode(this.states)
        ;
    return hash;
}

public double distanceFrom(State which) {
    double distance = 0;
    for (int i = 0; i < states.length; i++) {
        distance+=Math.abs(states[i]-which.states[i
            ]);
    }
    return distance;
}

public double distnceFromPOI(POI which) {

   //TODO: proper distance
    return Math.sqrt((states[0]-which.getX())*(
        states[0]-which.getX()));
}

public boolean inRangeOfPOI(POI which) {
    return (which.getMaxD() >= this.distnceFromPOI(
        which));
}

@Override
```

```
    public String toString() {
//          String stavy = "[";
//          for (double d : states) {
//              stavy = stavy + d +"," ;
//          }
//          return stavy + "]; ";
        return java.util.Arrays.toString(states);
    }


}
```

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package pokus_dyn_prog;

import java.util.Arrays;

/**
 *
 * @author Michal
 */
public class StateInfo {

    private double cost;
    private double[] optInput;
    private double[] POITime;
    private int currentPOIindex;


    public StateInfo(double cost, double[] optInput,
       double[] POITime, int index) {
        this.cost = cost;
        this.optInput = optInput;
        this.POITime = POITime;
        this.currentPOIindex = index;
    }


    public double getCost() {
        return cost;
    }
```

```
public double[] getOptInput() {
    return optInput;
}

public double[] getPOITime() {
    return POITime;
}

public double getPOITime(int k) {
    return this.POITime[k];
}

public void setCost(double cost) {
    this.cost = cost;
}

public void setOptInput(double[] optInput) {
    this.optInput = optInput;
}

public int getCurrentPOIindex() {
    return currentPOIindex;
}
```

```
@Override
public String toString() {
    String rizeni = "";
    for (double d : optInput) {
        rizeni = rizeni + d + ",";
    }
    return "optimální řízení: ["+rizeni+"]";
```

```
    }



    @Override
    public int hashCode() {
        int hash = 5;
        hash = 83 * hash + Arrays.hashCode(this.
            optInput) + (int)cost + Arrays.hashCode(this
            .POITime);
        return hash;
    }



    @Override
    public boolean equals(Object obj) {
        if(obj instanceof StateInfo) {
            StateInfo pom = (StateInfo) obj;
            return (this.cost == pom.cost && Arrays.
                equals(this.optInput, pom.optInput) &&
                Arrays.equals(this.POITime, pom.POITime)
                );
        }
        return false;
    }



}
```

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package pokus_dyn_prog;

/**
 *
 * @author Michal
 */
public class POI {

    private double x,y,minD,maxD;

    public POI(double x, double y, double minD, double
       maxD) {
        this.x = x;
        this.y = y;
        this.minD = minD;
        this.maxD = maxD;
    }

    public double getX() {
        return x;
    }

    public double getY() {
        return y;
    }

    public double getMaxD() {
        return maxD;
    }
```

```
public double getMinD() {
    return minD;
}



@Override
public boolean equals(Object obj) {
    if (obj == null) {
        return false;
    }
    if (getClass() != obj.getClass()) {
        return false;
    }
    final POI other = (POI) obj;
    if (this.x != other.x) {
        return false;
    }
    if (this.y != other.y) {
        return false;
    }
    return true;
}

@Override
public int hashCode() {
    int hash = 7;
    hash = 71 * hash + (int) (Double.
       doubleToLongBits(this.x) ^ (Double.
       doubleToLongBits(this.x) >>> 32));
    hash = 71 * hash + (int) (Double.
       doubleToLongBits(this.y) ^ (Double.
       doubleToLongBits(this.y) >>> 32));
```

```
        hash = 71 * hash + (int) (Double.
            doubleToLongBits(this.minD) ^ (Double.
            doubleToLongBits(this.minD) >>> 32));
        hash = 71 * hash + (int) (Double.
            doubleToLongBits(this.maxD) ^ (Double.
            doubleToLongBits(this.maxD) >>> 32));
        return hash;
    }



}
```

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package pokus_dyn_prog;

import java.util.LinkedList;

/**
 *
 * @author Michal
 */
public class Model1 {

    public static final int No_states = 2;
    public static final int No_inputs = 1;
    public static final double Ts = 0.5;
    public static final java.util.LinkedList<double[]>
       Permitted_inputs = new LinkedList<double[]>(){
        {add(new double[] {-1});
         add(new double[] {-0.5});
         add(new double[] {0});
         add(new double[] {0.5});
         add(new double[] {1});
        }
    };


    public static double cost(double t, State state,
       double[] input, POI currentPOI, double[] POItime
       , State goal) {
        double sum =10;
        for (double d : POItime) {
```

```
            sum += d;
        }
        double cost = input[0]*input[0] + sum;
//          if(currentPOI != null && state.distnceFromPOI
    (currentPOI) > currentPOI.getMaxD()) {
//              cost += state.distnceFromPOI(currentPOI);
//          }
        return cost;
    }

    public static State forward(State states, double[]
       inputs) {
        State res = new State(new double[] {(states.
           getStates()[0]+Ts*states.getStates()[1]+0.5*
           Ts*Ts*inputs[0]),(states.getStates()[1]+Ts*
           inputs[0])});
        return res;
    }

    public static State backward(State states, double[]
        inputs) {
        State res = new State( new double[] {(states.
           getStates()[0]-Ts*states.getStates()[1]+0.5*
           Ts*Ts*inputs[0]),(states.getStates()[1]-Ts*
           inputs[0])});
        return res;
    }

    public static boolean validState(double t, State s)
        {          //in different models this can be of
        course more complicated
        if(Math.abs(s.getState(0)) > 25) {
            return false;
        }
```

```
            return true;
        }
    }
```

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */
package pokus_dyn_prog;

import java.awt.*;
import java.awt.font.*;
import java.awt.geom.*;
import javax.swing.*;

public class GraphingData extends JPanel {

    final int PAD = 20;
    double[] dataX;
    double[] dataY;
    POI[] POIs;

    public GraphingData(double[] dataX, double[] dataY,
        POI[] POIs) {
        this.dataX = dataX;
        this.dataY = dataY;
        this.POIs = POIs;
    }

    @Override
    protected void paintComponent(Graphics g) {
        super.paintComponent(g);
        Graphics2D g2 = (Graphics2D) g;
        g2.setRenderingHint(RenderingHints.
            KEY_ANTIALIASING,
                RenderingHints.VALUE_ANTIALIAS_ON);

        this.setBackground(Color.white);
```

```
int w = getWidth();
int h = getHeight();
int maxX = (int) Math.ceil(getMaxX());
int maxY = (int) Math.ceil(getMaxY());
int minX = (int) Math.floor(getMinX());
int minY = (int) Math.floor(getMinY());
double xScale = (double) (w - 2 * PAD) / (maxX
    - minX);
double yScale = (double) (h - 2 * PAD) / (maxY
    - minY);
double X0 = PAD - minX * xScale;
double Y0 = PAD + maxY * yScale;
// Draw ordinate.
g2.draw(new Line2D.Double(X0, PAD, X0, h - PAD)
    );
// Draw abcissa.
g2.draw(new Line2D.Double(PAD, Y0, w - PAD, Y0)
    );
// Draw labels.
Font font = g2.getFont();
FontRenderContext frc = g2.getFontRenderContext
    ();
LineMetrics lm = font.getLineMetrics("0", frc);
float sh = lm.getAscent() + lm.getDescent();
// Ordinate label.
String s = "Y position";
float sy = PAD + ((h - 2 * PAD) - s.length() *
    sh) / 2 + lm.getAscent();
for (int i = 0; i < s.length(); i++) {
    String letter = String.valueOf(s.charAt(i))
        ;
    float sw = (float) font.getStringBounds(
        letter, frc).getWidth();
    float sx = (PAD - sw) / 2;
```

```
            g2.drawString(letter, sx, sy);
            sy += sh;
        }
        // Abcissa label.
        s = "X position";
        sy = h - PAD + (PAD - sh) / 2 + lm.getAscent();
        float sw = (float) font.getStringBounds(s, frc)
            .getWidth();
        float sx = (w - sw) / 2;
        g2.drawString(s, sx, sy);

        // Draw lines.
        g2.setPaint(Color.green.darker());
        for (int i = 0; i < dataX.length - 1; i++) {
            double x1 = X0 + xScale * dataX[i];
            double y1 = Y0 - yScale * dataY[i];
            double x2 = X0 + xScale * dataX[i + 1];
            double y2 = Y0 - yScale * dataY[i + 1];
            g2.draw(new Line2D.Double(x1, y1, x2, y2));

        }
        // Mark data points.
        g2.setPaint(Color.red);
        for (int i = 0; i < dataX.length; i++) {
            double x = X0 + xScale * dataX[i];
            double y = Y0 - yScale * dataY[i];
            g2.fill(new Ellipse2D.Double(x - 2, y - 2,
                4, 4));
        }


//          for (int i = 1; i <= max; i++) {
//              g2.draw(new Line2D.Double(2*PAD/3, h/2 +
    i*scale, 4*PAD/3, h/2 + i*scale));
```

```
//              g2.draw(new Line2D.Double(2*PAD/3, h/2 -
    i*scale, 4*PAD/3, h/2 - i*scale));
//              float nw = (float) font.getStringBounds(
    String.valueOf(i), frc).getWidth();
//              float nx = (PAD - nw) / 2;
//              float ny = (float) (h / 2 - i * scale +
    sh / 2);
//              g2.drawString(String.valueOf(i), nx, ny);
//
//          }

            for (POI P : POIs) {
                if (P != null) {
                    g2.setPaint(Color.black);
                    g2.fill(new Ellipse2D.Double(X0 +
                        xScale * P.getX(), Y0 - yScale * P.
                        getY(), 4, 4));
                    g2.setPaint(Color.red);
                    g2.draw(new Ellipse2D.Double(X0 +
                        xScale * P.getX() - xScale * P.
                        getMaxD(), Y0 - yScale * P.getY() -
                        yScale * P.getMaxD(), xScale * P.
                        getMaxD() * 2, yScale * P.getMaxD()
                        * 2));
                }
            }


        }

        private double getMaxX() {
            double max = Double.MIN_VALUE;
            for (int i = 0; i < dataX.length; i++) {
                if (dataX[i] > max) {
```

```
                max = dataX[i];
            }
        }
        if (max < 0) {
            return 0;
        }
        return max;
    }

    private double getMaxY() {
        double max = Double.MIN_VALUE;
        for (int i = 0; i < dataY.length; i++) {
            if (dataY[i] > max) {
                max = dataY[i];
            }
        }
        if (max < 0) {
            return 0;
        }
        return max;
    }

    private double getMinX() {
        double min = Double.MAX_VALUE;
        for (int i = 0; i < dataX.length; i++) {
            if (dataX[i] < min) {
                min = dataX[i];
            }
        }
        if (min > 0) {
            return 0;
        }
        return min;
    }
```

```
    private double getMinY() {
        double min = Double.MAX_VALUE;
        for (int i = 0; i < dataY.length; i++) {
            if (dataY[i] < min) {
                min = dataY[i];
            }
        }
        if (min > 0) {
            return 0;
        }
        return min;
    }
}
```