Jihočeská univerzita v Českých Budějovicích

Přírodovědecká fakulta

Konstrukce alternativního ovládání robotické ruky Arexx RA-1 Pro pomocí jednodeskového mikrokontroleru Arduino.

Bakalářská práce

Štěpán Mudra

Školitel: Mgr. Jiří Pech Ph.D.

České Budějovice 2019

Bibliografické údaje

Mudra, Š., 2018: Konstrukce alternativního ovládání robotické ruky Arexx RA-1 Pro pomocí jednodeskového mikrokontroleru Arduino. [The construction of an alternative controller the Arexx RA-1 Pro robotic arm with single – board microcontroller Arduino. Bc. Thesis, in Czech.] – 36 p., Faculty of Science, University of South Bohemia, České Budějovice, Czech Republic.

Anotace

Bakalářská práce se zabývá vytvořením nového řídicího systému pro robotickou ruku Arexx RA-1 Pro. Systém zahrnuje novou řídicí desku na bázi Arduina a software, naprogramovaný v Javě, umožňující uživateli ovládání robotické ruky. Dále je zde popsáno ovládání této ruky včetně teorie.

Klíčová slova

Robot, kinematika, matice, Arduino, java

Annotation

This bachelor‘s thesis focuses on the creation of a new control system for the Arexx RA-1 Pro robotic arm. The system includes a new control board based on Arduino and the software programmed in Java enabling user to control the arm. The thesis also contains a description of the controlling system including the theoretical knowledge.

Keywords

Robot, kinematisc, matrix, Arduino, java

Poděkování

Rád bych poděkoval Mgr. Jiřímu Pechovi, PhD. za vedení bakalářské práce, konzultace problematiky a odborný dohled. Dále bych rád poděkoval Mgr. Lence Zalabové, PhD. za objasnění vzniklých matematických problémů.

Prohlášení

Prohlašuji, že svoji bakalářskou práci jsem vypracoval samostatně pouze s použitím pramenů a literatury uvedených v seznamu citované literatury.

Prohlašuji, že v souladu s § 47b zákona č. 111/1998 Sb. v platném znění souhlasím se zveřejněním své bakalářské práce, a to v nezkrácené podobě elektronickou cestou ve veřejně přístupné části databáze STAG provozované Jihočeskou univerzitou v Českých Budějovicích na jejích internetových stránkách, a to se zachováním mého autorského práva k odevzdanému textu této kvalifikační práce. Souhlasím dále s tím, aby toutéž elektronickou cestou byly v souladu s uvedeným ustanovením zákona č. 111/1998 Sb. zveřejněny posudky školitele a oponentů práce i záznam o průběhu a výsledku obhajoby kvalifikační práce. Rovněž souhlasím s porovnáním textu mé kvalifikační práce s databází kvalifikačních prací Theses.cz provozovanou Národním registrem vysokoškolských kvalifikačních prací a systémem na odhalování plagiátů.

V Českých Budějovicích dne …… Štěpán Mudra

Obsah

1 Seznam pojmů a zkratek

Níže uvádím několik základních pojmů, některé z nich budou podrobněji rozebírány dále v práci.

- Robot – automatický manipulátor.
- Základna – místo, na kterém robot stojí.
- Kloub – pohyblivé místo, spojující dvě ramena.
- Rameno – nepohyblivá část robota (hýbe se jen díky kloubům).
- End efektor (někdy též chapadlo) – část robota, která umožňuje uchopit předměty.
- Stupeň volnosti – kloub robota.
- Kinematická dvojice – dvojice propojeného kloubu a ramene.
- Kinematický řetězec – skupina na sebe přímo navazujících kinematických dvojic.

2 Úvod

Produkt *Arexx RA1-PRO* obsahuje zdroj napájení, robotickou ruku, šest servomotorů, řídicí desku, překladač, tlačítkový ovladač, návod k použití a výchozí ovládací software, který je dostupný ke stažení z oficiálních stránek *Arexx:* (http://www.arexx.com/robot_arm/html/en/software.htm).

Hlavním důvodem pro vytváření tohoto řešení je nemožnost použít původní řešení. Programy původního řešení používají dřívější verze programovacích jazyků a zastaralé knihovny. Například program, který přeposílá instrukce v hex souborech, využívá pro sériovou komunikaci RX/TX knihovnu, která již nemusí být plně kompatibilní s novějšími verzemi Javy i sériových linek, protože není aktualizována.

Původní řešení obsahovalo více programů, přičemž každý sloužil pro jiný účel. Jeden pro „realTime“ ovládání, další pro nahrávání instrukcí v hex souborech. Dále původní ovládací deska buďto chybně interpretovala obdržená data, nebo nekomunikovala se servomotory. Odesílání instrukcí polohy servomotorů nezměnilo.

3 Cíle

Cílem této bakalářské práce je vytvořit zcela nové a funkční ovládání robotické ruky Arexx Ra-1 Pro.

Výsledný program bude obsahovat grafické rozhraní, které umožní ovládat robot v reálném čase, například posuvníky.

Aplikace dále dovolí nahrání jednotlivých pohybů a jejich následné uložení do souboru. Tento uložený soubor bude možné prostřednictvím aplikace otevřít, sérii pohybů prohlédnout a nechat robot provést tyto pohyby.

Program také zahrne i funkcionalitu pro výpočet přímé kinematické úlohy robotu. Pro zadání nastavení robotu bude vytvořeno grafické rozhraní, které umožní zadat jednotlivé parametry, tlačítko pro spuštění výpočtu a místo pro zobrazení výsledku.

4 Teoretická část

Teoretické podklady a jejich využití pro praktickou část práce.

4.1 Robotika

Součást mechatroniky, jejíž předmět zájmu jsou humanoidní roboti a průmyslové roboty. Pokrývá veškeré činnosti spojené s robotem (návrh robotu, vytvoření programu, simulace provozu, vytvoření funkčního celku). [1]

4.1.1 Robot

Pod slovem robot si většina lidí představí stroj, který se podobá člověku (C3PO, pan Dat, nebo třeba Terminátor). Jako první ovšem použil slovo robot český spisovatel Karel Čapek pro svoji známou vědeckofantastickou hru R.U.R. v roce 1920 na popud svého bratra Josefa. V tomto kontextu mluvíme o tzv. humanoidních robotech. [1], [4]

Dalším možným kontextem pro slovo robot je robot průmyslový. Mluvíme o manipulátorech, na které se zaměřuje tato práce. Proto nyní definujeme slovo robot takto.: **„Stroj, který je schopen vykonávat činnost nezávislou na lidské obsluze.“**

Roboty můžeme dělit dle kinematické struktury nebo euklidovského prostoru:

Podle druhu kinematické struktury:

- Sekvenční – sekvenčním robotem lze nazvat takového robota, jehož end efektor přímo navazuje na jeden kinematický řetězec začínající základnou. Tento řetězec je pak následně složen z několika kinematických dvojic. Sekvenční struktura dovoluje robotu mít takřka maximální pracovní prostor dle jeho možností. Nastává však problém, že každý kloub robota je nepřesný v pohybu s ramenem, takže výsledné souřadnice end efektoru jsou změněny sumarizací chyb všech kinematických dvojic oproti požadavkům. Chybovost roste s počtem stupňů volnosti robota a snižuje opakovatelnost pohybů bez kalibrace. Na obrázku můžeme vidět robota o dvou stupních volnosti pohybujícího se v prostoru. [1]

- Paralelní – end efektor paralelního robotu je přímo spojen s vícero kinematickými řetězci. Toto spojení eliminuje chybovost v umístění chapadla, avšak se zmenšeným pracovním prostorem robotu. [1]

Podle prostoru, ve kterém se robot pohybuje:

- L^2 – roboty se mohou pohybovat jen v rovině E^2, jde zejména například o plottery, které se pohybují posunutím po osách x a y.
- L^3 – roboty pohybující se v třídimenzionálním prostoru, vídáme je o něco častěji. Používají se například v továrnách k přemisťování. Tyto roboty jsou schopny pohybu v tříosovém souřadném systému E^3. Nemusí však jít pouze o pohyby posunutí po jednotlivých osách, ale i o rotaci kolem os.

4.1.2 Vlastnosti popisující robota

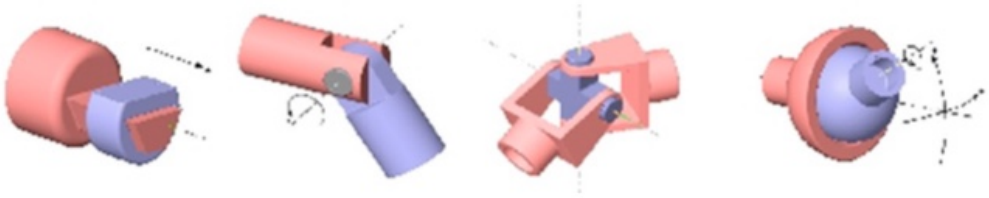
Kromě kinematické struktury a prostoru existují další faktory, kterými lze robota popsat:

Stupeň volnosti

Anglicky degree of freedom (DoF), označuje počet kloubů robotu. Každé těleso, dokonce i bod, má své stupně volnosti:

- 2D – pro dvoudimenzionální prostor má bod 2 stupně volnosti (posun po ose x, y). Těleso k těmto dvěma navíc může rotovat okolo bodu, díky tomu má o jeden DoF víc než bod.
- 3D – pro třídimenzionální prostor má bod 3 stupně volnosti (posun po ose x, y, z). V závislosti na rotaci je volnost rozšířena na 6 stupňů (posun po všech osách a zároveň rotaci okolo všech os).

Stupeň volnosti je realizován klouby (jointy) robotu. Základní druhy kloubů jsou: [1]



Jednotlivé klouby zleva:

- Posuvný kloub – je jím realizován stupeň volnosti pro posunutí. [1]
- Rotační kloub – realizuje rotační stupeň volnosti. [1]
- Univerzální – umožňuje dva rotační stupně volnosti. [1]
- Sférický – přidává všechny tři rotační DoF. [1]

Pracovní prostor

Pracovní prostor je další z vlastností robotu. Pracovní prostor robotu znamená takový prostor, ve kterém se robot pohybuje bez nárazu na překážku (např. stůl, na kterém robot stojí). Velikost a omezenost pracovního prostoru závisí na počtu stupňů volnosti robotu a délce jeho ramen.

Pro dvourozměrný prostor robot potřebuje alespoň 3 stupně volnosti, aby pro něj mohl existovat pracovní prostor, ve kterém by byl schopen dosáhnout jakéhokoli bodu s jakoukoli orientací end efektoru. [9]

Pro třídimenzionální prostor by pak robot musel disponovat minimálně šesti stupni volnosti, aby mohl existovat pracovní prostor, ve kterém robot dosáhne jakéhokoli bodu s jakoukoli orientací. [9]

4.2 Popis robotu

Existuje několik ustanovených formátů pro popis robotu. Nejvíce rozšířené možnosti pro popis robotu jsou tyto:

- Denavit–Hartenbergova notace.
- Hayati–Robertsova notace.
- URDF (Unified Robot Description Format).

V práci dále budou využívány právě D–H parametry, především pro praktickou část. Rozšířenost této metody přepisu robotu je dána zejména její jednoduchostí a čitelností oproti alternativám a dále tím, že je historicky nejstarší a snadno implementovatelná.

Existuje několik pravidel pro přepis robota do D–H notace: [6]

1. Osa kloubu je osa z pro kloub.
2. Rameni$_i$ je dovoleno, buď měnit svou délku po ose x_i, nebo okolo osy x rotovat.

4.3 Kinematika

Pro roboty řešíme přímou a zpětnou kinematiku.

Přímá kinematika znamená pro dané parametry najít pozici ee.

Pro výpočet přímé kinematiky je nutno přepsat robota z D–H notace do maticových souřadnic. D–H notace udává úhel, o který se kloub otočil kolem dané osy. Rotaci okolo osy lze zaznamenat v matici rotace pro danou osu. Matice rotace podle os x, y, z: [1]

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & \sin\alpha \\ 1 & \sin\alpha & \cos\alpha \end{pmatrix}$$

Matice podle x. 1

$$\begin{pmatrix} \cos\alpha & 0 & -\sin\alpha \\ 0 & 1 & 0 \\ \sin\alpha & 0 & \cos\alpha \end{pmatrix}$$

Matice podle y. 1

$$\begin{pmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Matice podle z. 1

Takovéto matice budou rozšířeny o vektor (4. sloupec), který udává souřadnice pro x, y, z. Následně přidáme řádek, který zajišťuje lineární posunutí. Vznikne tedy rozšířená matice:

$$\begin{pmatrix} R & v \\ 0 & 1 \end{pmatrix}$$

Rozšířená rovnice rotace 1

Kde R je matice rotace podle dané osy, vektor jsou kartézské souřadnice. Vektorové složky pro účely rotace zůstanou nulové. Vzniknou pak takovéto matice:

$$\begin{pmatrix} \cos\alpha & -\sin\alpha & 0 & 0 \\ \sin\alpha & \cos\alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Rozšířená rovnice rotace 2

Rotace není jediné, co musí být bráno v úvahu. Ještě je potřeba počítat s ramenem, které navazuje na kloub. Délku ramene je potřeba započítat. To umožňuje matice posunutí po dané ose.

Matice pro posuv vypadají jako jednotkové matice s tím rozdílem, že tři členy reprezentují souřadnice: [7]

$$\begin{pmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Matice posunutí po ose x 1

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Matice posunutí po ose y 1

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Matice posunutí po ose z 1

Pro každou kinematickou dvojici je vytvořena skupina dvou matic (matice rotace a matice posunutí). Tyto matice jsou dány do maticového součinu jak mezi sebou, tak společně s maticemi pro další kinematické dvojice.

Přičemž oblast prvního až třetího sloupce pro první až třetí řádek výsledné matice se zabývá orientací end efektoru. Další oblast vytyčuje pak čtvrtý sloupec pro první až třetí řádek, a ta je určena pro souřadnice end efektoru.

Lineární

Pokud robot obsahuje jen teleskopická ramena, mění pozici ee lineárně. Pro výpočet souřadnic se tedy používá lineární transformace (více o lineární transformaci viz. Příloha B).

Nelineární

Pokud robot obsahuje nějaký rotující kloub, tak jeho transformace není lineární (více o nelineární transformaci viz. Příloha B).

4.3.1 Přímá kinematická úloha

Pro sériové manipulátory přímá kinematika nepředstavuje větší problémy, protože každá z kartézských souřadnic se dá vyjádřit obecným předpisem.

Všeobecně lze přímou úlohu chápat jako hledání cíle. Start pro tento případ představuje bod [0, 0, 0]. Cestou jsou pak klouby a ramena robotu. Tuto cestu „volíme“ nastavením kloubů. Tudíž víme odkud kudy jdeme, ale nevíme kam. Právě takovýto problém řeší přímá kinematická úloha.

Úhlové omezení

Klouby mohou mít omezení pro povolené hodnoty, proto hodnoty musí být kontrolovány, jestli lze takový pohyb pro daný kloub žádat.

Prostor robotu

Ovšem nastavení robotu může dosáhnout hodnot pro umístění cílového bodu mimo pracovní prostor robotu. Pro takový případ potřebuje kontrolu. Program, než předá výsledek pro provedení pohybu, ověří, jestli cílový bod náleží pracovnímu prostoru. Díky úhlovému omezení pak většinou myslíme především kontrolu kolize s podložkou.

Výpočet

Všeobecný postup pro řešení přímé kinematické úlohy:

1. Přepsat robota do D–H notace nebo jiného formátu pro zápis robotu.
2. Sestavit maticový předpis pro jednotlivé členy robotu (klouby a ramena).
3. Dané matice spolu vynásobit - buď v přesném pořadí nebo upřednostnit matice kinematických dvojic, nelze však matice přehazovat.
4. Vyjádřit rovnice pro x, y a z.
5. Rovnice zjednodušit, pokud zjednodušení umožňují (**volitelný** krok, avšak vhodný pro optimalizaci).
6. Vypočítat rovnice pro požadované nastavení robotu.

4.3.2 Inverzní kinematická úloha

Jde o opačný problém, než je přímá kinematická úloha. Inverzní kinematická úloha se zabývá vztahem mezi kloubovými souřadnicemi robotu a kartézskými. [1] Jinými slovy pro tento problém jsou známy kartézské souřadnice bodu, kterého má robot dosáhnout. Na základě známých souřadnic pak inverzní kinematická úloha hledá úhlové souřadnice (cestu, kterou robot dosáhne požadovaného bodu).

Problémy, které mohou nastat při řešení inverzní kinematické úlohy: [9], [10], [11], [12], [13]

- Více možných řešení.
- Požadovaný bod mimo pracovní prostor robotu.
- Nalezení singularity (viz níže).

Singularity

Singularity jsou nebezpečné, protože pro tyto body není například jasné, jak se bude ee pohybovat při změně nastavení kloubů viz. Příloha B. Pro výpočet singulárních bodů bývá využita Jacobiho matice. Jacobiho matice obsahuje vždy stejný počet řádků jako funkcí, stejný počet sloupců jako proměnných. Jednotlivé prvky jsou pak definovány jako parciální derivace funkce/ parciální derivace proměnné.: [16], [17], [18]

$$J = \begin{pmatrix} \frac{dx}{d\alpha} & \frac{dx}{d\beta} & \frac{dx}{d\gamma} & \frac{dx}{d\delta} \\ \frac{dy}{d\alpha} & \frac{dy}{d\beta} & \frac{dy}{d\gamma} & \frac{dy}{d\delta} \\ \frac{dz}{d\alpha} & \frac{dz}{d\beta} & \frac{dz}{d\gamma} & \frac{dz}{d\delta} \end{pmatrix}$$

Sestavením jakobiánu získáme matici, pro kterou vyjádříme determinant. Dosadíme za proměnné. Pokud získáme hodnotu determinantu 0, narazili jsme na singularitu.

4.4 Komunikace

Komunikace probíhá pomocí sériového portu. Sériovým portem jsou odesílány příkazy. Javovská strana komunikace byla řešena pomocí knihovny jSerialComm.

4.4.1 Formát příkazů

Aplikace předává Arduinu příkaz, který má několik parametrů., jehož struktura je stejná jako struktura G – kódu (část příkazu „mezera“ část příkazu „mezera“ část příkazu). Pro oddělení parametrů příkazu byl použit prázdný znak neboli mezera. Příkaz se skládá z těchto parametrů:

1. Identifikátor – určuje, co se má vykonat a je vytvořen pro případnou rozšiřitelnost programu. Arduino v současné době nemá připojeno nic jiného než serva k robotu. Z tohoto důvodu Arduino neumí nic jiného než pohnout požadovaným servem na předanou hodnotu. Kdyby se k Arduinu připojilo i něco jiného než serva, například diody, vyžadoval by kód pro Arduino minimální úpravy pro přijímání zpráv, jak pro servomotory, tak pro diody.
2. Id serva – díky tomuto parametru Arduino ví, kterému servu má změnit šířku signálu.
3. Požadovaný úhel – hodnota, která udává cílený úhel pro servo.

5 Praktická část

5.1 Robot Arexx RA–1 Pro a jeho vlastnosti

- Sériový robot.
- Pohybující se v třírozměrném prostoru.
- 5 stupňů volnosti.
- 5 ramen (musí souhlasit s počtem stupňů volnosti). Rozměry ramen v cm pak jsou: {3.5; 8; 8; 7; 7.5}

5.1.1 D–H notace

Dle pravidel pro D–H notaci byl vytvořen předpis ramene *Arexx RA-1 Pro* v D–H notaci.

První kloub se otáčí okolo osy z. K tomuto kloubu není upevněno rameno, ale další kloub. Vzdálenost, kterou udává parametr d pro první kloub, je posunutí po ose z pro druhý kloub (druhý kloub má souřadnice [0, 0, 3.5]). Otočení kolem osy z je 90°, protože osa rotace pro další kloub je osa y. Tímto způsobem se zapisují i další ramena. [8]

- i – číslo kloubu
- θ - úhel pro kloub okolo osy z.
- d – posunutí po ose z (cm)
- a – posunutí po ose x (cm)
- α rotace okolo osy x

i	θ	d	a	α
1	q_1	3.5	0	90°
2	q_2	0	8	0
3	q_3	0	8	-90
4	q_4	0	7	90
5	q_5	7.5	0	0

DH notace 1

5.1.2 Hardware

Jakožto ovládací deska robotické ruky byl stanoven mikrokontroler Arduino Uno. Oficiální shieldy pro Arduino ovšem nepodporují připojení potřebných šesti servomotorů. Pro jejich ovládání máme dvě možnosti:

1. Použít více Arduin – řešení pomocí více Arduin zvyšuje finanční náročnost řešení.
2. Použít pro Arduino kompatibilní shield.

Vybraná metoda byla *použití pro Arduino kompatibilního shieldu*. Důvodem, proč byla vybrána tato metoda, je potřeba pouze jednoho Arduina a jednoho shieldu. Tím byla snížena finanční náročnost řešení. Dále se pak tímto rozhodnutím redukoval počet potřebných USB portů. Díky tomu lze veškeré příkazy posílat přes jeden USB port a není potřeba odlišovat, která serva ovládá které Arduino.

Serva

Robot se standardně dodává se 6 servomotory, které umožňují rotaci o 180°. Z těchto 6 serv jsou 4 k nalezení pod označením „S06NF-U STD“ a 2 pod „S05NF-U STD“. Úhel, o který se servomotor otočí, udává šířka pulzu, který je vyslán shieldem. Různé druhy serv mohou mít tyto hodnoty odlišné.

Vlastnosti serv, jež pro tuto práci bereme jako relevantní, jsou tyto tři:

1. Potřebný proud k provozu serv – pro serva „S06NF-U STD“ ani „S05NF-U STD“ nebyl parametr uveden v datasheetu. Proto bylo přihlédnuto ke zdroji proudu, který byl dodán, který poskytuje proud 2250 mA. [2], [3]
2. Napětí – serva „S06NF-U STD“ pracují s napětím 6V, nebo 7.2V. Pracovní napětí serv „S05NF-U STD“ je pak 4.8V a 6V. [2], [3]
3. Šířka signálu – další parametr, který není přímo uveden v datasheetech. [2], [3] Pro neuvedení tohoto parametru v datasheetech, byla prvotně použita šířka signálu pro jiné servo. Šířka byla následně upravena pro potřeby dodaných serv.

Arduino

Arduino se řadí mezi jednodeskové mikrokontrolery. Díky knihovnám, které ulehčují práci, jsou Arduina jednodušší pro programování, ale na druhou stranu funkčnost kódu je přímo závislá na použitých knihovnách. Na rozdíl od programování jednodeskového počítače bez knihoven. Pro tuto práci bylo vybráno Arduino Uno, které je nejběžnější a nejlépe zdokumentováno.

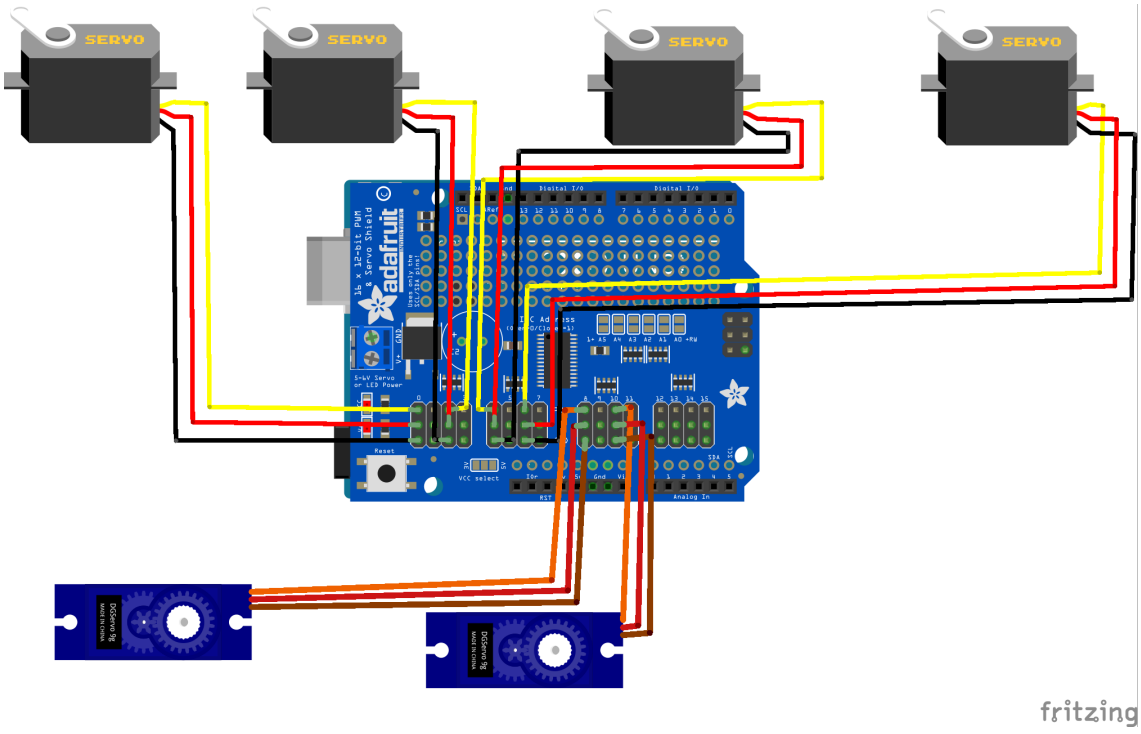
Shield

Jelikož oficiální shieldy pro Arduino Uno nepodporují připojení potřebných 6 servomotorů pomocí PWR konektorů, byl vybrán Arduino kompatibilní shield, který toto umožňuje. Lze k němu dokonce připojit až 16 servomotorů. Shield vysílá stálý PWM signál. Arduino umožňuje délku vysílaného signálu měnit podle potřeby. [19]

Shield je klon výrobku od firmy Adafruit, díky čemuž spolupracuje s jeho knihovnou, kterou je nutno stáhnout. [5] Knihovnu je pak potřeba importovat. [20]

Napájení pro servo motory nezískává z Arduina, ale z externího zdroje. Shield pracuje s vstupním napětím 3-5V. Jeho výstupním napětím je pak 6V. [19]

Zde pak vidíme schéma zapojení hardware, které bylo vytvořeno programem Fritzing.:



5.2 Kinematika našeho robotu

Náš robot obsahuje šest servomotorů. Jeden servomotor ovládá rozevření endEfectoru, další jeho orientaci, ostatní pozici. Můžeme tedy prohlásit, že robot má 4 DoF. Neexistuje pro něj prostor, kde dosáhne všech bodů pro jakoukoli orientaci endEfektoru.

5.2.1 Klouby podrobněji

Následně jsou rozepsány vlastnosti jednotlivých kloubů od základny robotu. Osy rotace byly takto stanoveny pro jednotlivé klouby robotu:

1. Kloub pro rotaci kolem osy z. Možnosti pohybu <0°;180°>. Servo: S06NF-U STD.
2. Kloub pro rotaci kolem osy y. Možnosti pohybu <0°;180°>. Servo: S06NF-U STD.
3. Kloub pro rotaci kolem osy y. Možnosti pohybu <0°;180°>. Servo: S06NF-U STD.
4. Kloub pro rotaci kolem osy y. Možnosti pohybu <0°;180°>. Servo: S06NF-U STD.
5. Kloub pro rotaci kolem osy z. Možnosti pohybu <0°;180°>. Servo: S05NF-U STD.
6. Kloub ovládající rozevření endEfectoru.

5.2.2 Ramena

Samozřejmě robot není tvořen jen klouby. Každý kloub je následován ramenem. Ramena neumožňují transformaci prostoru (nejsou teleskopická). Ovšem nutno s nimi počítat při výpočtu kinematiky.

Vlastnosti ramen od základny.:

1. Délka 3,5 cm. Osa z.
2. Délka 8 cm. Osa x.
3. Délka 8 cm. Osa x.
4. Délka 7 cm. Osa x.
5. Délka 7,5 cm. Osa x.

Pro kinematické úlohy je nejprve potřeba složit rovnici matic všech částí robotu (kloubů i ramen). Klouby jsou reprezentovány v maticovém tvaru jako matice rotace kolem osy. Ramena robotu jsou zde znázorněna jako matice posunutí po ose. [4] Výsledná maticová rovnice (přičemž q_1 je úhel prvního kloubu q_2 druhého, atp):

$$\begin{pmatrix} \cos(q_1) & -\sin(q_1) & 0 & 0 \\ \sin(q_1) & \cos(q_1) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 3.5 \\ 0 & 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} \cos(q_2) & 0 & -\sin(q_2) & 0 \\ 0 & 1 & 0 & 0 \\ \sin(q_2) & 0 & \cos(q_2) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$
$$\times \begin{pmatrix} 1 & 0 & 0 & 8 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} \cos(q_3) & 0 & -\sin(q_3) & 0 \\ 0 & 1 & 0 & 0 \\ \sin(q_3) & 0 & \cos(q_3) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} 1 & 0 & 0 & 8 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$
$$\times \begin{pmatrix} \cos(q_4) & 0 & -\sin(q_4) & 0 \\ 0 & 1 & 0 & 0 \\ \sin(q_4) & 0 & \cos(q_4) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} 1 & 0 & 0 & 7 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$
$$\times \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(q_5) & -\sin(q_5) & 0 \\ 0 & \sin(q_5) & \cos(q_5) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} 1 & 0 & 0 & 7.5 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Rovnice matic 1

Můžeme upřednostnit násobení prvků patřící jedné kinematické dvojici. Upřednostněním tohoto kroku vznikne jiný předpis pro výpočet souřadnic. Výsledek však bude stejný.

Výsledná matice byla vypočtena programem Maxima. Pro přehlednost bude použita pomocná proměnná „$M_{\text{sloupec řádek}}$“ pro každý prvek matice, přičemž C_1 je cos (prvního úhlu q_1), S_1 je sin (prvního úhlu q_1), C_2 je cos(druhého úhlu q_2), S_2 je sin(druhého úhlu q_2) a stejně pro ostatní:

$$M_{1\,1} = C_1 \times \left(C_2 \times (C_3 \times C_4 - S_3 \times S_4) + S_2 \times (-C_3 \times S_4 - S_3 \times C_4)\right)$$

$$M_{2\,1} = C_1 \times \left(S_2 \times (C_3 \times C_4 \times S_5 - S_3 \times S_4 \times S_5) + C_2 \times (C_3 \times S_4 \times S_5 + S_3 \times C_4 \times S_5)\right) - S_1 \times C_5$$

$$M_{3\,1} = S_1 \times S_5 + C_1 \times \left(S_2 \times (C_3 \times C_4 \times C_5 - S_3 \times S_4 \times C_5) + C_2 \times (C_3 \times S_4 \times C_5 + S_3 \times C_4 \times C_5)\right)$$

$$M_{4\,1} = C_1 \times \left(C_2 \times (-14.5S_3 \times S_4 + C_3 \times (14.5C_3 + 8) + 8) + S_2 \times \left(-14.5C_3 \times S_4 - S_3 \times (14.5C_4 + 8)\right)\right)$$

$$M_{1\,2} = S_1 \times \left(C_2 \times (C_3 \times C_4 - S_3 \times S_4) + S_2 \times (-C_3 \times S_4 - S_3 \times C_4)\right)$$

$$M_{2\,2} = S_1 \times \left(S_2 \times (C_3 \times C_4 \times S_5 - S_3 \times S_4 \times S_5) + C_2 \times (C_3 \times S_4 \times S_5 + S_3 \times C_4 \times S_5)\right) + C_1 \times C_5$$

$$M_{2\,3} = S_1 \times \left(S_2 \times (C_3 \times C_4 \times C_5 - S_3 \times S_4 \times C_5) + C_2 \times (C_3 \times S_4 \times C_5 + S_3 \times C_4 \times C_5)\right) - C_1 \times S_5$$

$$M_{2\,4} = S_1 \times \left(C_2 \times (-14.5S_3 \times S_4 + C_3 \times (14.5C_3 + 8) + 8) + S_2 \times \left(-14.5C_3 \times S_4 - S_3 \times (14.5C_4 + 8)\right)\right)$$

$$M_{1\,3} = C_2 \times (-C_3 \times S_4 - S_3 \times C_4) - S_2 \times (C_3 \times C_4 - S_3 \times S_4)$$

$$M_{2\,3} = C_2 \times (C_3 \times C_4 \times S_5 - S_3 \times S_4 \times S_5) - S_2 \times (C_3 \times S_4 \times S_5 + S_3 \times C_4 \times S_5)$$

$$M_{3\,3} = C_2 \times (C_3 \times C_4 \times C_5 - S_3 \times S_4 \times C_5) - S_2 \times (C_3 \times S_4 \times C_5 + S_3 \times C_4 \times C_5)$$

$$M_{4\,3} = -S_2 \times (-14.5S_3 \times S_4 + C_3 \times (14.5C_4 + 8) + 8) + C_2 \times \left(-14.5C_3 \times S_4 - S_3 \times (14.5C_4 + 8)\right) + 3.5$$

$M_{1\,4} = 0$

$M_{2\,4} = 0$

$M_{3\,4} = 0$

$M_{4\,4} = 1$

$$\begin{pmatrix} M_{11} & \cdots & M_{14} \\ \vdots & \ddots & \vdots \\ M_{41} & \cdots & M_{44} \end{pmatrix}$$

Pro účely určení polohy end efektoru jsou prvky „$M_{4\,1}$, $M_{4\,2}$, $M_{4\,3}$“, jež reprezentují součadnice x, y a z v tomto pořadí. Prvky „$M_{1\,1}$“ až „$M_{3\,3}$“ jsou zajímavé pro určení orientace end efektoru.

5.2.3 Přímá

Přímá kinematika spočívá v počítání výsledných souřadnic z daných úhlů pro jednotlivé klouby a délek jejich ramen.

Z výsledné matice jsou k určení polohy end efektoru jsou potřeba jen souřadnicové hodnoty tzn. z posledního sloupce první tři řádky. První pro výpočet hodnoty x, druhý pro y a třetí pro z. Dostaneme tyto upravené rovnice, k úpravě byl využit program WolframAlpha:

$$x = C_1 \times (14.5C_{2+3+4} + 8C_{2+3} + 8C_2)$$

$$y = S_1 \times (14.5C_{2+3+4} + 8C_{2+3} + 8C_2)$$

$$z = -14.5S_{2+3+4} - 8S_{2+3} - 8S_2 + 3.5$$

Přičemž C_1 zde reprezentuje cosinus prvního úhlu. C_2 cosinus druhého úhlu. C_{2+3} cosinus součtu druhého a třetího úhlu. C_{2+3+4} cosinus součtu druhého, třetího a čtvrtého úhlu. Obdobně pak S_1 reprezentuje sinus prvního úhlu. S_2 sinus druhého úhlu. S_{2+3} sinus součtu druhého a třetího úhlu. S_{2+3+4} sinus součtu druhého, třetího a čtvrtého úhlu.

5.2.4 Inverzní

Zpětná kinematická úloha je opak přímé, na vstupu jsou zadány výsledné souřadnice, systém pak musí zjistit dané úhly pro jednotlivá ramena.

Pro robota, s méně než šesti stupni volnosti, neexistuje prostor, ve kterém by dokázal dosáhnout jakékoli pozice a orientace v prostoru robota. Robot, pro kterého inverzní kinematika byla zkoumána obsahuje pět stupňů volnosti, tudíž není možné dosáhnout pro jakoukoli pozici libovolné orientace ee.

Nejedno možné řešení

Problémy může způsobit i vícero možných cest od robotí základny k cílovému bodu.

Vzniklá soustava tří rovnic o čtyřech neznámých:

$$x = C_1 \times (14.5C_{2+3+4} + 8C_{2+3} + 8C_2)$$

$$y = S_1 \times (14.5C_{2+3+4} + 8C_{2+3} + 8C_2)$$

$$z = -14.5S_{2+3+4} - 8S_{2+3} - 8S_2 + 3.5$$

Hodnotu pro první úhel lze získat poměrně snadno. Kolem osy z rotuje jen jeden kloub. Díky tomu lze využít vlastností pravoúhlého trojúhelníku k dopočtení sinu, nebo cosinu prvního úhlu. Nejprve vypočteme přeponu (p) pomocného trojúhelníku Pythagorovou větou:

$$p = \sqrt{x^2 + y^2}$$

Pak vypočteme sinus a cosinus dle dalších vlastností pomocného pravoúhlého trojúhelníku v následujících vzorcích:

$$C_1 = \frac{x}{p}$$

$$S_1 = \frac{y}{p}$$

Díky nalezení předpisu pro vypočtení sinu a cosinu prvního úhlu pak vznikne takováto obecná soustava rovnic:

$$x = \frac{x}{p}(14.5C_{2+3+4} + 8C_{2+3} + 8C_2)$$

$$y = \frac{y}{p}(14.5C_{2+3+4} + 8C_{2+3} + 8C_2)$$

$$z = -14.5S_{2+3+4} - 8S_{2+3} - 8S_2 + 3.5$$

V třetí rovnici jsou však sinové hodnoty, které lze přepsat do cosinových pomocí posunutí o 90° vlevo:

$$S_{2+3+4} = C_{90°-(2+3+4)}$$

Stejně tak pro ostatní úhly.

Problémem je, že v tomto stavu stále nelze vyjádřit analytický vzorec pro vypočtení jednotlivých úhlů. Pro analytické vyjádření dalších úhlů bychom napřed potřebovali alespoň jeden úhel vhodně zvolit.

Alternativou pro analytické řešení by pak bylo řešení pomocí diferenciální kinematiky. Diferenciální kinematika, ale funguje pouze pro malé vzdálenosti. Čím je větší vzdálenost aktuálního bodu od cíleného bodu, tím větší je i chyba vypočtených úhlů. [14], [15]

Implementovaný algoritmus

Pro nemožnost nalezení analytického vzorce pro výpočet jednotlivých úhlů, musel být zvolen jiný algoritmus pro výpočet inverzní kinematiky.

Možnosti byly následující. Využit algoritmu pro výpočet diferenciální kinematiky, nebo použít algoritmus „hrubé síly“. Pro účely této práce byl použit algoritmus „hrubé síly“, jen byl použit, tak aby většina výpočtů byla provedena předem, jejich výsledky obsahuje soubor, tudíž není nutno vždy provádět veškeré výpočty. Pro ilustraci máme čtyři klouby, které umožňují pohyb pro rozmezí <0°;180°>. Byly počítány kombinace vrozmení 5 ti stupňů, pro vytvoření „škatulek“, což dává celkem 1 679 616 možností pro výpočet, které nejsou počítány pokaždé. Výpočet veškerých těchto možností trvá, pro procesor Intel core i5, 2 s 291 ms. Počet jader pro tento program není podstatný, protože program byl psán jednovláknově.

Singularity

Singularity jsou nebezpečné, protože nelze přesně určit chování robotu. Výpočet singularit probíhá vypočtením jacobiho matice, následným spočtením determinatu.

Jacobiho matice.:

$$\begin{pmatrix} \frac{dx}{dq_1} & \frac{dx}{dq_2} & \frac{dx}{dq_3} & \frac{dx}{dq_4} \\ \frac{dy}{dq_1} & \frac{dy}{dq_2} & \frac{dy}{dq_3} & \frac{dy}{dq_4} \\ \frac{dz}{dq_1} & \frac{dz}{dq_2} & \frac{dz}{dq_3} & \frac{dz}{dq_4} \end{pmatrix}$$

Pro výpočet Jacobiánu byl použit program „maxima“:

```
(%i3)  jacobian ([F, G], [y, z]);
                                [ dF  dF ]
                                [ --  -- ]
                                [ dy  dz ]
(%o3)                           [        ]
                                [ dG  dG ]
                                [ --  -- ]
                                [ dy  dz ]
(%i4) jacobian ([cos(a)*(14.5*cos(b+c+d)+8*cos(b+c)+8*cos(b)), sin(a)*
(14.5*cos(b+c+d)+8*cos(b+c)+8*cos(b)), -14.5*sin(b+c+d)-8*sin(b+c)-8*sin(b)+3.5], [a,b,c,d]);
              [ - sin(a) (14.5 cos(d + c + b) + 8 cos(c + b) + 8 cos(b)) ]
              [                                                          ]
(%o4)  Col 1 = [  cos(a) (14.5 cos(d + c + b) + 8 cos(c + b) + 8 cos(b))  ]
              [                                                          ]
              [                              0                           ]
        [ cos(a) (- 14.5 sin(d + c + b) - 8 sin(c + b) - 8 sin(b)) ]
        [                                                          ]
 Col 2 = [ sin(a) (- 14.5 sin(d + c + b) - 8 sin(c + b) - 8 sin(b)) ]
        [                                                          ]
        [     - 14.5 cos(d + c + b) - 8 cos(c + b) - 8 cos(b)      ]
        [ cos(a) (- 14.5 sin(d + c + b) - 8 sin(c + b)) ]
        [                                               ]
 Col 3 = [ sin(a) (- 14.5 sin(d + c + b) - 8 sin(c + b)) ]
        [                                               ]
        [     - 14.5 cos(d + c + b) - 8 cos(c + b)      ]
        [ - 14.5 cos(a) sin(d + c + b) ]
        [                              ]
 Col 4 = [ - 14.5 sin(a) sin(d + c + b) ]
        [                              ]
        [    - 14.5 cos(d + c + b)     ]
```

Následně potřebujeme rovnice pro determinant, abychom mohli spolehlivě určit singularitu. Matice není čtvercová, proto determinant nelze přesně určit. Nelze tedy označit nějaký bod singularitou. Pokud bychom matici doplnily, aby byla čtvercová - dostaneme nepřesný výsledek, protože tím upravíme hodnotu determinantu. Pro tyto důvody singularity nebyly implementovány.

5.3 Překlad úhlových jednotek

Serva umožňují rotaci kolem osy o 180°. Pro sjednocení hodnot pro všechny klouby program umožňuje nastavit úhel nabývající hodnot od 0° do 180°. Stejně tak mapování úhlů v Arduino kódu na šířku signálu probíhá s hodnotami od 0 do 180.

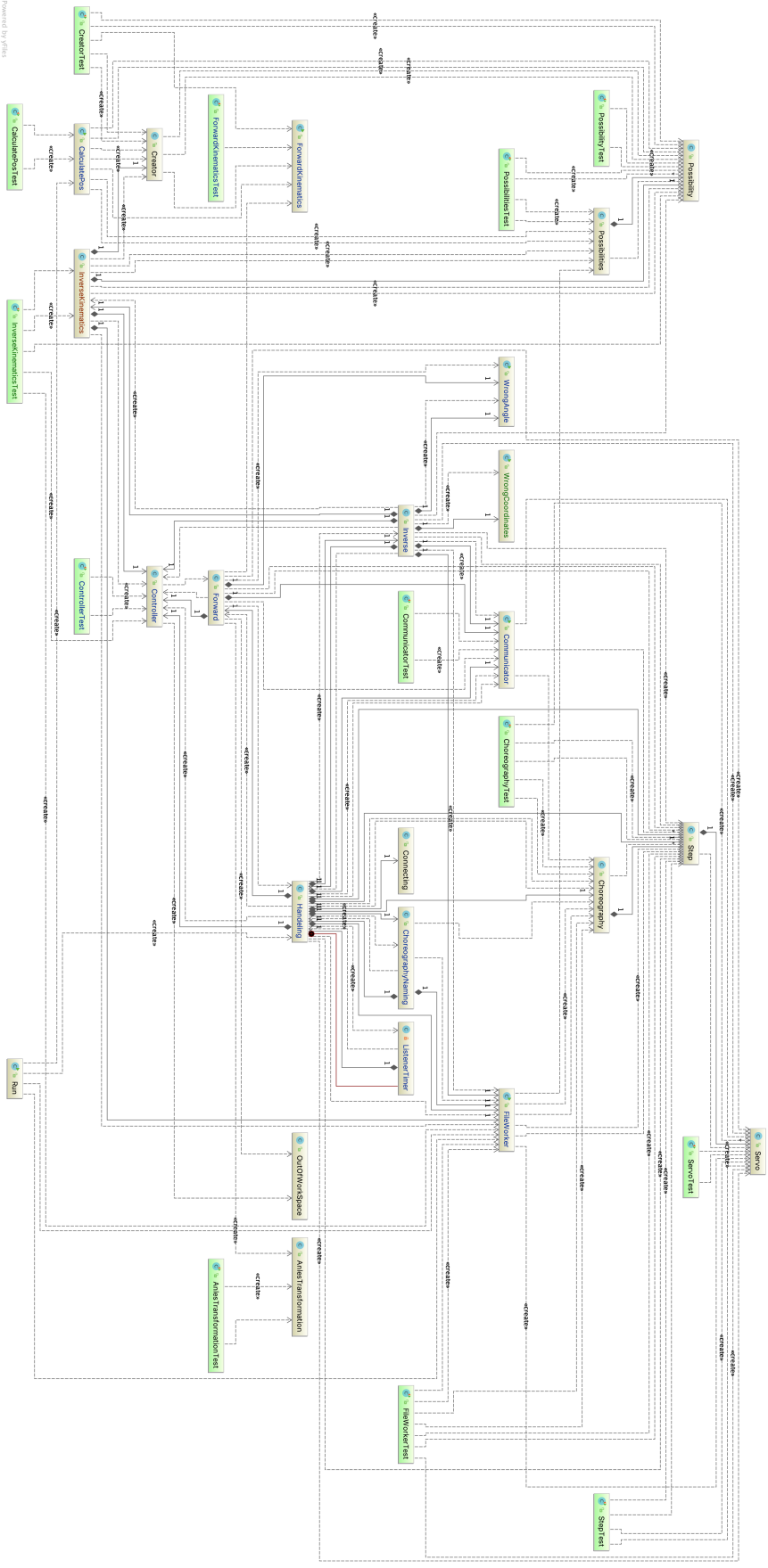
Problém nastává ve chvíli, kdy chceme spočítat přímou kinematickou úlohu. Protože pro přímou kinematiku mají serva nulový stupeň všechna na stejném místě. Ovšem napojením ramene na kloub můžeme posunout úhel, který svírá rameno s souřadnicovými osami. Mějme například robot, který se skládá z jedné kinematické dvojce. Kloub je pro dvojici rotační okolo osy z a umožňuje rotovat v rozsahu od 0°do 180°. Rameno je dlouhé 7 cm ve směru osy x (při výchozím stavu). Úhel serva je nastaven na 0°, takže očekáváme souřadnice x a y takovéto [7, 0]. Ovšem rameno svým posunutím o 45°takovéto souřadnice nemá na místo nich má pro 0°souřadnice [4.9497, 4.9497].

5.4 Optimalizace

Veškeré úkony, které program počítá byly optimalizovány matematickými úpravami. Nejen pro důvod výpočetní náročnosti, ale také paměťové. Matematické úpravy byly prováděny buď předem papírově, nebo předem použitím některého volně dostupného matematického programu viz. Příloha C.

5.5 Aplikační návrh

Celou aplikaci spustí třída „Run“. Pokračuje spuštěním třídy „Handeling“, která vytvoří grafické okno **Chyba! Nenalezen zdroj odkazů.**.V tomto okně lze najít největší část aplikace. Umožňuje ukládání pohybů, ovládání ruky ve dvou módech „realTime“ a „step by step“, dále při načtení uložené soustavy pohybů nastavit „mezičas“ mezi pohyby, které mají být vykonány. Následující obrázek znázorňuje diagram tříd celkové aplikace:


CreatorTest
CalculatePosTest
CalculatePos
Creator
ForwardKinematicsTest
ForwardKinematics
PossibilityTest
Possibility
PossibilitiesTest
Possibilities
InverseKinematics
InverseKinematicsTest
WrongAngle
WrongCoordinates
Inverse
ControllerTest
Controller
Forward
CommunicatorTest
Communicator
ChoreographyTest
Step
Connecting
Choreography
Handling
ChoreographyNaming
ListenerTimer
Run
RobWorker
OutOfWorkSpace
AnlesTransformation
AnlesTransformationTest
RobWorkerTest
ServoTest
Servo
StepTest
<create>

5.5.1 Grafika

Grafická část byla navrhována grafickým nástrojem pro tvorbu formulářových oken, kterým disponuje vývojové prostředí IntelliJ IDEA.

Takto vytvořené grafické okno obsahuje „grafickou“ stránku, která je popsána souborem .form. Soubory .form jsou XML dokumenty, které obsahují XML popis grafického rozhraní.

Další soubor, který definuje grafické rozhraní již .java, obsahuje veškerou logickou část nezbytnou pro vykonávání grafických úkonů například funkci, která přepisuje hodnoty, které jsou viditelné vedle posuvníků.

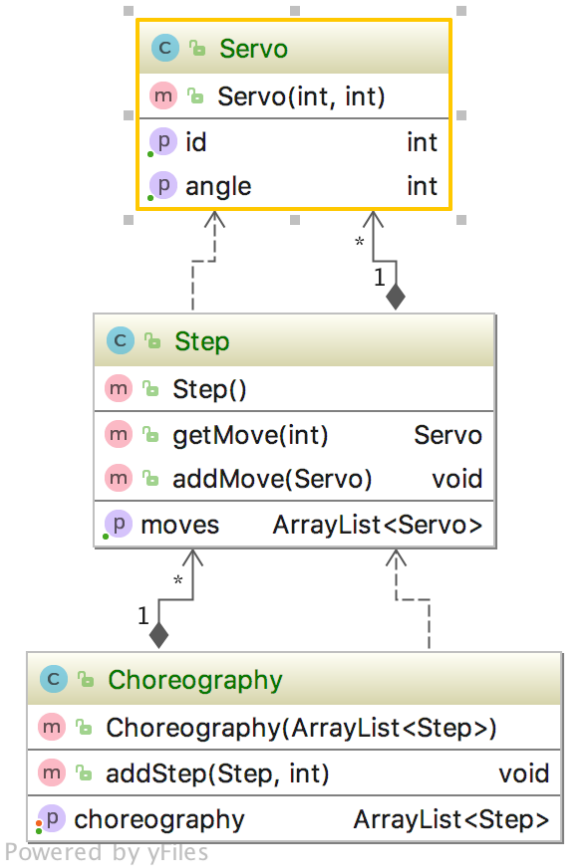
5.5.2 Logika

Logická část celé aplikace byla tvořena odděleně, funguje tedy nezávisle. Logická část, pro její rozložitost, byla rozdělena v několika „package“. Přičemž hlavním dělením jsou „objekty“, obsahující veškeré nově definované objekty pro aplikaci a „služby“, které obsahují předpisy pro vykonávání jednotlivých služeb aplikace. „Controller“ obstarává kontrolu, zda byly úhly správně zadány. A jestli bod náleží pracovnímu prostoru.

Objekty

Balíček objekty obsahuje celkem 5 tříd (Choreography, Possibilities, Possibility, Servo, Step). Objekty „Choreography“, „Servo“, „Step“ spolu souvisí, jelikož „Choreography“ obsahuje množinu „Servo“ a „Servo“ obsahuje množinu „Step“.

Pohyby servo motorů jsou ukládány objektem „Servo“, k jehož definování je potřeba označení serva a hodnoty pro dané servo. Ukládání jen těchto objektů ovšem neřeší složitější pohyby, proto jsou tyto jednotlivá nastavení seskupována v jednotlivé kroky (Step). Kroky symbolizují pohyby, které jsou prováděny současně v rámci možností programu. Dále jsou tyto kroky seskupeny v choreografii, která reprezentuje soustavu pohybů. Pro jejich uložení musejí všechny tyto objekty implementovat serializaci.
Class diagram:



Class digram pohybů 1

Služby

Systém služeb byl rozvržen několika třídami, která každá poskytuje služby podobného druhu. Například třída „FileWorker“ provádí souborové práce (čtení, zápis). Tyto třídy obsahuje celý program pouze jednou (neroste zbytečně vytížení paměti), proto musí být předávány.

6 Implementace

Pro desktopovou část programu byl vybrán programovací jazyk Java.

Arduino je nativně programovatelné prostřednictvím programovacího jazyka Wiring. Jde Arduinu „vnutit“ i další programovací jazyky, kterými jej lze ovládat, například Python. Ovšem programování Arduina pomocí jazyka Wiring je nejlépe zdokumentováno a většina knihoven, ovladačů pro hardware pracujících s Arduinem je napsána právě v jazyce Wiring. Právě pro dokumentaci Wiringu a množství Wiringových knihoven byl vybrán pro programování Arduina jazyk Wiring.

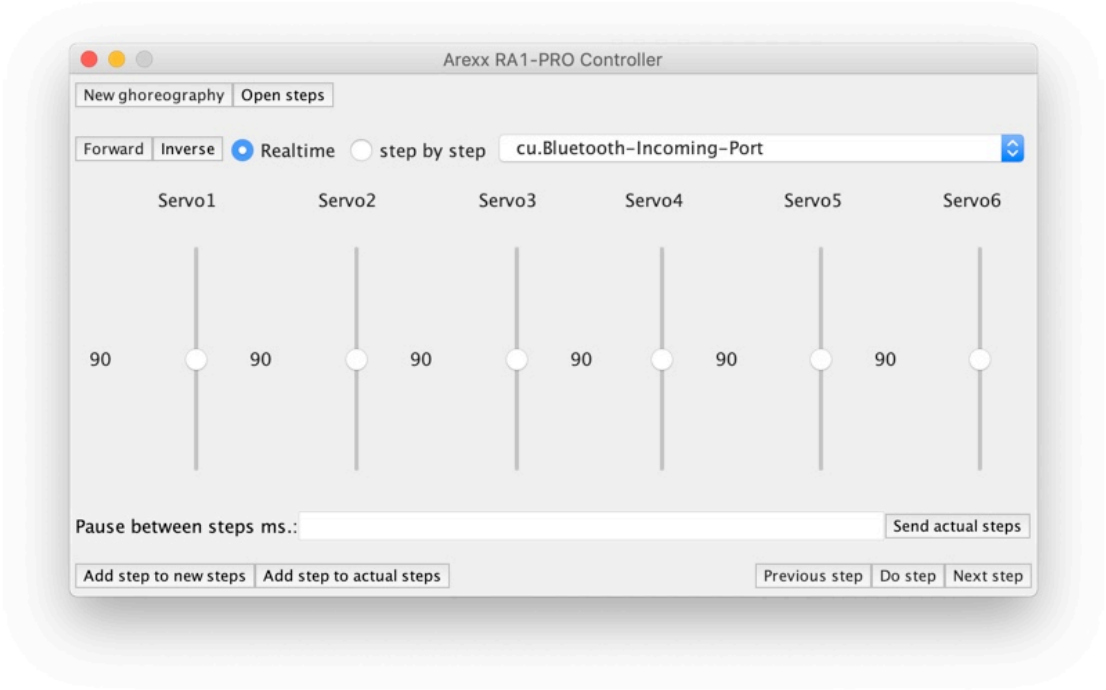
Původně byl zamýšlen prostředník pro komunikaci těchto součástí, v podobě scriptu programovaného v jazyce Python 2.7.

6.1 Java

Aplikaci v Javě lze rozdělit do několika částí, podle jednotlivých úloh.

6.1.1 Grafické rozhraní

Podoba aplikace byla vytvořena s pomocí formulářových oken. Složení těchto formulářů jsou jeden soubor formátu „form“, v kterém jsou uložena data pro vzhled jednotlivých oken, dále pak soubor formátu „java“, obsahující kódové informace jednotlivých grafických komponent například: akce tlačítek, hodnoty pro posuvníky, události… Při spuštění programu dojde k zobrazení následujícího okna:



6.1.2 Ukládání/načítání pohybů

Jak již bylo sděleno výše, pozice jednotlivých motorů jsou ukládány společně s označením serva v objektu „Servo“. Objekty jsou sdružovány do kroků „Steps“ pomocí listu. Seznamy pak tvoří dvourozměrné pole objektů, které jsou ukládány do binárního souboru pomocí serializace dat. S soubory pracuje třída „FileWorker“. V této třídě jsou dvě metody.

1. Metoda „saveChoreography“, ta choreografii přenesenou parametrem uloží pod jménem, které dostane také parametrem společně s cestou, kde má soubor být uložen. Vytvoří „FileOutputStream“ (dále FOS), který využívá pro zápis informací souboru. Následně pomocí „ObjectOutputStream“ (dále OOS) uloží daný objekt do streamu k zápisu do souboru.

```
/**
 * Method which can create a new file for you choreography.
 * @param choreography Your choreography to save.
 * @param path Path with name of the choreography.
 */
public void saveChoreography(Choreography choreography, String
path){
    try {
        FileOutputStream fileOut = new FileOutputStream(path);

        ObjectOutputStream out = new ObjectOutputStream(fileOut);
        out.writeObject(choreography);
        out.close();
        fileOut.close();

    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

2. Metoda „loadChoreography“, která choreografii z souboru načítá. Jako parametr dostane název souboru, který doplní do cesty k souboru a soubor načte.

```
/**
 * Method which serve as a loader of your choreography.
 * @param name Name of your choreography.
 * @return choreography Loaded choreography.
 */
public Choreography loadChoreography(String name){

    Choreography choreography = null;

    try {
        FileInputStream fileIn = new
FileInputStream("choreography/"+name+".bin");
        ObjectInputStream in = new ObjectInputStream(fileIn);

        try {
            choreography = (Choreography) in.readObject();
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
        in.close();
        fileIn.close();

    } catch (FileNotFoundException e) {
        // Program can not find your choreography
        System.out.println("Make new choreography.");
    } catch (IOException e) {
        e.printStackTrace();
    }
    return choreography;
}
```

6.1.3 Odesílání příkazů Arduinu

Příkazy jsou odesílány pomocí třídy „Communicator“, která obsahuje metody.:

1. Metodu „sendData“, která obdrží parametrem integerové pole. Pole obsahuje dva prvky ID serva a hodnotu, na kterou je potřeba servo nastavit. Pro odesílání příkazu pak využívá metod objektu „PrintWriter“.

```
public void sendData (int[] data) {
    PrintWriter writer = new
PrintWriter(this.serialPort.getOutputStream());

    System.out.println("m "+data[0]+" "+data[1]);
    writer.println("m "+data[0]+" "+data[1]);
    writer.flush();
}
```

2. Další metodou je znovu metoda „sendData“ jen s parametrem typu „Step“. Objekt typu „Step metoda rozloží na jednotlivé informace o servech, následně volá metodu, „sendData, které předá parametrem integerové pole. Prvním prvkem pole je id serva a druhým úhel, na který se servo má nastavit. Mezi nastavením více serv vzniká prodleva 30 ms, která je určena ozkoušením programu.

```
public void sendData(Step step){
    for (int i = 0; i < step.getMoves().size(); i++) {
        int id = step.getMoves().get(i).getId();
        int angle = step.getMoves().get(i).getAngle();
        int[] data = {id,angle};

        sendData(data);
        try {
            Thread.sleep(30);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

3. Další metodou je stále „sendData“, tentokrát parametrem získává celou „choreografii“, kterou je nutno provést. Objekt opět rozloží na menší, které posílá metodě „sendData“, která tyto objekty přijímá. Lze nastavit prodlevu mezi jednolivými kroky (Step), kterou metoda rovněž přijímá parametrem.

```
public void sendData(Choreography choreography, int waitingTime){
    for (int i = 0; i < choreography.getChoreography().size(); i++) {
        Step step = choreography.getChoreography().get(i);

        sendData(step);
        try {
            Thread.sleep(waitingTime);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

6.1.4 Přímá kinematika

Výpočet přímé kinematiky je realizován třídou „ForwardKinematics“, která obsahuje jen jednu statickou metodu „calculateFK“, která vypadá následujícím způsobem:

```
public static double[] calculateFK(int[]angles){
    double servo1Rad = Math.toRadians(angles[0]);
    double servo2Rad = Math.toRadians(angles[1]);
    double servo3Rad = Math.toRadians(angles[2]);
    double servo4Rad = Math.toRadians(angles[3]);
    double servo5Rad = Math.toRadians(angles[4]);
    double servo6Rad = Math.toRadians(angles[5]);

    double x = Math.cos(servo1Rad)*(14.5*Math.cos(servo2Rad + servo3Rad
+ servo4Rad) + 8 * Math.cos(servo2Rad + servo3Rad) + 8 *
Math.cos(servo2Rad));

    double y = Math.sin(servo1Rad)*(14.5*Math.cos(servo2Rad + servo3Rad
+ servo4Rad) + 8 * Math.cos(servo2Rad + servo3Rad) + 8 *
Math.cos(servo2Rad));

    double z = -14.5*Math.sin(servo2Rad + servo3Rad + servo4Rad)-
8*Math.sin(servo2Rad + servo3Rad)-8*Math.sin(servo2Rad)+3.5;

    double[]coordinates = {x, y, z};
    return coordinates;
}
```

Přímá kinematika 1

6.1.5 Inverzní kinematika

Část inverzní úlohy zde provádí funkce „findNearest“, která obdrží parametry pole cílových souřadnic a objekt typu Possibilities. Funkce nejprve definuje potřebné proměnné a inicializuje potřebné proměnné. Následně prochází celý seznam možností nastavení („Possibilities“), který byl vytvořen pomocí přímé kinematiky. Rozmezí jednotlivých možností je 5°. Dále vypočte rozdíl souřadnic cílového bodu s aktuálním bodem procházené možnosti. Tento rozdíl program vypočítává pro jednotlivé souřadnice zvlášť. Výsledek je uložen do proměnné „tmpdif“, jako dočasný rozdíl. Program tento rozdíl nastaví aktuálně procházené možnosti jako diferenci. Dále pak je dočasný rozdíl porovnán s aktuálním nejmenším rozdílem, který je uložen v proměnné „diffenrence“. Pokud je dočasný menší, přepíše se hodnota aktuálně nejmenšího na hodnotu dočasného. Dále bude přepsán index vrácené možnosti na index aktuálně procházené možnosti. Když projde celé pole, vrátí buď prázdnou nově vytvořenou možnost, nebo možnost jejíž bod měl nejmenší odchylku.

```
/**
 * Method finding nearest possibility.
 * @param coordinates
 * @return possibility
 */
public Possibility findNearest(double[] coordinates, Possibilities
possibilities){
    double tmpdif;
    this.diference = 100000;
    double xDif;
    double yDif;
    double zDif;
    int index = -3;
    //for all possibilities calculate difference.
    for (int i = 0; i < possibilities.getAllPossibilities().size();
i++) {
        xDif = coordinates[0] -
possibilities.getAllPossibilities().get(i).getX();
        yDif = coordinates[1] -
possibilities.getAllPossibilities().get(i).getY();
        zDif = coordinates[2] -
possibilities.getAllPossibilities().get(i).getZ();
        tmpdif = xDif + yDif + zDif;

possibilities.getAllPossibilities().get(i).setDifference(tmpdif);
        System.out.println(tmpdif);
        //remember min difference and index ofHer possibility
        if(tmpdif < this.diference){
            this.diference = tmpdif;
            index = i;
        }
    }
return index < 0 ? new Possibility() :
possibilities.getAllPossibilities().get(index);
}
```

Tímto nalezneme nejbližší možnost z aktuálního seznamu. Pro přesnější výsledek musíme vygenerovat nový seznam, který bude přesnější. Takto vygenerovaný seznam pak znovu předáme funkci „findNearest“, aby našla přesnější výsledek. Takto pracuje metoda „getPossibility“, která obdrží parametrem souřadnice cílového bodu. Dále inicializuje proměnné, přičemž hodnota proměnné „possibility“ bude inicializována jako výsledek funkce „findNearest“. Následně bude inicializován prázdný objekt typu „Possibilities“. Dále rozhodovací logika na základě odchylky možnosti, určí způsob tvoření přesnějšího seznamu – například pokud je odchylka záporná, musí navýšit hodnoty souřadnic. Vytvořený seznam předá metodě „findNearest“ a vrátí takto obdrženou hodnotu.

```
public Possibility getPossibility(double[] coordinates){
    Possibility possibility = findNearest(coordinates,
fileWorker.loadMoznosti());
    Possibilities possibilities = new Possibilities();
    //
    if(possibility.getDifference() < 0){
        for (int i = possibility.getElement(0); i >
possibility.getElement(0) - 4; i--) {
            for (int j = possibility.getElement(1); j >
possibility.getElement(1) - 4; j--) {
                for (int k = possibility.getElement(2); k >
possibility.getElement(2) - 4; k--) {
                    for (int l = possibility.getElement(3); l >
possibility.getElement(3) - 4; l--) {

possibilities.addPossibility(this.creator.createPossibility(i, j, k,
l));
                    }
                }
            }
        }
        //
        return findNearest(coordinates, possibilities);
        //
    }else if(possibility.getDifference() > 0){
        for (int i = possibility.getElement(0); i <
possibility.getElement(0) + 4; i++) {
            for (int j = possibility.getElement(1); j <
possibility.getElement(1) + 4; j++) {
                for (int k = possibility.getElement(2); k <
possibility.getElement(2) + 4; k++) {
                    for (int l = possibility.getElement(3); l <
possibility.getElement(3) + 4; l++) {

possibilities.addPossibility(this.creator.createPossibility(i, j, k,
l));
                    }
                }
            }
        }
        //
        return findNearest(coordinates, possibilities);
        //
    }else {
        //if difference 0 return actual possibility
        return possibility;
    }
}
```

6.2 Wiring (Arduino)

Pro Arduino byly vytvořeny čtyři následující funkce.

6.2.1 Loop

Metoda loop čte sériovou komunikaci jako sérii znaků. Přeskakuje „nechtěné“ znaky, například prázdné nebo newLine. Metoda také využívá pomocnou proměnnou „tokenCounter“, která stanovuje aktuální část příkazu (jestli aktuálně Arduino načítá požadovanou akci, index serva, nebo úhel). Dále používá „buffer“, kam ukládá důležité příchozí znaky. Následně hodnotu „bufferu“ transformuje pro požadovanou část příkazu. Pokud příkaz přišel již celý, volá metodu „doCommand“, resetuje „buffer“.

```
void loop() {
  // parse incomming commands
  while (Serial.available() > 0) {
    char c = Serial.read();
    Serial.print(c);
    if (lg < BUFFER_SZ - 1) {
     // on whitespace, process token buffer
     if (c == ' ' || c == '\n'){
       switch (tokenCounter){
         case 0:
           command = String(buffer);
           break;
         case 1:
           firstParam = atoi(buffer);
           break;
         case 2:
           secondParam = atoi(buffer);
           break;
       }
       // when we have all the parameters, perform a command
       if (tokenCounter >= 2){
         doCommand(command, firstParam, secondParam);
         // reset temp variables for next command
         command = "";
         firstParam = 0;
         secondParam = 0;
         tokenCounter = 0;
       } else {
        tokenCounter++;
       }
```

```
            // reset buffer for next token
            lg = 0;
            memset(&buffer[0], 0, sizeof(buffer));

          } else if (c == '\r'){
            // skip these characters
          } else {
            // add char to the token bufffer
            buffer[lg++] = c;
          }
        }
    }
}
```

6.2.2 doCommand

Funkce doCommand vykonává daný příkaz. Pokud proměná „cmd“ (command) obsauje

```
// perform a command
void doCommand(String cmd, int fp, int sp){
  if (cmd.equals("m")){
       Serial.print("Servo ");
       Serial.print(fp);
       Serial.print(" uhel ");
       Serial.println(sp);
       setPin(firstParam);
       pwm.setPWM(pin, 0, pulseWidth(secondParam));
  } else {
       // no command or unrecognized
  }
}
```

jen písmeno „m“, provede daný kód. Kód provádí kontrolní výpisy, dále volá metodu „setPin“, aby nastavila výstupní pin dle globální proměnné „firstParam“. Pak volá knihovní funkci „pwm.setPWM“, která nastaví šířku signálu dle parametru.

6.2.3 setPin

Metoda, která podle obdrženého indexu nastaví výstupní pin.

```
void setPin(int index){
  switch(index){
  case 0:
    pin = 1;
    break;
  case 1:
    pin = 3;
    break;
  case 2:
    pin = 4;
    break;
  case 3:
    pin = 6;
    break;
  case 4:
    pin = 8;
    break;
  case 5:
    pin = 10;
    break;
  }
}
```

6.2.4 pulseWidth

Nastavuje šířku signálu pro servo motory.

```
int pulseWidth(int angle)
{
  int pulse_wide, analog_value;
  if(pin == 8 || pin == 10){
    pulse_wide   = map(angle, 0, 180, 500, 2350);
  }else{
    pulse_wide   = map(angle, 0, 180, 350, 2700);
  }
  analog_value = int(float(pulse_wide) / 1000000 * FREQUENCY * 4096);
  return analog_value;
}
```

6.3 Testování

Program byl testován pomocí knihovny JUnit, která umožňuje kódové testování programu. Ovšem některé části programu takto kódově testovat nelze (například grafické rozhraní). Proto bylo kódové testování provedeno pouze pro logickou část programu, další části programu byly testovány uživatelsky viz. Příloha D.

Uživatelské testování ale neprobíhalo jen pro grafickou část aplikace. Další část, která vyžadovala uživatele bylo rozhraní pro komunikaci. Vše probíhalo díky virtuálnímu portu viz. Příloha D.

7 Vzniklé problémy a řešení těchto problémů

Posuvníky ovládají jednotlivá serva. Výběrový box vpravo umožňuje vybrat port, pro komunikaci s Arduinem. Lze vybrat i jiný port – záleží na uživateli, jestli vybere skutečně ten, který komunikuje s Arduinem. Funkce pro tlačítka jsou definovány podle popisu jednotlivých tlačítek.

7.1 Komunikace desktopové aplikace s ovládací deskou Arduino

Původně byl zamýšlen prostředník mezi Arduinem a desktopovou aplikaci script napsaný v programovacím jazyce Python. Důvod byl zastaralost knihovny RX/TX pro Javu a pohled na časovou a paměťovou náročnost programu.

7.1.1 Python

Při rozhodování, jakou verzi Pythonu využít, hrála roli funkčnost knihoven. Pro Python 2.7 je dostupná knihovna pyserial, která umožňuje z popisku připojeného zařízení zjistit, o jaké zařízení jde. Uživateli by pak byly nabízeny jen porty, na kterých je připojeno Arduino. Alternativa knihovny pyserial pro Python 3.6 tuto funkci neumožňovala. Proto programovací jazyk pro script byl vybrán Python 2.7.

Ovšem problém byl spustit script z programu napsaného v Javě. Po vytvoření prostředí pro vykonávání příkazů v příkazovém řádku tzv. RunTime prostředí, bylo možné vykonat příkaz, tedy i spustit pythonovský script. Ovšem není vhodné, aby se stále spouštěl celý script, pro jeho pár procent a mít script pro vše, také není nejlepší. Kód byl tedy přepsán do metod, které měly být volány jednotlivě pomocí příkazu.:

```
"python – c'form jmenoScriptu import jmenoMetody; jmenoMetody()'"
```

Pokud by script byl uložen ve stejné složce jako javovská aplikace, nevznikal by žádný problém. Jelikož scripty mohou být umístěny jinde, Vše by bylo v pořádku, pokud by složka, z které se spouští javovský program byla stejná, jako složka, ve které je uložen script. Jelikož tomu tak nemusí být (rozložení souborů uživatelem nelze ovlivnit). Nabízela se možnost zadat absolutní/relativní cestu, nebo před příkazem pro spuštění scriptu vybrat jiný adresář. Java ovšem příkazy brala úplně odděleně, dokonce výsledek prvního nebyl brán v potaz.

7.1.2 Java

Pro výše popsané důvody bylo provedeno přehodnocení využití Javy i pro sériovou komunikaci.

Knihovna

Knihovny pro programovací jazyk Java, které zprostředkovávají sériovou komunikaci, jsou dostupné tři:

1. RX/TX – Knihovna použita v původním řešení programu.
2. JSerialCom – Nástupce knihovny RX/TX.
3. Arduino – Nástavba pro kniovnu JSerialCom, která umožňuje přímé zacházení s Arduinem například výběr portů s Arduinem.

Knihovna přímo pro komunikaci s Arduinem nebyla vybrána, protože při vytváření aplikace neměla stabilní verzi, pouze beta.

S využitím knihovny pro Javu, která se jmenuje „JSerialCom“, bylo vytvořeno rozhraní pro sériovou komunikaci aplikace s Arduinem. Knihovna je dostupná pod licencí Apache2.

Ladění programu

První potíž vyvstala při vyzkoušení této komunikace. Odesílané instrukce nebyly Arduinem vykonávány dle očekávání (Arduino buď nedělalo nic, nebo vykovávalo jiné pohyby). Pro ukázku sériové komunikace byla vytvořena odpověď, která obsahovala příchozí zprávu. Vzniklý problém byl v kódování posílaných informací. Původně byl vytvořen StreamWriter s parametrem outputStreamu sériového portu. Následovalo využití metody „write“, která odeslala Integer. Arduino ovšem nezpracovávalo příchozí data dle očekávání.

K vytvoření lepší představy o znacích, které jsou posílány, bylo potřeba vytvořit virtuální sériový port. Pro vytvoření virtuální sériové komunikace, která existuje pro systém Windows, byl program pouštěn systémem Windows. Jednu stranu reprezentovala Javovská aplikace, druhou stranu představoval program TeraTerm – ten vypisoval příchozí znaky. Forma dat, která byla posílána metodou „write“ byla nečitelná

(pravděpodobně vlivem kódování). Dalším pokusem bylo vytvoření PrintWriteru s parametrem outputStreamu sériového propojení. PrintWriter použil pro poslání informací metodu print. Výstup programu nyní vypadal dle očekávání.

Další problém byl v odesílaných znacích pro „newLine“, které Arduino pomocí metody „Serial.parseInt()“ mělo tendenci zpracovávat také. Proto bylo použito čtení znak po znaku a tyto „nechtěné“ znaky jsou přeskakovány.

7.2 Nemožnost analytického řešení

Pro nemožnost nalezení analytického vyjádření proměnných pro výpočet inverzní kinematiky, musela být použita jiná metody pro řešení.

7.3 Nepřesný hardware

Práce obsahuje poměrně přesné výpočty. Výsledky, prováděné hardware, již tak přesné nejsou. Serva jsou opotřebována, takže snadno vlivem okolí mění pozici. Dále pak nereagují pro několik krajních úhlů. Nutno tedy počítat s ovládáním serv pro rozmezí přibližně <5°; 175°>. Tento problém nemají jen tyto serva, většina serv obecně pro krajní úhly nepracuje dle předpokladu.

8 Diskuze

Program umožňuje ovládání robotické ruky, pomocí mikrokontroleru Arduino Uno, jak v reálném čase s minimálním zpožděním, tak ovládání po krocích.

Dovoluje „zaznamenat“ jednotlivé kroky, které je následně možnost uložit do souboru. Soubory lze načítat a procházet si jednotlivé kroky, bez nutnosti jejich provedení robotickou rukou. Provedení právě načteného kroku umožňuje tlačítko „Do step“.

Program umožňuje řešit přímou kinematickou úlohu, pro pozici robotu, avšak pracovní prostor robota je hlídán jen kódově. Pro změnu pracovního prostoru je nutnost zasáhnout do kódu.

Aplikace řeší inverzní kinematickou úlohu, pro pozici robotu, pomocí nového algoritmu, který vychází z algoritmu „hrubé síly“.

9 Závěr

Bakalářská práce splnila všechny své cíle (ovládání robotické ruky, ukládání a načítání pohybů, programové testování ovládání, dokumentace složitějšího kódu a počítání přímé kinematické úlohy). Program funguje dle očekávání. Díky formátu příkazu umožňuje rozšíření příkazů pro robotickou ruku, se zásahem do Arduino kódu (definování akce pro nový příkaz).

Pro ovládání Arduina lze použít libovolný program, který umožňuje komunikovat prostřednictvím sériového portu. Odesílané příkazy musí mít stejný formát, aby je Arduino dokázalo správně interpretovat.

Program navíc oproti původnímu umí počítat přímou kinematickou úlohu. Díky vyjádření jednotlivých neznámých pomocí zjednodušených vzorců programu stačí vypočítat tyto vzorce. Z tohoto důvodu nemusí pro každou úlohu násobit matice, z kterých by následně proměnné vyjádřil. Tím se snižuje náročnost programu a zrychluje výpočet.

9.1 Porovnání

Krátká tabulka pro porovnání původního a nového řešení.

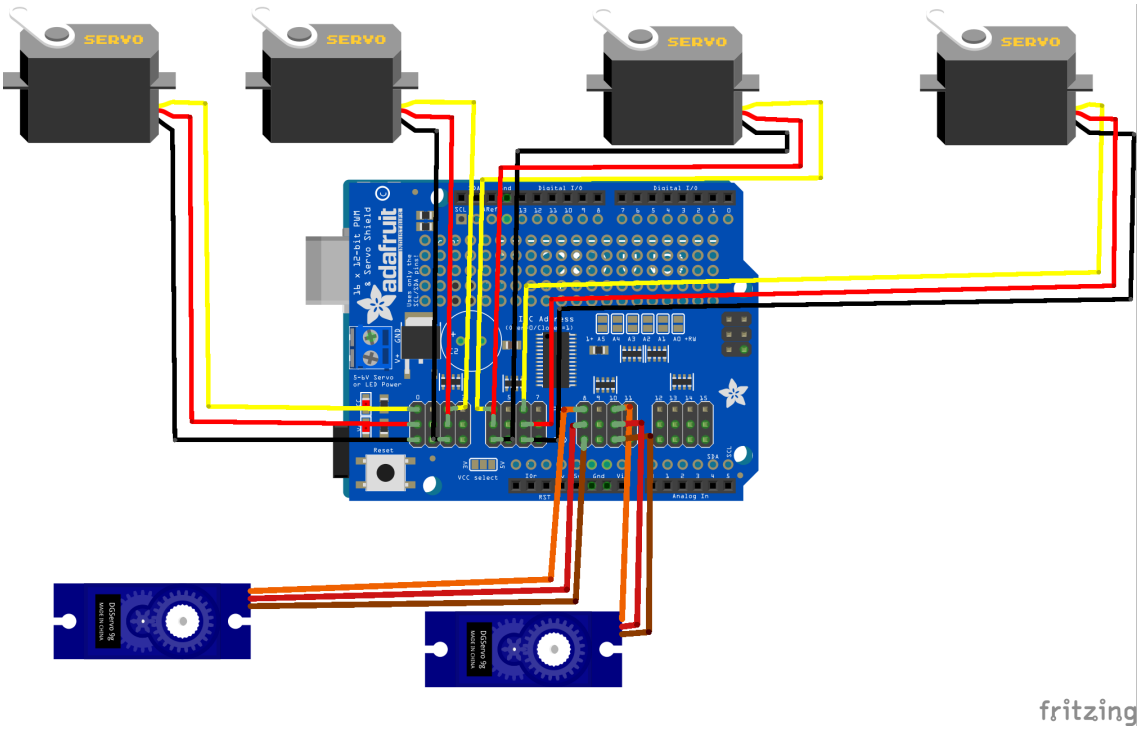
Vlastnosti	Původní řešení	Nové řešení
Arduino	Ne	Ano
Verze Javy	6	10
Funkčnost pro nové OS	Ne	Ano
Potřeba hardwarového překladače	Ano	Ne
Schopnost řešit přímou kinematickou úlohu	Ne	Ano
Schopnost řešit inverzní kinematickou úlohu	Ne	Ano
Rozšiřitelnost	Ne	Ano

9.2 Rozpočet

Součástka	Kč
Arduino	659,--
Shield	188,--
Robotická ruka	7 590,--
Celkem	8 437,--

9.3 Instalace

1. Nasadit zakoupený shield na řídicí desku Arduino Uno.
2. Připojit servomotory dle schéma zapojení.



3. Stáhnout repozitář https://github.com/StepanMudra/Arexx-RA-1-Pro-Control.
4. Použitá knihovna není zahrnuta v github repozitáři, proto při stažení zdrojového kódu musí být stažena a importována dodatečně. Knihovna je dostupná ke stažení z těchto stránek.: http://fazecast.github.io/jSerialComm/

5. Následně je potřeba knihovnu importovat. Importování knihovny uděláte ve svém vývojovém prostředí pomocí Project structure -> Project settings -> Libraries -> ikona + -> Java -> následně vybrat stažený soubor .jar.
6. Importovat Arduino knihovnu
    a. Knihova je dostupná na adrese: https://github.com/adafruit/Adafruit-PWM-Servo-Driver-Library
    b. V Arduino IDE pak: Projekt -> Přidat knihovnu -> Přidat Zip. knihovnu a vybrat Zip. soubor.
7. Nahrát Arduino kód.

10 Použitá literatura

[1] GOUBEJ, Martin, Martin ŠVEJDA a Miloš SCHLEGEL. *Úvod do mechatroniky, robotiky a systémů řízení pohybu* [online]. Plzeň, 2012 [cit. 2018-02-07]. Dostupné z: http://home.zcu.cz/~msvejda/URM/materialy/Uvod%20do%20mechatroniky.pdf. Scripta. Katedra Kybernetiky, Západočeská univerzita v Plzni.

[2] *S05NF STD* [online]. [cit. 2018-03-17]. Dostupné z: https://cdn.sparkfun.com/datasheets/Robotics/S05NF%20STD.pdf

[3] *S06NF STD* [online]. [cit. 2018-03-17]. Dostupné z: https://cdn.sparkfun.com/datasheets/Robotics/S06NFSTD.pdf

[4] ŠOLC, František a Luděk ŽALUD. *Robotika* [online]. Brno, 2002 [cit. 2018-03-11]. Dostupné z: http://media1.wgz.cz/files/media1:5100dca52f8f1.pdf.upl/Robotika.pdf. Scripta. Vysoké učení technické v Brně, fakulta elektrotechniky a komunikačních technologií.

[5] Adafruit-PWM-Servo-Driver-Library. *GitHub* [online]. Adafruit, 2012 [cit. 2018-03-10]. Dostupné z: https://github.com/adafruit/Adafruit-PWM-Servo-Driver-Library

[6] PRODUCTION, CVN. D-H notation. In: *YouTube* [online]. New York: CVN Production, 2017 [cit. 2018-04-11]. Dostupné z: https://www.youtube.com/watch?v=w5RXVHHfjy4

[7] Denavit-Hartenberg Notation. In: *YouTube* [online]. CVN Production, 2017 [cit. 2018-01-18]. Dostupné z: https://www.youtube.com/watch?v=8bty4buvOVs

[8] DH Example: 6DOF Robot. *YouTube* [online]. CVN Production, 2017 [cit. 2018-02-20]. Dostupné z: https://www.youtube.com/watch?v=ANcyqhkbqKA

[9] Inverse Kinematics. In: *YouTube* [online]. CVN Production, 2017 [cit. 2018-02-15]. Dostupné z: https://www.youtube.com/watch?v=M-U77bOQCD4

[10] *Inverse Kinematics* [online]. In: . CVN Production, 2017 [cit. 2018-04-18]. Dostupné z: https://www.youtube.com/watch?v=TuyFdfQQnb4

[11] In: *YouTube* [online]. CVN Production, 2017 [cit. 2018-02-15]. Dostupné z: https://www.youtube.com/watch?v=kYBGUiOYKFg

[12] Inverse Kinematics. In: *YouTube* [online]. CVN Production, 2017 [cit. 2018-02-10]. Dostupné z: https://www.youtube.com/watch?v=LtnUxTcx124

[13] Inverse Kinematics. In: *YouTube* [online]. CVN Production, 2017 [cit. 2018-02-10]. Dostupné z: https://www.youtube.com/watch?v=62g_1bCm4vE

[14] Differential Kinematics. In: *YouTube* [online]. CVN Production, 2017 [cit. 2018-01-25]. Dostupné z: https://www.youtube.com/watch?time_continue=1&v=niKuAVowT7o

[15] Differential Kinematics. In: *YouTube* [online]. CVN Production, 2017 [cit. 2018-01-25]. Dostupné z: https://www.youtube.com/watch?time_continue=6&v=Yb8oRU7wMHM

[16] Manipulator Jacobian. In: *YouTube* [online]. CVN Production, 2017 [cit. 2018-02-05]. Dostupné z: https://www.youtube.com/watch?v=yq0GbFNM-vY

[17] Manipulator Jacobian. In: *YouTube* [online]. CVN Production, 2017 [cit. 2018-02-05]. Dostupné z: https://www.youtube.com/watch?v=3QAUExDtAPk

[18] Singularities. In: *YouTube* [online]. CVN Production, 2017 [cit. 2018-02-05]. Dostupné z: https://www.youtube.com/watch?time_continue=556&v=KQid7QExWCA

[19] ADA, Lady. *Adafruit 16-channel PWM/Servo Shield* [online]. [cit. 2018-03-05]. Dostupné z: https://drive.google.com/file/d/0B4B30jzMyzG8Q1E4SjFrd0tSYU0/view

[20] How to Control Servo Motor Up To 16 with Arduino Uno R3 | Multiple Servo. In: *YouTube*[online]. Mert Arduino and Tech, 2017 [cit. 2018-02-03]. Dostupné z: https://www.youtube.com/watch?v=QjX4JKU_I9M

[21] Jacobian prerequisite knowledge. In: *Khan Academy* [online]. 2017, 2017 [cit. 2018-12-11]. Dostupné z: https://www.khanacademy.org/math/multivariable-calculus/multivariable-derivatives/jacobian/v/jacobian-prerequisite-knowledge

[22] Local linearity for a multivariable function. In: *Khan Academy* [online]. 2017 [cit. 2018-12-12]. Dostupné z: https://www.khanacademy.org/math/multivariable-calculus/multivariable-derivatives/jacobian/v/local-linearity-for-a-multivariable-function

[23] The Jacobian matrix. In: *Khan Academy* [online]. 2017 [cit. 2018-12-12]. Dostupné z: https://www.khanacademy.org/math/multivariable-calculus/multivariable-derivatives/jacobian/v/the-jacobian-matrix

[24] Computing a Jacobian matrix. In: *Khan Academy* [online]. 2017, 2017 [cit. 2018-12-12]. Dostupné z: https://www.khanacademy.org/math/multivariable-calculus/multivariable-derivatives/jacobian/v/computing-a-jacobian-matrix

[25] The Jacobian Determinant. In: *Khan Academy* [online]. 2017, 2017 [cit. 2018-12-12]. Dostupné z: https://www.khanacademy.org/math/multivariable-calculus/multivariable-derivatives/jacobian/v/the-jacobian-determinant

11 Příloha A

11.1 Algoritmus pro inverzní kinematickou úlohu.

1. Načti předpřipravený seznam kombinací úhlů (rozmezí pět stupňů) a souřadnic.
2. Nastav proměnnou „diference“, která znázorňuje rozdíl souřadnic, výchozí hodnotou.
3. Nastav pomocnou proměnnou pro indexaci „i“, hodnotou 0.
4. Načti kombinaci i.
5. Vypočti odchylku pro souřadnice x, y, z.
6. Sečti tyto odchylky a přepiš výslednou hodnotou hodnotu proměnné „tmpDif“, jež reprezentuje „dočasnou“ hodnotu odchylky.
7. Pokud „tmpDif“ < „diference“, pokračuj bodem 8, pro jiný případ pokračuj bodem 12.
8. Proveď kontrolu, jestli bod náleží pracovnímu prostoru. Pokud patří Pokračuj bodem 9, pokud nepatří tak bodem 12.
9. Přepiš hodnotu „diference“ hodnotou „tmpDif“
10. Hodnotu „index“ přepiš hodnotou indexu právě kontrolované kombinace.
11. Iteruj proměnnou „i“.
12. Pokud není aktuální index větší než nejvyšší dostupný, pokračuj bodem 4.
13. Vypočti další seznam. Rozhodovací logika určí, jestli hodnoty úhlů budou navyšovány, nebo snižovány. Pro oba případy pokračuj bodem 1. Pokud „diference“ výsledné možnosti pro požadovanou = 0 vrať výslednou možnost.

12 Příloha B

Předmětem této přílohy jsou singularity. Nejprve tato příloha obsahuje základy lineární algebry. Tyto základy pomohou při celkovém pochopení singularit, jejich významu a funkce.

12.1 Transformace prostoru

Transformace prostoru je zobrazována pomocí násobení matic s vektory os. Počet řádků i sloupců matic souhlasí s počtem dimenzí prostoru. Pokud tedy jde o L^2 prostor, transformační matice transformace prostoru obsahuje 2 sloupce, 2 řádky. Pro lepší představitelnost, zde bude používán právě dvoudimenzionální prostor. [21]

12.1.1 Lineární

Základní vektory os, vždy obsahují jedničku, pro řádek osy, po které mohou měnit hodnotu. Pro další osu/osy obsahují nulu. Základní vektor osy x, pro dvoudimenzionální prostor, tedy vypadá takto: [21]

$$\begin{pmatrix}1\\0\end{pmatrix}$$

Základní vektor osy y takto:

$$\begin{pmatrix}0\\1\end{pmatrix}$$

Mějme tedy matici, která znázorňuje transformaci prostoru:

$$\begin{pmatrix}1 & 2\\3 & 4\end{pmatrix}$$

Následně uděláme součin s každým vektorem osy zvlášť.

$$\begin{pmatrix}1 & 2\\3 & 4\end{pmatrix} \times \begin{pmatrix}1\\0\end{pmatrix} = \begin{pmatrix}1\\3\end{pmatrix}$$

$$\begin{pmatrix}1 & 2\\3 & 4\end{pmatrix} \times \begin{pmatrix}0\\1\end{pmatrix} = \begin{pmatrix}2\\4\end{pmatrix}$$

Nyní je vidět, že souřadnice koncového bodu vektoru osy x jsou [1; 3], nikoliv [1; 0]. Koncový bod vektoru osy y nyní náleží souřadnicím [2; 4], místo [0; 1].

Násobení transformační matice s vektory os nás dovedlo ke stejnému výsledku jako přepis transformace pomocí transformační matice vynásobenou vektorem s proměnnými x a y.:

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \times \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 1x + 2y \\ 3x + 4y \end{pmatrix}$$

Čímž po dosazení hodnot dostáváme stejný výsledek pro vektory.

Kdybychom přidávali nekonečně malé hodnoty pro x dostatečně dlouho, označili každý jednotlivý bod, vznikne **polopřímka**. Právě proto, že spojnice bodů nebyla nikterak pokřivena, můžeme tuto transformaci nazvat lineární. [21]

12.1.2 Nelineární

Nelineární transformaci poznáme tak, že při vytvoření spojnice bodů splňující podmínky funkce, vznikne **křivka**. [22]

Jak takovouto nelineárnost zajistit?

Nelineárnosti docílíme přidáním některé trigonometrické funkce do funkčního předpisu pro transformaci. Mějme funkci f, která vypadá následujícím způsobem: [22]

$$f\left(\begin{pmatrix} x \\ y \end{pmatrix}\right) = \begin{pmatrix} x + \cos(y) \\ y + \cos(x) \end{pmatrix}$$

Pro takovýto funkční předpis, když budeme dostatečně dlouho navyšovat hodnoty pro x, při ponechání hodnoty pro y, označíme jednotlivé body, propojením těchto bodů vznikne křivka. Pro tento případ sinusoida. Proto můžeme transformaci označit jako nelineární, jelikož hodnoty body nemají lineární postup grafem.

Lokální lineárnost

Tento fenomén označuje jev, díky kterému lze křivku pro její jednotlivé, přiblížené úseky vnímat jako polopřímku. [22]

Pro ilustraci použijeme vodováhu. Kdyby někdo z neznámého důvodu chtěl postavit přibližně 40 100 kilometrů dlouhou dálnici. Pro ověření rovnosti dálnice použil vodováhu, tak při dokončení práce zjistí, že končí kde začal. Přitom jednotlivé úseky dálnice byly rovné. Ačkoli Země má jiné důvody, proč být kulatá než, že její funkční předpis obsahuje goniometrické funkce, zde byla uvedena pouze jako příklad pro ilustraci lokální lineárnosti.

Díky tomuto můžeme i pro nelineární transformaci pro velmi malé úseky použít lineární transformaci s minimální chybou. [22]

12.2 Jacobiho matice

Abychom dokázali pro zvětšené úseky nelineární transformace použít lineární transformaci, musíme nejprve sestavit Jacobiho matici. [23]

Velikost Jacobiho matice definuje počet dimenzí (pro výpočet souřadnice jedné dimenze existuje právě jedna funkce) a počet proměnných, které ovlivňují výsledek funkcí. Počet dimenzí (funkcí) udává počet řádků matice. Počet proměnných pak znamená počet sloupců matice. [23]

Mějme tedy stejnou funkci f, jako pro ukázku nelineární transformace.:

$$f\left(\begin{pmatrix} x \\ y \end{pmatrix}\right) = \begin{pmatrix} x + \cos(y) \\ y + \cos(x) \end{pmatrix}$$

Pro takovýto případ rozměry Jacobiho matice budou 2x2.

Funkce pro výpočet souřadnic jednotlivých dimenzí pojmenujeme:

$$f_x = x + \cos(y)$$

$$f_y = y + \cos(x)$$

Obsah Jacobiho matice

Členy Jacobiho matice vždy obsahují zlomek. Čitatelem pak určíme parciální derivaci funkce, jmenovatelem parciální derivace proměnné. Nelze použít jeden stejný zlomek pro každý člen matice. Funkce, pro kterou bude prováděna derivace bude pro **každý** jednotlivý sloupec matice rozdílná. Výměnu derivovaných proměnných pro jmenovatel provedeme pro každý řádek. Výsledná matice pak bude vypadat takto: [24]

$$\begin{pmatrix} \frac{df_1}{dx} & \frac{df_2}{dx} \\ \frac{df_1}{dy} & \frac{df_2}{dy} \end{pmatrix}$$

Touto maticí jsme schopni určit souřadnice směrových vektorů pro každý bod.

12.3 Význam singularit

Singularitou lze označit bod, kterého funkční křivky jsou vlivem transformace „slisované“, jinými slovy splývají.

Funkční křivkou rozumějme křivku vytvořenou spojením mnoha bodů při stejné proměnné y, pro rozdílné hodnoty x-ové proměnné.

Splynutí křivek měříme pomocí determinantu Jacobiho matice pro daný bod. Determinant pro tento případ určuje míru odtažení křivek. Čím větší hodnotu má determinant, tím více jsou křivky jednotlivě rozlišitelné. [25]

Vlivem splynutí těchto křivek tedy nelze přesně určit pohyb robotu z takového bodu, při jakékoli změně nastavení parametrů.

13 Příloha C

13.1 Optimalizace

Jelikož program řeší jen kinematické úlohy konkrétní ruky, můžeme „vypustit“ některé prvky, nutné pro výpočet kinematiky.

Pokud by program však měl řešit kinematickou úlohu pro různé robotické ruce včetně orientace, všechny prvky jsou nutné. Některé prvky by dokonce musely být variabilní (matice rotace podle osy rotace ramene, …).

13.1.1 Programové neřešení matic

Ano, dokonce můžeme vypustit matice. Netřeba, aby program stále dokola počítal násobení několika matic, když souřadnice reprezentuje jen několik prvků výsledné matice.

Nejen, že vypuštěním matic ulehčíme procesoru, ale také dostaneme lepší odezvu pro výpočet.

Pro představu uvádím ještě jednou maticovou rovnici pro výpočet výsledné matice.:

$$\begin{pmatrix} \cos(q_1) & -\sin(q_1) & 0 & 0 \\ \sin(q_1) & \cos(q_1) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 3.5 \\ 0 & 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} \cos(q_2) & 0 & -\sin(q_2) & 0 \\ 0 & 1 & 0 & 0 \\ \sin(q_2) & 0 & \cos(q_2) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$
$$\times \begin{pmatrix} 1 & 0 & 0 & 8 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} \cos(q_3) & 0 & -\sin(q_3) & 0 \\ 0 & 1 & 0 & 0 \\ \sin(q_3) & 0 & \cos(q_3) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} 1 & 0 & 0 & 8 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$
$$\times \begin{pmatrix} \cos(q_4) & 0 & -\sin(q_4) & 0 \\ 0 & 1 & 0 & 0 \\ \sin(q_4) & 0 & \cos(q_4) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} 1 & 0 & 0 & 7 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$
$$\times \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(q_5) & -\sin(q_5) & 0 \\ 0 & \sin(q_5) & \cos(q_5) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} 1 & 0 & 0 & 7.5 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Vhodnějším řešením než pokaždé při výpočtu přímé kinematiky nechat procesor počítat takovouto soustavu matic, pak vypočítat a použít jen některé prvky, bylo vypočíst výslednou matici buď ručně, nebo použít jiný program pro její výpočet.

Byla vybrána možnost jiného programu. Použitým programem byla Maxima.

Následujícím krokem, jelikož neřešíme orientaci robotu byl výběr prvků, které jsou potřeba pro výpočet souřadnic.

13.1.2 Úprava důležitých vzorců

Pro úpravu důležitých vzorců pomocí matematických úprav byl použit program WolframAlpha. Tento program vytvořil jednodušší alternativní zápis vzorců. Tento zápis následně byl implementován.

13.1.3 Výsledek

Výsledkem bylo snížení výpočetní náročnosti programu. Ovšem vzrostla tím paměťová náročnost, ale nevýrazně.

Soubor, který obsahuje veškeré kinematické možnosti nastavení pro hodnoty <0°; 360°> (72^4) byl vypočten pod 7 vteřin, včetně kontroly souřadnic, jestli náleží pracovnímu prostoru.

14 Příloha D

14.1 Kódové testování

14.1.1 Objektové testování

Servo test

```
package Pc.Logic.Objects;

import org.junit.Test;

import static org.junit.Assert.*;

public class ServoTest {

    @Test
    public void getId() {
        Servo servo = new Servo(0, 150);
        assertTrue(servo.getId() == 0);
    }

    @Test
    public void getAngle() {
        Servo servo = new Servo(5, 15);
        assertTrue(servo.getAngle() == 15);
    }
}
```

Step test

```
package Pc.Logic.Objects;

import org.junit.Test;

import static org.junit.Assert.*;

public class StepTest {

    @Test
    public void getMoves() {
        Step step = new Step();
        Step stepNew = new Step();
        assertEquals(stepNew.getMoves(), step.getMoves());
        step.addMove(new Servo(3, 15));
        stepNew.addMove(new Servo(3, 15));
        assertEquals(stepNew.getMoves().get(0).getAngle(),
step.getMoves().get(0).getAngle());
        assertEquals(stepNew.getMoves().get(0).getId(),
step.getMoves().get(0).getId());
    }

    @Test
    public void getMove() {
        Step step = new Step();
        Servo servo = new Servo(3, 15);
        step.addMove(servo);
        assertEquals(servo, step.getMove(0));
    }

    @Test
    public void addMove() {
        Step step = new Step();
        assertEquals(0, step.getMoves().size());
        Step stepNew = new Step();
        stepNew.addMove(new Servo(3, 15));
        stepNew.addMove(new Servo(1, 15));
        stepNew.addMove(new Servo(4, 15));
        assertEquals(3, stepNew.getMoves().size());
    }
}
```

Choreography test

```
package Pc.Logic.Objects;

import org.junit.Test;

import java.util.ArrayList;

import static org.junit.Assert.*;

public class ChoreographyTest {

    @Test
    public void getSteps() {
        ArrayList<Step> steps = new ArrayList<>();
        Choreography choreography = new Choreography(steps);
        assertEquals(steps, choreography.getSteps());
    }

    @Test
    public void setSteps() {
        ArrayList<Step> steps = new ArrayList<>();
        Choreography choreography = new Choreography(steps);
        Step step = new Step();
        step.addMove(new Servo(0,180));
        step.addMove(new Servo(1,0));
        step.addMove(new Servo(2,18));
        step.addMove(new Servo(3,15));
        step.addMove(new Servo(4,90));
        step.addMove(new Servo(5,160));
        steps.add(step);
        assertEquals(steps, choreography.getSteps());

        ArrayList<Step> steps1 = new ArrayList<>();
        Choreography choreography1 = new Choreography(steps);
        Step step1 = new Step();
        step1.addMove(new Servo(0,150));
        step1.addMove(new Servo(1,0));
        step1.addMove(new Servo(2,18));
        step1.addMove(new Servo(3,15));
        step1.addMove(new Servo(4,90));
        step1.addMove(new Servo(5,160));
        steps.add(step);
        choreography.setSteps(steps1);
        assertEquals(steps, choreography1.getSteps());
        assertEquals(steps.get(0), choreography1.getSteps().get(0));
        assertEquals(steps.get(0).getMoves(),
choreography1.getSteps().get(0).getMoves());
        assertEquals(steps.get(0).getMoves().get(0),
choreography1.getSteps().get(0).getMoves().get(0));
        assertNotSame(steps.get(0).getMoves().get(0).getAngle(),
choreography1.getSteps().get(0).getMoves().get(0).getAngle());
        assertEquals(steps.get(0).getMoves().get(1).getAngle(),
choreography1.getSteps().get(0).getMoves().get(1).getAngle());
        assertEquals(steps.get(0).getMoves().get(2).getAngle(),
choreography1.getSteps().get(0).getMoves().get(2).getAngle());
        assertEquals(steps.get(0).getMoves().get(3).getAngle(),
choreography1.getSteps().get(0).getMoves().get(3).getAngle());
        assertEquals(steps.get(0).getMoves().get(4).getAngle(),
choreography1.getSteps().get(0).getMoves().get(4).getAngle());
        assertEquals(steps.get(0).getMoves().get(5).getAngle(),
choreography1.getSteps().get(0).getMoves().get(5).getAngle());
    }

    @Test
```

```
    public void addStep() {
        Step step1 = new Step();
        step1.addMove(new Servo(0,180));
        step1.addMove(new Servo(1,0));
        step1.addMove(new Servo(2,18));
        step1.addMove(new Servo(3,15));
        step1.addMove(new Servo(4,90));
        step1.addMove(new Servo(5,160));

        ArrayList<Step> steps = new ArrayList<>();
        Choreography choreography = new Choreography(steps);

        assertEquals(0, choreography.getSteps().size());

        choreography.addStep(step1, -1);

        assertEquals(step1, choreography.getSteps().get(0));
        assertEquals(1, choreography.getSteps().size());
        assertTrue(step1.equals(choreography.getSteps().get(0)));

    }
}
```

Possibility test

```
package Pc.Logic.Objects;

import org.junit.Test;

import static org.junit.Assert.*;

public class PossibilityTest {

    @Test
    public void getElement() {
        Possibility possibility = new Possibility();
        assertEquals(0, possibility.getElement(0));
        assertEquals(0, possibility.getElement(1));
        assertEquals(0, possibility.getElement(2));
        assertEquals(0, possibility.getElement(3));
    }

    @Test
    public void setElement() {
        Possibility possibility = new Possibility();
        assertEquals(0, possibility.getElement(0));
        possibility.setElement(0,5);
        assertEquals(5, possibility.getElement(0));
    }

    @Test
    public void getX() {
        Possibility possibility = new Possibility();
        assertEquals(0, possibility.getX(), 0.0);
    }

    @Test
    public void setX() {
        Possibility possibility = new Possibility();
        assertEquals(0, possibility.getX(), 0.0);
        possibility.setX(5);
        assertEquals(5, possibility.getX(), 0.0);
    }

    @Test
```

```
    public void getY() {
        Possibility possibility = new Possibility();
        assertEquals(0, possibility.getY(), 0.0);
    }

    @Test
    public void setY() {
        Possibility possibility = new Possibility();
        assertEquals(0, possibility.getY(), 0.0);
        possibility.setY(5);
        assertEquals(5, possibility.getY(), 0.0);
    }

    @Test
    public void getZ() {
        Possibility possibility = new Possibility();
        assertEquals(0, possibility.getZ(), 0.0);
    }

    @Test
    public void setZ() {
        Possibility possibility = new Possibility();
        assertEquals(0, possibility.getZ(), 0.0);
        possibility.setZ(5);
        assertEquals(5, possibility.getZ(), 0.0);
    }

    @Test
    public void isSingularite() {
        Possibility possibility = new Possibility();
        assertFalse(possibility.isSingularite());
    }

    @Test
    public void setSingularite() {
        Possibility possibility = new Possibility();
        assertFalse(possibility.isSingularite());
        possibility.setSingularite(true);
        assertTrue(possibility.isSingularite());
    }

    @Test
    public void getAngles() {
        Possibility possibility = new Possibility();
        assertEquals(0, possibility.getAngles()[0]);
        assertEquals(0, possibility.getAngles()[1]);
        assertEquals(0, possibility.getAngles()[2]);
        assertEquals(0, possibility.getAngles()[3]);
    }
}
```

Possibilities test

```
package Pc.Logic.Objects;

import org.junit.Test;

import java.util.ArrayList;

import static org.junit.Assert.*;

public class PossibilitiesTest {

    @Test
    public void getAllPossibilities() {
        ArrayList<Possibility> possibilities = new ArrayList<>();
        Possibility possibility = new Possibility();
        Possibilities possibilities1 = new Possibilities();
        assertEquals(possibilities, possibilities1.getAllPossibilities());
        possibilities.add(possibility);
        possibilities1.addPossibility(possibility);
        assertEquals(possibilities, possibilities1.getAllPossibilities());
    }

    @Test
    public void addPossibility() {
        Possibilities possibilities = new Possibilities();
        Possibility possibility = new Possibility();
        possibilities.addPossibility(possibility);
        assertTrue(possibilities.getAllPossibilities().size() == 1);

assertTrue(possibilities.getAllPossibilities().get(0).equals(possibility));
    }
}
```

14.1.2 Služby test

ForwardKinematics test

```
package Pc.Logic.Services.Calculations;

import org.junit.Test;

import static junit.framework.TestCase.assertEquals;
import static junit.framework.TestCase.assertTrue;
import static org.junit.Assert.*;

public class ForwardKinematicsTest {

    @Test
    public void calculateFK() {
        int[] angles =  new int[6];
        angles[0] = 0;
        angles[1] = 0;
        angles[2] = 0;
        angles[3] = 0;
        angles[4] = 0;
        angles[5] = 0;
        double[] expected = new double[3];
        expected[0] = 30.50;
        expected[1] = 0.00;
        expected[2] = 3.50;
```

```
        double[] coordinates = ForwardKinematics.calculateFK(angles);
        for (int i = 0; i < coordinates.length; i++) {
            assertEquals(expected[i], coordinates[i], 0.0);
        }

        angles[0] = 90;
        angles[1] = 0;
        angles[2] = 0;
        angles[3] = 0;
        angles[4] = 0;
        angles[5] = 0;

        expected[0] = 0.00;
        expected[1] = 30.5;
        expected[2] = 3.5;

        coordinates = ForwardKinematics.calculateFK(angles);
        for (int i = 0; i < coordinates.length; i++) {
            assertEquals(expected[i], coordinates[i], 0.01);
        }
    }
}
```

Communicator test

```
package Pc.Logic.Services.Communication;

import org.junit.Test;

import static org.junit.Assert.*;

public class CommunicatorTest {

    @Test
    public void findPorts() {
        Communicator communicator = new Communicator();
    }

    @Test
    public void openPort() {
    }

    @Test
    public void sendData() {
    }

    @Test
    public void sendData1() {
    }

    @Test
    public void sendData2() {
    }
}
```

AnglesTransformation test

```
package Pc.Logic.Services;

import org.junit.Test;

import static org.junit.Assert.*;

public class AnglesTransformationTest {

    @Test
    public void agles() {
        AnlesTransformation anlesTransformation = new AnlesTransformation();
    }
}
```

Controller

```
package Pc.Logic.Services;

import org.junit.Test;

import static org.junit.Assert.*;

public class ControllerTest {

    @Test
    public void chceckAngles() {
        Controller controller = new Controller();
        boolean error = true;
        int[] angles = new int[6];
        angles[0] = 0;
        angles[1] = 0;
        angles[2] = 0;
        angles[3] = 0;
        angles[4] = 0;
        angles[5] = 0;
        error = controller.chceckAngles(null, angles, error);
        assertEquals(false, error);
        angles[0] = 180;
        angles[1] = 180;
        angles[2] = 180;
        angles[3] = 180;
        angles[4] = 180;
        angles[5] = 180;
        error = controller.chceckAngles(null, angles, error);
        assertEquals(false, error);
        angles[0] = 180;
        angles[1] = 180;
        angles[2] = 180;
        angles[3] = 180;
        angles[4] = 180;
        angles[5] = 181;
        error = controller.chceckAngles(null, angles, error);
        assertEquals(true, error);
        angles[0] = 0;
        angles[1] = 0;
        angles[2] = 0;
        angles[3] = 0;
        angles[4] = 0;
        angles[5] = -1;
        error = controller.chceckAngles(null, angles, error);
        assertEquals(true, error);
    }

    @Test
    public void workSpaceControl() {
        Controller controller = new Controller();
        double[] coordinates = new double[3];
        controller.workSpaceControl(coordinates);
    }
}
```

Creator test

```
package Pc.Logic.Services;

import Pc.Logic.Objects.Possibility;
import Pc.Logic.Services.Calculations.ForwardKinematics;
import org.junit.Test;

import static org.junit.Assert.*;

public class CreatorTest {

    @Test
    public void createPossibility() {
        Creator creator = new Creator();
        Possibility possibilityCreated = creator.createPossibility(0,0,0,0);
        Possibility possibilityUser = new Possibility();
        int[] angles = {0,0,0,0};
        possibilityUser.setX(ForwardKinematics.calculateFK(angles)[0]);
        possibilityUser.setY(ForwardKinematics.calculateFK(angles)[1]);
        possibilityUser.setZ(ForwardKinematics.calculateFK(angles)[2]);
        assertTrue(possibilityUser.equals(possibilityCreated));

    }
}
```

FileWorker test

```
@Test
public void loadChoreography() {
    FileWorker fileWorker = new FileWorker();
    Choreography choreography =
fileWorker.loadChoreography("choreographyTest");

    Step step1 = new Step();
    step1.addMove(new Servo(0,180));
    step1.addMove(new Servo(1,0));
    step1.addMove(new Servo(2,18));
    step1.addMove(new Servo(3,15));
    step1.addMove(new Servo(4,90));
    step1.addMove(new Servo(5,160));

    Step step2 = new Step();
    step2.addMove(new Servo(0,0));
    step2.addMove(new Servo(1,15));
    step2.addMove(new Servo(2,180));
    step2.addMove(new Servo(3,30));
    step2.addMove(new Servo(4,90));
    step2.addMove(new Servo(5,170));

    Step step3 = new Step();
    step3.addMove(new Servo(0,42));
    step3.addMove(new Servo(1,42));
    step3.addMove(new Servo(2,42));
    step3.addMove(new Servo(3,42));
    step3.addMove(new Servo(4,42));
    step3.addMove(new Servo(5,43));

    ArrayList<Step> steps = new ArrayList<>();
    steps.add(step1);
    steps.add(step2);
    steps.add(step3);
```

```
    Choreography choreography1 = new Choreography(steps);

    assertTrue(choreography.getChoreography().size() ==
choreography1.getChoreography().size());

    for (int i = 0; i < choreography.getChoreography().size(); i++) {
        for (int j = 0; j <
choreography.getChoreography().get(i).getMoves().size(); j++) {

assertTrue(choreography.getChoreography().get(i).getMoves().get(j).getId() ==
choreography1.getChoreography().get(i).getMoves().get(j).getId());

assertTrue(choreography.getChoreography().get(i).getMoves().get(j).getAngle()
== choreography1.getChoreography().get(i).getMoves().get(j).getAngle());
        }
    }
}
```

14.2 Uživatelské testování

14.2.1 Grafická část

Grafickou část lze otestovat jen tak, že vyzkoušíme reakci jednotlivých grafických komponent. Pokud reakce souhlasí s očekávanou reakcí, není nikde problém.

14.2.2 Komunikace

Komunikační část byla testována vytvořením virtuálního sériového portu. Jednou stranou tohoto portu pak byla testovaná aplikace, druhou byl program Teraterm. Teraterm jen zobrazoval příchozí informace. Pokud se zobrazená informace shodovala s odeslanou bylo vše v pořádku. Jelikož nepřesnosti ohledně kódování informací byly vyřešeny během vývoje aplikace nebyly zaznamenány v průběhu testování potíže.