VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA STROJNÍHO INŽENÝRSTVÍ
FACULTY OF MECHANICAL ENGINEERING

ÚSTAV AUTOMATIZACE A INFORMATIKY
INSTITUTE OF AUTOMATION AND COMPUTER SCIENCE

MONITOROVÁNÍ PRACOVNÍHO PROSTORU ROBOTU

THE SAME IN ENGLISH WORKSPACE MONITORING OF ROBOT

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — Bc. Hana Gurecká
AUTHOR

VEDOUCÍ PRÁCE — Ing. et Ing. Stanislav Lang
SUPERVISOR

BRNO 2018

ZADÁNÍ VŠKP 1

(tento list nahradíte oficiálním zadáním práce)

ABSTRAKT

Cílem teoretické části práce je shrnutí postupů a algoritmů určených pro monitorování pracovního prostoru robota, kterým je věnována první kapitola. Dalším důležitým tématem je problematika přesnosti výpočtů při použití datových typů s plovoucí desetinnou čárkou. V poslední teoretické kapitole budou představeny možnosti reprezentace kuželu pomocí kvadratických ploch či analytické geometrie. Praktická část je věnována implementaci vlastního algoritmu pro monitorování pracovního prostoru kuželu a komolého kuželu v jazyce C++.

ABSTRACT

The goal of theoretical part of the thesis is to summarize basic procedures and algorithms for robotic workspace monitoring. The first chapter is focused on those procedures. Another important topic of thesis is problematics of floating point error in calculations accuracy. Representations of cones and conical surfaces are discussed in the last theoretical chapter. The practical part of the thesis is focused on implementation of cone AND truncated cone workspace monitoring algorithm in C++.

KLÍČOVÁ SLOVA

Monitorování pracovního prostoru, pracovní prostor tvaru kuželu, pracovní prostor tvaru komolého kuželu, chyba proměnných s plovoucí desetinnou čárkou.

KEYWORDS

Workspace monitoring, cone workspace, truncated cone workspace, float point error.

ÚSTAV AUTOMATIZACE
A INFORMATIKY

BIBLIOGRAFICKÁ CITACE

GURECKÁ,Hana. *Monitorování pracovního prostoru robotu,* Brno, 2018. Diplomová práce. Vysoké učení technické v Brně, Fakulta strojního inženýrství, Ústav automatizace a informatiky.

PODĚKOVÁNÍ

Děkuji svému vedoucímu Ing. et Ing. Stanislavu Langovi za odborné vedení a konzultace při sepisování práce. Dále děkuji konzultantovi z firmy B&R Mgr. Zbyňkovi Uhrovi za cenné rady v oblasti programování v Automation Studiu.

ČESTNÉ PROHLÁŠENÍ

Prohlašuji, že tato práce je mým původním dílem, zpracoval jsem ji samostatně pod vedením Ing. et Ing. Stanislava Langa a s použitím literatury uvedené v seznamu literatury.

V Brně dne 28. 5. 2018 ……

Hana Gurecká

OBSAH

1 ÚVOD

Práce se bude zabývat detekcí robota v pracovním prostoru kuželu a komolého kuželu. První část je teoretická, kdy se nejdříve seznámíme s monitorováním pracovního prostoru robotu v kontextu detekce kolizí. Dále si představíme problematiku chyb vzniklých použitím proměnných s plovoucí desetinnou čárkou. Třetí teoretická část je věnována kuželům a kuželovým plochám.

V první kapitole si vymezíme možnosti monitorování pracovního prostoru robotu, resp. detekci kolizí. Představíme si voxelové mřížky, binární stromy, oktalové stromy a v neposlední řadě také CSG - konstruktivní geometrii pevných těles. Dále si představíme princip monitorování pracovního prostoru firmou B&R Automation a jejich způsob detekce robota uvnitř vymezeného prostoru.

Druhá kapitola je věnována problematice přesnosti datového typu float, resp. všech datových typů pracujících s reálnými čísly. Nejdříve si vysvětlíme, jak jsou tato data reprezentována v paměti počítače a jak chyby spojené s datovými typy s desetinnou čárkou vznikají. Kapitola pokračuje způsoby měření chyb vzniklých použitím proměnných s plovoucí desetinnou čárkou. Jedná se například o absolutní nebo relativní chybu. Téma je zakončeno popisem vhodných způsobů porovnávání proměnných s ohledem na možnou vzniklou chybu.

Třetí kapitola bude věnována kuželům. Jednak si kužely představíme v kontextu kvadratických ploch, kdy je kuželová ploch určena jednou rovnicí, resp. maticí. U kvadrik si také naznačíme možnost výpočtu průsečíku přímky s kvadratickou plochou. Dále si představíme kužel pohledem analytické geometrie a způsob, jakým zjistíme průsečíky přímky s takovým objektem. Ke konci kapitoly budeme diskutovat, která z výše zmíněných struktur je vhodnější pro praktickou implementaci.

Bude následovat část práce věnovaná implementovanému programu, který hledá průsečíky přímky a kuželu, resp. přímky a komolého kuželu. V první části algoritmu bude zadán kužel (resp. komolý kužel), bude popsána funkce pro kontrolu vstupních dat a jejich transformaci. Dále bude v praktické části rozebrána cyklická část programu, kdy budou postupně načítána data o úsečkách (ramenech robota), kužel bude zmenšen a budou vypočteny průsečíky přímky s kuželem. Na základě těchto průsečíků pak budeme moci určit, zda daná část robotického ramena leží uvnitř zadaného kuželu.

Práce bude zakončena praktickou ukázkou z implementace algoritmu a diskuzí nad získanými výsledky.

2 MONITOROVÁNÍ PRACOVNÍHO PROSTORU ROBOTU

Pracovní prostor robota můžeme zjednodušeně definovat jako prostor, ve kterém se nachází všechny dílčí části robota v každém okamžiku jeho činnosti. Nejjednoduššími pracovními prostory jsou základní geometrické útvary, např. kvádr, koule, kužel atp. Sjednocením, případně rozdílem takovýchto útvarů lze dosáhnout komplexnějších pracovních prostorů. Vždy však musíme uvážit, že čím komplexnější útvar volíme, tím náročnější, a tedy pomalejší bude výpočet. Pracovní prostor, stejně jako robot (nebo jeho část) jsou trojrozměrné objekty, můžeme je tedy s výhodou popisovat stejnou metodou. V první kapitole si představíme některé z možností vymezení pracovních prostorů nebo objektů (robotů) a na závěr uvedeme metodu firmy B&R Automation, ze které bude následně vycházet algoritmus pro detekci robota v pracovní prostoru tvaru kužele.

2.1 Vymezení pracovního prostoru a objektu

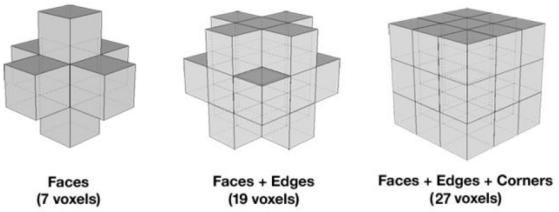
Podmnožinu trojrozměrného euklidovského prostoru, kterou nazveme pracovním prostorem, můžeme definovat nekonečně mnoha způsoby. Jelikož však máme ve vymezeném prostoru za úkol detekovat objekt (robota), je vhodné zvolit definici v kontextu detekcí kolizí. Proto si v této podkapitole představíme základní metody aproximace objektů a prostorů, které jsou používány nejen v robotice, ale i 3D grafice, počítačových hrách, medicíně atp.

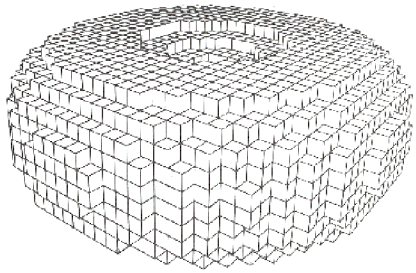
2.1.1 Mřížky

Nejjednodušší reprezentací prostoru, nebo objektu v něm, jsou mřížky v souřadném systému. Mřížka může být uniformní, tedy tvořena krychlemi, nebo neuniformní, tvořená kvádry. V trojrozměrném prostoru buňky utvořené mřížkou nazýváme *voxely*(volume element). Pomocí voxelů nereprezentujeme objekt jako celek, ale určujeme, zda jednotlivé buňky v prostoru jsou obsazené nebo neobsazené (Obr. 1).

Metoda je principiálně skutečně velmi jednoduchá. Pokud chceme vědět, jestli určitý bod leží uvnitř tělesa, stačí se zeptat na obsazenost voxelu, který přísluší souřadnicím bodu. Nicméně existují i úskalí reprezentace těles pomocí mřížek v souřadném systému, které spočívá v tom, že opravdu přesně pomocí ní lze reprezentovat pouze takové objekty, které mají všechny své hrany rovnoběžné s osami souřadného systému. V opačném případě vznikají pouze kvalitnější, či méně kvalitní aproximace zadaných těles (Obr. 2). Samozřejmě lze namítnout, že s dostatečným zjemněním mřížky můžeme dosáhnout podle potřeby dostatečně dobré aproximace. Dále každý jednotlivý voxel musí mít svou vlastní informaci o hodnotě. Pokud tedy zjemníme mřížku příliš, stane se objekt paměťově náročným. Mějme například mřížku

100x100x100 (která je stále velmi nepřesnou) a tedy milion krychlí, o kterých potřebujeme uchovávat informaci typu bool (0,125Mb). I přes své nevýhody však voxely i v dnešní době nachází uplatnění např. v medicíně (CT, MRI)[1].

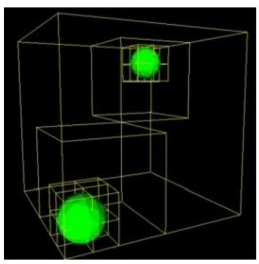


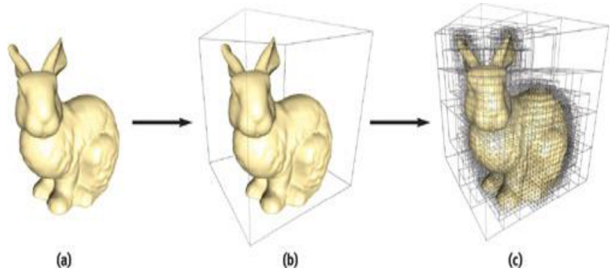


Obr. 1: jednoduché objekty vymezené mřížkou[2]
Obr. 2: komplexní objekt reprezentovaný mřížkou [3]

2.1.2 Oktalový strom

Prostorově výhodnější, než mřížková reprezentace, jsou *oktalové stromy* (*octree*). V souvislosti s octree je také velmi často zmiňován *kvadrantový strom* (*quadtree*), který je dvojdimenzionální obdobou octree pracující ovšem na stejném principu, a proto je snazší pro počáteční představu. Jak již název napovídá, jedná se o grafové struktury, kdy z každého uzlu vedou čtyři hrany (v druhém případě osm hran). Zlepšení spočívá v adaptivním přizpůsobení velikosti jednotlivých voxelů povrchu součásti. Velké plné nebo prázdné oblasti reprezentujeme voxely velkých rozměrů (tedy šetříme mnoho paměti), zatímco na povrchu objektu rekurzivně dělíme voxely tak, abychom dosáhli požadované přesnosti (Obr. 3 a 4).





Obr. 3: Octree - malé rozlišení[4]
Obr. 4: Octree velké rozlišení[5]

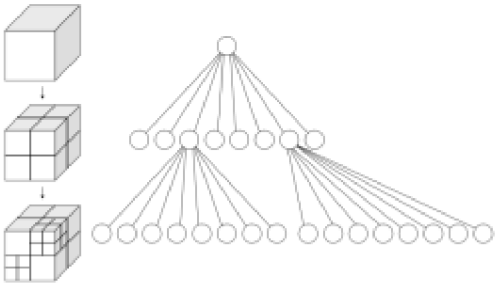
Jak je výše zmíněno, princip spočívá v adaptivním dělení prostoru. Při utváření stromové struktury, ať už octree nebo quadtree, začínáme tím, že máme požadovaný tvar (těleso/prostor) uvnitř krychle/čtverce, které reprezentují okolní prostor. Následující

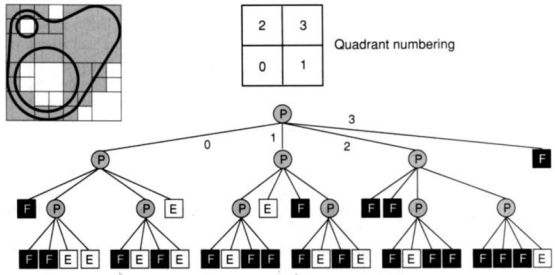
[1] FUNKHOUSER, Thomas. Solid Modeling.
[2] Dostupné z: https://fcp-indi.github.io/docs/user/reho.html
[3] Dostupné z: https://pdfs.semanticscholar.org/presentation/32cc/6e29963ed0d0d13e9f633398f267d91856fb.pdf
[4] Dostupné z: http://nehe.ceske-hry.cz/cl_gl_octree.php
[5] Dostupné z: https://developer.nvidia.com/gpugems/GPUGems2/gpugems2_chapter37.html

popis algoritmu bude uveden pro trojrozměrný prostor, nicméně pro quadtree je princip stejný[6] (Obr. 5 a 6).

1. Krychli rozdělíme na menší dílky tak, že ji v polovině každého rozměru (x,y,z) rozřízneme rovinou. Tak nám vznikne 8 menších vzájemně disjunktních krychlí, které se dotýkají v jednom bodě (těžišti původní krychle). Tento bod se stává uzlem grafu, zatímco jednotlivé krychle jsou listy grafu.
2. V dalším kroku probíhá kontrola obsazenosti každé, z nově vzniklých krychlí. Možné stavy jsou: plně obsazena, zcela prázdná nebo částečně obsazena.
3. V případech, kdy je krychle zcela prázdná/zcela vyplněná objektem je algoritmus pro tuto větev ukončen, resp. krychle představuje list grafu. V případě částečně obsazených krychlí zkontrolujeme koncovou podmínku, kterou je v tomto případě dosažení maximálního rozlišení (maximální hloubka grafu). Pokud maximálního rozlišení dosaženo nebylo, je pro částečně obsazené kvádry celý proces opakován.

Zdůrazněme, že i po konci algoritmu jsou některé listy grafu částečně zaplněny. Naším požadavkem však je, aby se o každém listu stromu dalo jednoznačně určit, zda je zaplněn, či nikoliv. Zde závisí na aplikaci, ke které oktanový strom využíváme. Buď můžeme určit míru zaplnění a na základě té rozhodnout, zda pole označíme jako obsazeno/neobsazeno. V případě detekce kolizí - tedy v našem případě - je výhodnější brát každou částečně vyplněnou oblast jako zcela obsazenou. Je to z toho důvodu, že upřednostňujeme vznik falešných kolizí před nedetekovanými kolizemi.





Obr. 5: Octree grafová struktura[7]

Obr. 6: Qardtree reprezentace (E-prázdné pole, F-plné, P-částečně zaplněné)[8]

2.1.3 Binární rozdělení prostoru - BSP trees

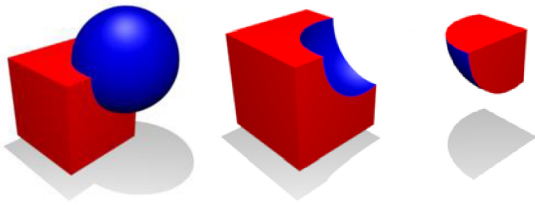
Další zajímavou stromovou strukturou jsou binární stromy. Binární stromy fungují rekurzivně a vymezují prostor pomocí orientovaných polorovin. Jsou tedy vhodné pro definice objektů nebo prostorů tvaru mnohostěnu.

2.1.4 Konstruktivní geometrie pevných těles

Jedním ze způsobů vymezení prostoru je tzv. *konstruktivní geometrie pevných těles*, známější pod anglickým názvem *Constructive solid geometry* (CSG). Vymezení

[6] NEVALA, Eric. Introduction to Octrees
[7] Dostupné z: https://en.wikipedia.org/wiki/Octree
[8] Dostupné z: https://www.slideshare.net/KRvEsL/solid-modeling

prostoru pomocí CSG spočívá ve skládání jednoduchých objektů tzv. primitivů (*primitives*) do složitějších útvarů za pomoci regularizovaných booleovských operací[9]: sjednocení, rozdílu a průniku. (Regulární těleso je takové, které je rovno uzávěru svého vnitřku). Ačkoliv výchozí objekty jsou zpravidla velmi jednoduše definované tak, aby se rychle dalo určit, zda je nějaký bod uvnitř, nebo vně tělesa (prostoru), vhodným skládáním můžeme ve výsledku vymodelovat velmi komplexní těleso (Obr. 7 a 8).



Obr. 7: sjednocení, rozdíl, průnik[10]

Obr. 8: komplexní těleso[11]

Základními tělesy pro metodu jsou obvykle: koule, kvádr, válec, kužel, torus nebo hranol. Jsou používána pro svou jednoduchou reprezentaci (například kouli můžeme reprezentovat jejím středem a poloměrem, kvádr jako dva body na tělesové úhlopříčce,..). Jedním ze způsobů určení, zda nějaký bod tělesu náleží, či nikoliv, je sestavení implicitní funkce ze zadaných informací[12]:

$$F(x,y,z) = 0.$$

Pak můžeme pro jednotlivé body říci, že:

a) leží uvnitř tělesa: $F(x,y,z) < 0 \quad \Rightarrow -1$

b) leží na povrchu tělesa: $F(x,y,z) = 0 \quad \Rightarrow 0$

c) leží mimo těleso: $F(x,y,z) > 0. \quad \Rightarrow 1$

Pro zjednodušení zavedeme $F(x,y,z) \in \{-1,0,1\}$. Takto můžeme specifikovat polohu bodu vůči tělesu, i když zvolíme jinou metodu klasifikace, než je právě implicitní funkce (u koule například postačí vzdálenost zadaného bodu od středu), a tedy pro zadaný bod p a dané booleovské operace nad tělesy A a B platí:

- $F_{A\cap B}(p) = \max(F(p)_A, F(p)_B)$
- $F_{A\cup B}(p) = \min(F(p)_A, F(p)_B)$
- $F_{A-B}(p) = \max(F(p)_A, -F(p)_B)$

Při provedení jakékoliv booleovské operace nad dvěma základními tělesy musí vzniknout buď nový 3D útvar, nebo prázdná množina. Ačkoliv se ve spojitosti s CSG mluví velmi často pouze o kombinacích trojrozměrných těles, některé zdroje uvádí, že jedním ze základních útvarů (nebo i jediným základním útvarem) mohou být roviny (resp. poloroviny)[13] [14], kdy je jasně určeno, která polorovina definuje „vnitřek“ a která „vnějšek“ roviny. Následně je pak možné výše zmíněné základní útvary definovat pouze

[9] STRACHOTA, Pavel. *Modelování těles*
[10] STRACHOTA, Pavel. *Modelování těles*
[11] Dostupné z: https://renderman.pixar.com/resources/RenderMan_20/constructiveSolidGeometry.html
[12] DOUG, James. Spatial Data Structures.
[13] Constructive Solid Geometry CSG. In: *Netgen/NGSolve*
[14] HEGDE, Shriram. *Solid Modeling Techniques: Constructive Solid Geometry (CSG)*

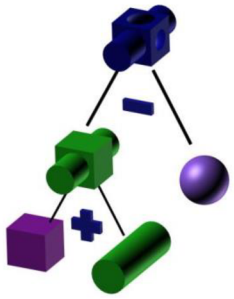
pomocí těchto polorovin. Jako příklad si můžeme uvést kvádr, který lze snadno definovat jako průnik šesti polorovin. Výhradně rovinná reprezentace je však poměrně nevýhodná, jelikož jinak zcela základní útvary, jako je válec nebo koule musíme aproximovat velkým množstvím polorovin. Proto se nejčastěji roviny využívají jako doplněk k objemovým základním tělesům, kdy potřebujeme například část tělesa „odříznout“, k čemuž bychom v opačném případě museli použít rozdíl s dalším tělesem.

Jak jsme si již zmínili výše, základní tělesa skládáme pomocí booleovských operátorů. Nesmíme opomenout, že tyto operátory nejsou obecně komutativní, což znamená, že záleží na pořadí, v jakém funkce na jednotlivá tělesa aplikujeme. Mějme například kouli K, čtverec C a jehlan J, které se vzájemně protínají v prostoru. Dále nad nimi mějme operace sjednocení a rozdíl, které aplikujeme v různém pořadí, například takto:

a) $(K \cup J) \cap C$

b) $K \cup (J \cap C)$

Je zřejmé, že výsledky se pro dané uspořádání těles liší. Proto tělesa uspořádáme do binárního stromu, jehož listy jsou základní tělesa a hrany představují operace mezi tělesy (sjednocení, průnik nebo rozdíl) (Obr. 9). V některých aplikacích můžeme do hran ukládat i unární translaci nebo rotaci tělesa (do následníka rotovaného/posouvaného tělesa pak zřejmě vede pouze jedna hrana)[15]. V uzlech stromu jsou nově vzniklá tělesa a kořenem stromu je výsledné těleso. Takto získáme jednoznačně definovaný prostor/objekt. V procesu pak postupujeme od listů ke kořenu stromu (depth first tree walk).



Obr. 9: Hierarchie operací[16]

2.2 Vymezení pracovního prostoru a detekce kolizí v pojetí B&R Automation

Pracovní prostor je vymezen šesti typy oblastí. Jednak je to Workspace, tedy jednoduchý základní prostor, ve kterém následně můžeme vymezit oblasti typu Safespace, Safe half-space,Work half-space, Exclusive workspace nebo Protected

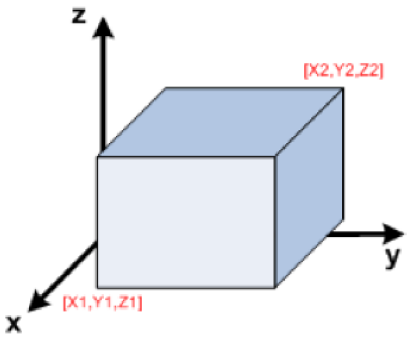
[15] STRACHOTA, Pavel. *Modelování těles*

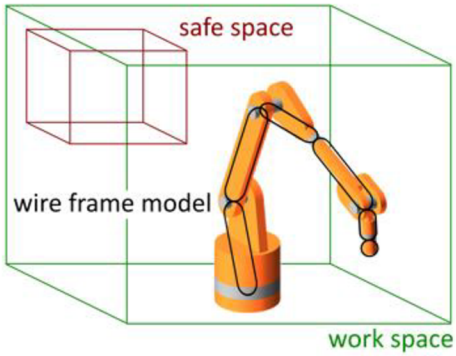
[16] Dostupné z: http://groups.csail.mit.edu/graphics/classes/6.837/F98/talecture/

workspace, které robotu pohyb v dané oblasti dle potřeby povolují, zakazují nebo omezují.

Prvním základním pojmem, který si definujeme je *Workspace*. V našem případě se jedná o kvádr vytnutý v prostoru (product coordinate system (PCS)). Workspace je prostor, který robot nesmí žádnou svou částí opustit po celou dobu svého běhu. Kvádr je zcela určen dvěma body v prostoru {[X1,Y1,Z1],[X2,Y2,Z2]}. Jedná se o body na diagonále kvádru, tedy levý dolní bod přední stěny a pravý horní bod zadní stěny, (Obr. 10). Dalším parametrem, prostoru typu workspace, je identifikátor, který používáme, jelikož každému robotu přísluší právě jeden workspace, nicméně, jak si vysvětlíme dále, hlavní program často kontroluje více robotů, jejichž pracovní prostory se často překrývají.

Safespace je definován obdobně jako workspace. Opět se jedná o kvádr definovaný dvěma body na úhlopříčce a identifikátorem. Zásadní rozdíl je však v použití. Zatímco je pro každého robota definován právě jeden Workspace, jakožto prostor, ve kterém se vyskytovat může, Safespace se může vyskytovat v libovolném počtu a vytíná prostor, do kterého robot nesmí vstoupit. Logicky tedy vyplývá, že safespace má smysl definovat pouze uvnitř workspace. Sjednocením workspace a safespace můžeme dostat již složitější (nekonvexní) pracovní oblast robota (Obr.11).

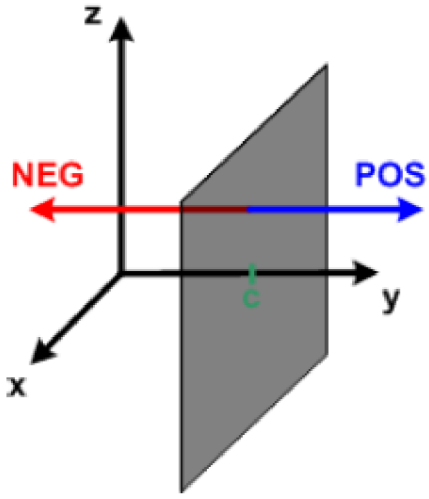




Obr. 10: Definice workspace a safespace pomocí bodů na úhlopříčce

Obr. 11: Kombinace workspace a safespace

Dalšími dvěma pojmy jsou *Work half-space* a *Safehalf-space*. Nyní, jak názvy napovídají, nebudeme definovat povolený/ zakázaný prostor pomocí celých kvádrů, ale pouze polorovin. Work half-space představuje poloprostor, ve které musí ležet všechny části robota. Naopak safe half-space je poloprostor do kterého nesmí ani jedna část robota zasáhnout. Poloprostory musí být vždy rovnoběžné s některou ze základních rovin (XY, XZ, YZ). Potom jsou jednoznačně určeny identifikátorem roviny, se kterou jsou rovnoběžné, souřadnicemi, určující polohu poloroviny, a orientací (POS,NEG) viz (Obr. 12).



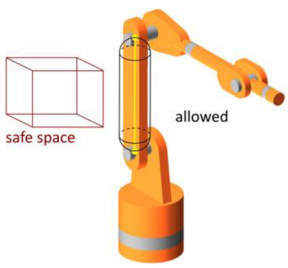
Obr. 12: Definice safe half-space/ work half-space

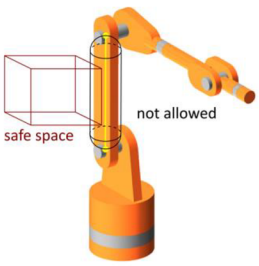
V předchozích odstavcích jsme se zabývali pouze případy, kdy robot do dané oblasti jednoznačně smí/nesmí zasáhnout. Za speciální případ můžeme považovat tzv. *Exclusive workspace*, který určuje omezený přístup do oblasti. Jedná se o vymezení prostoru, ve kterém se smí v daném čase nacházet právě jeden robot. Pokud nějaká část jednoho robota zasáhne do takto definovaného prostoru, oblast se uzamkne. V případě, že do uzamčené oblasti bude plánován pohyb jiného robota, tento robot bude zastaven, dokud první robot oblast neopustí a oblast nebude znova odemčena. Všimněme si, že na rozdíl od předchozích pojmů se exclusive workspace vztahuje k použití více robotů s překrytými pracovními prostory.

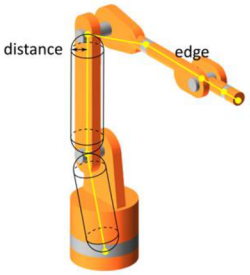
2.3 Detekce robota uvnitř pracovního prostoru

V předchozí podkapitole jsme si představili, jak vymezit prostor, ve kterém se robot může pohybovat. Nyní přejdeme k samotným principům monitorování robotů. Již dříve jsme uvedli, že cílem monitorování pracovního prostoru je zamezení jakékoliv části robota v opuštění povoleného prostoru. Výpočty pro každý jednotlivý bod stroje by však byly velmi náročné. Proto se zavádí zjednodušení, které umožňuje robotům pracovat v reálném čase.

Výpočet zjednodušíme pomyslným rozložením robotického ramene na jednotlivé díly. Každý díl následně obalíme válcem, kde výška odpovídá délce dílu a průměr odpovídá nejširšímu místu součásti. Každý díl tedy nyní máme definován jako úsečku (dva body) a poloměr. Nově vzniklá úloha spočívá v zjištění, zda jednotlivé válce leží uvnitř povoleného pracovního prostoru. Další zjednodušení spočívá v redukci daných válců na úsečky tak, že povolený pracovní prostor zmenšíme o poloměr válce, který původně aproximoval danou část robota. Tím jsme získali jednoduchou úlohu, kdy pro každou úsečku (část robota) zjišťujeme, zda leží v definovaném pracovním prostoru zmenšeném o r - poloměr součástky viz (Obr 13, Obr. 14).







Obr. 13: Náhrada robota válci

Obr. 14: Náhrada úsečkami

Detekce skutečnosti, zda se celý robot nachází v pracovní prostoru, je nyní již velmi jednoduchá úloha. Připomeňme si, že pracovní prostor máme zadán pomocí (zmenšených) kvádrů a polorovin (s posunutými souřadnicemi). Pro kvádry, jakožto konvexní tělesa platí, že leží-li uvnitř oba krajní body úsečky (definující součást robota), leží uvnitř i celá úsečka. Stačí tedy jednoduchým porovnáváním zjistit, zda každá souřadnice (X, Y, Z) bodů úseček leží mezi body (souřadnicemi) povoleného pracovního prostoru a zároveň mimo souřadnice safespace. Pro poloroviny je úloha ještě jednodušší, jelikož stačí porovnat pouze jedinou souřadnici bodu proti pozici, kterou vytíná polorovina na ose, na kterou je kolmá.

Vidíme, že zjednodušením jsme získali velmi efektivní metou založenou na jednoduchém porovnávání souřadnic.

3 PROBLEMATIKA PŘESNOSTI - FLOATING POINT ERROR

Další nemalá část práce bude věnována problematice přesnosti výpočtů, která v oblasti robotiky hraje významnou roli. Roboty mají celou řadu využití od nejrůznějších montáží, přes obrábění až po přístroje na operačních sálech, kde je chyba ve výpočtu takřka netolerovatelná. Uvědomme si, že ti nejpřesnější roboti dnes již dosahují přesnosti až 0,1 nanometru[17]. Ovšem již v rámci výpočtové části programu mohou vzniknout menší či velké nepřesnosti, jejichž příčinou je především způsob ukládání hodnot proměnných do paměti počítače a další práce s nimi. Proto si v následující kapitole představíme tzv. *aritmetiku s plovoucí desetinnou čárkou* (*floating-point arithmetic*) tedy jak a proč chyby vznikají, jaké v souvislosti s nimi vznikají problémy a jak je možné je řešit.

3.1 Reprezentace reálných čísel v počítači

Jak již bylo výše zmíněno, problém s nepřesnostmi je především otázka datového typu float (double, extended,...), tedy reálného čísla s pohyblivou desetinnou čárkou, a to především kvůli jeho reprezentaci v paměti počítače. Pro začátek je vhodné zmínit, že tuto reprezentaci upravuje standard IEEE 754 (resp. IEEE 854).

Reprezentovat jediné reálné číslo lze celou řadou způsobů. Nejintuitivnější zápis je pro většinu lidí pravděpodobně v desítkové soustavě. Nicméně i zde existují různé varianty zápisu. Například 0.5, $\frac{1}{2}$, $\frac{5}{10}$, $5.0 * 10^{-1}$, $0.005 * 10^{2}$ nebo $1.0 * 2^{-1}$ představují zápisy jediné hodnoty. Hodnoty uložené v paměti počítače jsou velmi podobné tzv. vědecké notaci.

Výhodou vědecké notace je, že může reprezentovat velmi malá i obrovská reálná čísla, která by v desítkovém zápisu zabrala zbytečně mnoho místa, velmi jednoduše pomocí součinu dvou čísel. Další výhodou je pak významné zjednodušení některých aritmetických operací. Prvním číslem, ze kterého je složen zápis vědecké notace je tzv. mantisa (nebo též signifikantu), která představuje tzv. platné číslice reálného čísla. Za platné číslice považujeme takovou posloupnost číslic, které v desítkovém zápisu nepředchází nic, případně samé nuly a následují taktéž už jen nuly. Velikost mantisy roste s přesností reálného čísla. Druhé číslo pak představuje exponent čísla 10, kladný nebo záporný podle toho, jestli chceme posouvat desetinnou čárku doprava nebo doleva.

[17]Dostupné z: http://www.fanuc.eu/cz/cs/robonano

Exponent je volen tak, abychom při umístění desetinné čárky za první platnou číslici a vynásobení mocninou desíti dostali původní reálné číslo.[18][19]

Původní reálné číslo	Vědecká notace
0.000000000000580231	5.80231×10^{-13}
3652000000000000000000	3.652×10^{21}

Nicméně jak je známo, počítače nepracují v desítkové, nýbrž v dvojkové soustavě. A zde se i nachází rozdíl mezi vědeckou notací a reprezentací čísla v paměti počítače. Zatímco v prvním případě mocníme bázi rovnu 10, čísla typu float v počítači mají bázi 2. Dalším rozdílem je, že jednotlivé datové typy v počítači jsou standardizovány, mají v paměti vymezen určitý počet bitů (klasický float 32 bitů). Signifikand i exponent bývají zpravidla ukládány jako celá čísla (integer se znaménkem). Výsledné číslo pak dostaneme následovně:

$$(znaménko) * signifikand * 2^{\pm exponent}$$

Otázkou zůstává, jak místo rozdělit. Chceme-li získat reálné číslo s co nejvyšší přesností, pak větší prostor dostává signifikand. V opačném případě, kdy požadujeme větší rozsah čísla, má větší prostor exponenciální část čísla, ovšem na úkor přesnosti[20]. Standard IEEE 754 rozlišuje 4 formáty typu float, pro které jsou jasně stanoveny rozmezí exponentů, signifikandů viz tab. 1. Po revizi standardu v roce 2008 pak přibyl ještě Quadruple-precision formát, který poskytuje ještě větší míru přesnosti.[21]

Parameter	Format			
	Single	Single-Extended	Double	Double-Extended
p	24	≥ 32	53	≥ 64
e_{max}	+127	≥ 1023	+1023	> 16383
e_{min}	-126	≤ -1022	-1022	≤ -16382
Exponent width in bits	8	≤ 11	11	≥ 15
Format width in bits	32	≥ 43	64	≥ 79

Tab. 1: přehled formátů typu float[22]

b_0	$b_1\ b_2\ b_3$ ..... b_8	$b_9\ b_{10}\ b_{11}$.....$b_{30}\ b_{31}$
Sign	Exponent	Significand

Obr. 15: Reprezentace single float hodnoty v paměti počítače

V tabulce můžeme pozorovat, že zatímco klasický single a double float formát má přesně stanovený počet bitů, tedy 32 a 64, pro extended verze jsou určeny pouze dolní hranicí a mohou být nastaveny dle potřeby.

[18]Scientific Notation. In: *Department of Chemistry Texas A&M University*
[19]Significant Figures. In: *Department of Chemistry Texas A&M University*
[20]NULL, Linda. a Julia. LOBUR. *The essentials of computer organization and architecture*
[21]KEARFOTT, Ralph Baker. *The IEEE 754-2008 Floating Point Standard and its Pending Revision*
[22]GOLDBERG, David. *What Every Computer Scientist Should Know About Floating-Point Arithmetic*

3.2 Vznik chyby při počítání s reálnými čísly

Nyní, když jsme si představili reprezentaci čísel s plovoucí desetinnou čárkou v paměti počítače, pojďme se podívat, jakým způsobem vznikají chyby při práci s nimi. V předchozí podkapitole jsme si příklady uváděli pouze na konečných reálných číslech. Paměť v počítači však není neomezená a tudíž není možné zcela přesně reprezentovat libovolné reálné číslo. Příkladem v desítkové soustavě jsou racionální nebo iracionální čísla. Má-li číslo neukončený desetinný rozvoj v jedné soustavě (bázi), nemusí to nutně znamenat, že tomu tak je i v bázi jiné. Uveďme si třeba příklad čísla $\frac{1}{3}$, které vyčísleno v desítkové soustavě odpovídá $0.333\overline{3}$, tedy má nekonečný desetinný rozvoj. Avšak chceme-li jej převést do báze 3, pak dostaneme $1 * 3^{-1}$, což je reprezentace zcela přesná. Jak také bylo výše zmíněno, počítače interně pracují v binární soustavě, kde taktéž platí, že některá čísla můžeme reprezentovat zcela přesně, zatímco jiná aproximujeme, abychom získali výsledek alespoň přibližný. Obecně lze říci, že přesně můžeme reprezentovat takové zlomky, jejichž jmenovatel má společného prvočinitele s danou bází[23]. Kdykoliv nastane situace, že číslo není možné v dané bázi vyjádřit přesně, přichází na řadu aproximace v podobě zaokrouhlování.

Standard IEEE 754 rozlišuje dokonce 4 různé způsoby zaokrouhlování: zaokrouhlení k nejbližšímu číslu, k $+\infty$, k $-\infty$ a k 0. Zaokrouhlení k 0 znamená k číslu blíž nule, v důsledku se tedy jedná o odříznutí části za poslední platnou číslicí, což je sice nejjednodušší způsob zaokrouhlení, nicméně způsobuje relativně největší chybu[24]. Zaokrouhleni k $+\infty$ znamená, že kdykoliv bude za poslední platnou číslicí číslice jiná než nula, bude se zaokrouhlovat nahoru, resp. poslední platná číslice se zvýší resp. sníží podle toho, zda je číslo kladné resp. záporné. Zaokrouhlení k $-\infty$ je opakem zaokrouhlení k $+\infty$, tedy v případě, že je číslo větší než 0, pak se část za poslední platnou číslicí maže, zatímco je-li číslo celkově záporné, pak je poslední platná číslice zvýšena. Posledním, avšak nejčastěji používaným typem zaokrouhlování je zaokrouhlení k nejbližšímu číslu, což odpovídá klasickému zaokrouhlování vyučovanému na základních školách, kdy jsou číslice 5 a výše v desítkové soustavě, resp. polovina báze a vyšší čísla obecně, zaokrouhleny na číslo vyšší, zatímco nižší číslice jsou odříznuty a zůstává číslo nižší[25].

Abychom mohli s reálnými čísly dále pracovat, je vhodné velikost možné chyby, tedy míry, jakou se od sebe liší skutečné reálné číslo a jeho obdoba s plovoucí desetinnou čárkou, alespoň odhadnout. Můžeme rozlišovat chybu absolutní a relativní. Absolutní chybou je rozdíl mezi skutečným číslem a jeho aproximovanou hodnotou:

$$E_{abs} = |původníreálnéčíslo - aproximacefloatem|$$

Absolutní chyba může v závislosti na druhu zaokrouhlování dosáhnout velikosti až celého rozměru tzv. jednotky na posledním místě, v literatuře často zkracováno

[23] *Rounding Errors* Dostupné z: http://floating-point-gui.de/errors/rounding/
[24] *Rounding Errors* Dostupné z: http://floating-point-gui.de/errors/rounding/
[25] An introduction to different rounding algorithms. In: *EETimes* Dostupné z: https://www.eetimes.com/document.asp?doc_id=1274485

na *ulps* (units in the last place). Při klasickém zaokrouhlení k nejbližšímu číslu je zřejmé, že chyba může dosahovat 0 až 0.5 ulps v závislosti na číslicích, které chceme zaokrouhlit. Vezměme si jako příklad 1.22356 zaokrouhleno a uloženo jako $1.22 * 10^0$, pak absolutní chyba odpovídá $E_{abs} = 0.356ulps$. Někdy je však vhodnější volit relativní způsob měření chyb. Tento způsob, spíše než absolutní hodnotu rozdílu, určuje, jak moc významnou část čísla jsme zaokrouhlováním eliminovali. Relativní chybu vypočteme tak, že absolutní chybu podělíme reálným číslem, tedy:

$$E_{rel} = |\frac{původníreálnéčíslo - aproximacefloatem}{původníreálnéčíslo}|$$

Chceme-li tedy spočítat relativní chybu čísla 1.22356 zaokrouhleného na $1.22 * 10^0$, pak dostáváme $E_{rel} \approx 0.003$. Zaokrouhlujeme-li k nejbližšímu číslu, pak se dá o relativní chybě dokázat, že je v závislosti na své bázi ohraničena dvěma hodnotami následovně[26]:

$$\frac{1}{2}\beta^{-p} \leq \frac{1}{2}ulp \leq \frac{\beta}{2}\beta^{-p}$$

kde β je báze a p je přesnost. Vidíme tedy, že relativní chyba nepřekročí hodnotu $\varepsilon = \frac{\beta}{2}\beta^{-p}$, známou jako *strojové epsilon* (machine epsilon).

3.3 Porovnávání čísel s plovoucí desetinnou čárkou

V předchozí části jsme si vysvětlili, jak vznikají chyby v počítání s čísly s plovoucí desetinnou čárkou. Samotné zaokrouhlení však ve většině případů ještě nepředstavuje zásadní problém, jelikož přesnost si volíme dle potřeb programu. Hlavním úskalím zaokrouhlování bývá, pokud chceme výsledné hodnoty (výpočtu, algoritmu,...) porovnávat s pevně zadaným číslem nebo mezi sebou. Mějme například reálná čísla $a = 0.15 + 0.15$ a $b = 0.1 + 0.2$, pak vlivem převodu do binární soustavy a zaokrouhlovacích chyb se a nemusí rovnat b. Jak tedy vhodně porovnat dvě čísla?

Nejjednodušším a nejintuitivnějším způsobem jak lze porovnávat dvě hodnoty s ohledem na možnost vzniku zaokrouhlovacích chyb je stanovení epsilon jako pevně zvolené konstanty. Dále je vypočtena absolutní chyba E_{absc}, kdy ovšem neodečítáme původní reálné číslo od jeho aproximace, nýbrž dvě čísla (a a b), která chceme srovnat.

$$E_{absc} = |a - b|$$

Takto získaná odchylka je s pevně zvolenou konstantou porovnána. Pokud chyba menší nebo rovna epsilon, pak jsou čísla považována za rovná. Tento způsob však zdaleka není univerzální. Mějme například porovnání čísel 24.0000001 a 24.0 a dále 0.0000011 a 0.0000012. Je patrné, že v obou případech je $E_{abs} = 0.0000001$, avšak zatímco v prvním případě bychom čísla mohli považovat za rovná, v případě druhém se již může jednat o významnou odchylku. Z tohoto důvodu je samostatné použití

[26]GOLDBERG, David. *What Every Computer Scientist Should Know About Floating-Point Arithmetic*

absolutní chyby pro určení blízkosti dvou čísel značně limitováno na případy, kdy očekáváme porovnávané hodnoty z užšího rozsahu floatových hodnot. Pak můžeme po úvaze zvolit epsilon vyhovující celému rozsahu očekávaných hodnot.[27]

Představme si nyní další způsoby porovnání čísel s plovoucí desetinnou čárkou. V předcházejících částech jsme si kromě absolutní odchylky představili i odchylku relativní E_{rel}, která je další z možností, jak porovnávat floatové hodnoty a zároveň bere ohled na velikost chyby vůči velikosti čísla. Opět si vzorec pro výpočet upravíme pro potřeby porovnávání:

$$E_{relc} = |\frac{a-b}{\max(|a|,|b|)}|$$[28]

I v tomto případě pevně volíme hodnotu epsilon, která ovšem na rozdíl od porovnávání pomocí absolutní chyby, představuje jak velký podíl chyby oproti větší z porovnávaných hodnot dovolíme (v případě vynásobením 100 dostáváme hodnotu v procentech).

Třetím způsobem, jak porovnat dvě desetinná čísla je jejich převedení na celočíselnou reprezentaci a následné porovnání. Toto je možné díky tomu, že jestliže jsou si dvě čísla blízká ve floatové reprezentaci, pak jsou si blízká i v reprezentaci integerové. Odtud pak na základě „Dawson's obvious-in-hindsight theorem", který říká: „rozdíl celočíselných reprezentací desetinných čísel se stejným znaménkem odpovídá 1+počet reprezentovatelných floatových hodnot mezi nimi".[29] Tedy pokud se tento rozdíl rovná 1, pak jsou si dvě porovnávaná čísla nejblíže, jak to jde v rámci floatové reprezentace, aniž by si byla rovná. Při porovnávání tedy opět můžeme stanovit, kolik jednotek na posledním místě je ještě akceptovatelných pro to, abychom o dvou číslech řekli, že jsou si rovna. Prakticky je však porovnávání na základě integerové reprezentace obdobou porovnávání na základě relativní chyby.

Při implementaci porovnávání - ať již pomocí relativní chyby, nebo pomocí reprezentace integery - je důležité myslet na extrémní případy, kdy se výše zmíněné postupy nemusí chovat očekávatelně. Typickým příkladem je okolí nuly, resp. nula samotná. V případě porovnávání pomocí relativní chyby je zřejmé, že pokud zvolíme k porovnání dvě čísla blízká nule, pak budeme jedním z těchto čísel dělit, čímž může výsledná relativní chyba značně narůst, případně pokud porovnáváme záporné číslo oproti nule, pak může nastat případ dělení nulou. Obdobně porovnáváme-li pomocí integerové reprezentace, pak i opravdu malé čísla ($2.2 * 10^{-10}$) mohou být i o tisíce jednotek na posledním místě (ULPs) vzdáleny od nuly a výsledkem bude tedy vždy, že se dané číslo nerovná 0. Abychom se vyhnuli takovýmto situacím, je vhodné při implementaci funkce použít i kontrolu pomocí absolutní chyby, která si bez obtíží s případy v okolí nuly poradí.

[27]DAWSON, Bruce. Comparing Floating Point Numbers, 2012 Edition
[28]DAWSON, Bruce. Comparing Floating Point Numbers, 2012 Edition
[29]DAWSON, Bruce. Comparing Floating Point Numbers, 2012 Edition

4 KUŽELY

V následující krátké kapitole si uvedeme možnosti, jak reprezentujeme kužely tak, abychom následně snadno získali informaci o tom, zda robotické rameno leží celé uvnitř takto definovaných ploch. Možnosti jsou dvě. Jednak můžeme kužel popsat pomocí kvadratických ploch jako kuželovou plochu a rovinu, která představuje podstavu. Další způsob je analytická geometrie v prostoru, kdy kužel zadáváme jako kružnici podstavy a vrchol.

4.1 Kužely jako kvadratické plochy

Kužely mohou být reprezentovány pomocí kvadratických ploch. Kvadratickou plochou nazveme množinu bodů v prostoru, která splňuje rovnici:

$$a_{11}x^2 + a_{22}y^2 + a_{33}z^2 + 2a_{12}xy + 2a_{13}xz + 2a_{23}yz + 2a_{14}x + 2a_{24}y + 2a_{34}z + a_{44} = 0$$

přičemž alespoň jeden ze členů $a_{ij} \in \mathbb{R}$, kde i,j=1,2,3, je nenulový. Rovnice kvadrik bývají také reprezentovány maticovým tvarem[30]:

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix}$$

kdy se jedná o matici symetrickou. Podle hodnot jednotlivých koeficientů určujeme, o jakou konkrétní kvadratickou plochu se jedná. Rozlišujeme elipsoidy, hyperboloidy (jednodílný, dvojdílný), paraboloidy (eliptický, hyperbolický), válcové plochy a kuželové plochy.

Kuželová plocha je kvadrika, jejíž koeficient $a_{44} = 0$. V kartézské soustavě souřadnic má rovnici:

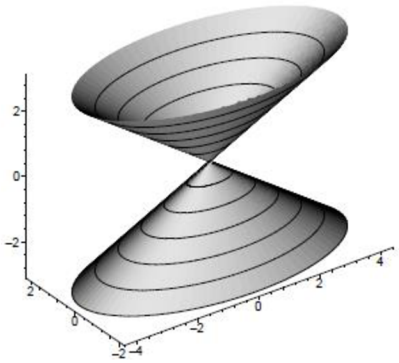
$$\frac{x^2}{a^2} + \frac{y^2}{b^2} - \frac{z^2}{c^2} = 0$$

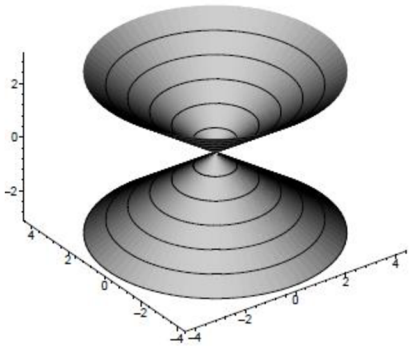
a jedná se o tzv. eliptickou kuželovou plochu (Obr. 16). Speciálním případem eliptické kuželové plochy je rotační kuželová plocha (Obr. 17), kdy se koeficienty rovnice *a* a *b* rovnají.

Kuželová plocha je středová singulární kvadrika, přičemž singulární znamená, že determinant matice kvadriky je roven 0. Mimo jiné to také znamená, že kvadrika má singulární bod. Singulární bod je středem kvadriky, který zároveň na této kvadratické ploše leží. Středem kvadriky rozumíme bod souměrnosti kvadriky, tedy ke každému bodu x_1 náležící kvadrice existuje bod x_2, taktéž ležící na ploše, kde střed kvadriky je

[30] HAŠEK, Roman a Pavel PECH. *Kvadratické plochy a jejich prezentace v programu Maple*.

středem úsečky $|x_1x_2|$.[31] V případě kuželové plochy je pak singulárním bodem zřejmě vrchol.





Obr. 16: Kuželová plocha[32]

Obr. 17: Rotační kuželová plocha[33]

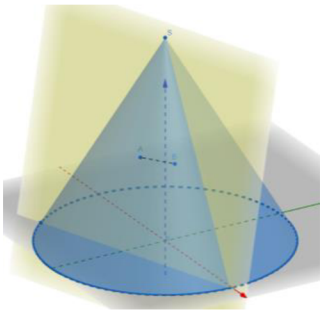
Výhodou výpočtu vzájemné polohy kuželu a úsečky pomocí kvadratických ploch je snadné dosazení bodů do rovnice kuželu, díky čemuž zjistíme, zda bod leží uvnitř, vně, nebo na kuželové ploše. Velkou nevýhodou je, že kuželová plocha jako celek nevymezuje konvexní oblast a proto ani v případě, že oba dva body úsečky leží uvnitř plochy, nemusí tam ležet i celá úsečka.

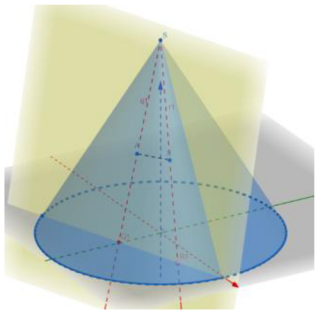
4.2 Kužely v analytické geometrii

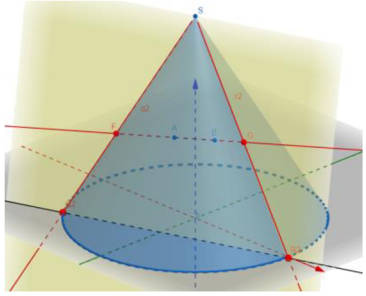
Druhým způsobem, jak jednoznačně definovat kužel v prostoru, je zadání kružnice a vrcholu. Oproti výše zmiňovaným kvadratickým plochám se vždy jedná o konvexní oblast, a stačí tedy dokázat, že dva body, reprezentující robotické rameno, leží uvnitř zadaného kuželu.

Nejdříve hledáme průsečíky přímky *p* (procházející úsečkou *AB*) s kuželem. Následně porovnáváme, zda oba body úsečky leží mezi nalezenými průsečíky. Průsečíky nalezneme sestrojením vrcholové roviny kuželu. Vrcholová rovina ρ je rovina procházející vrcholem a body *A* a *B* (Obr. 18). Dále provedeme průmět úsečky *AB* (resp. přímky *p*) do roviny podstavy tak, že vedeme přímky *q1* a *r1* z vrcholu *V* dvěma různými body na úsečce. Body vzniklé průsečíky přímek *q, r* s rovinou podstavy nazýváme půdorysnými stopníky a značíme *Q1* a *R1* (Obr. 19). Dále zkonstruujeme body *Q2*, *R2*, které jsou průsečíky s kružnicí podstavy. Každým z těchto bodů následně vedeme přímku *q2,r2* procházející vrcholem *V*. Závěrem stačí najít průsečíky *F* a *G* přímek *q2* a *r2* s původní *p*. Body jsou zároveň průsečíky přímky *p* se zadaným kuželem (Obr. 20). Jak bylo výše zmíněno, postačí nyní zkontrolovat, zda původně zadané body *A* a *B* leží mezi nalezenými průsečíky.

[31] HAŠEK, Roman a Pavel PECH. *Kvadratické plochy a jejich prezentace v programu Maple*.
[32] HAŠEK, Roman a Pavel PECH. *Kvadratické plochy a jejich prezentace v programu Maple*.
[33] HAŠEK, Roman a Pavel PECH. *Kvadratické plochy a jejich prezentace v programu Maple*.







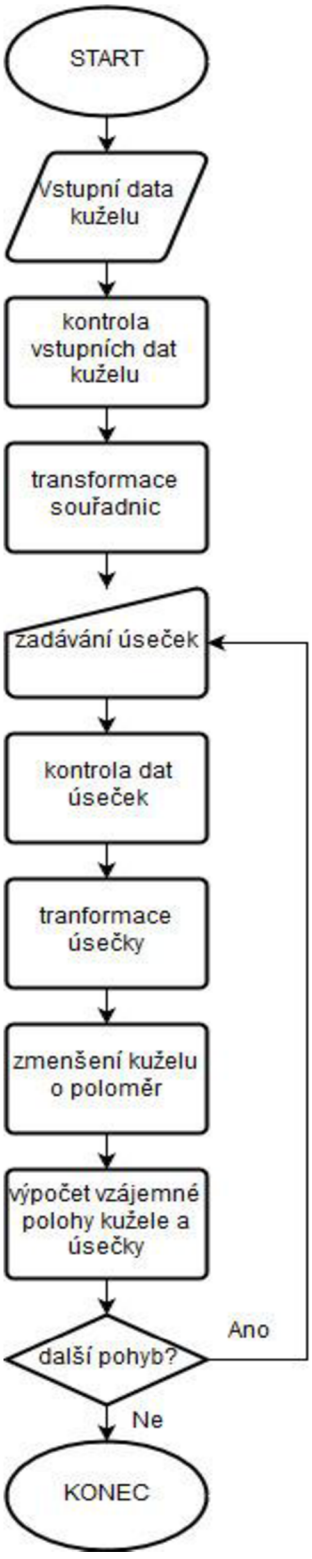
Obr. 18: Vrcholová rovina
Obr. 19: Půdorysné stopníky
Obr. 20: Průsečíky přímky a kužel

Zadání kuželu pomocí kružnice a vrcholu je vybráno pro realizaci algoritmu a to hned z několika důvodů. Jednak vycházíme z toho, že vstupní data od uživatele budou právě tři body tvořící kružnici a vrchol (nepředpokládáme, že uživatel zadá koeficienty rovnice kvadratické plochy), takže není třeba dalšího přepočtu. Oproti kuželové ploše se navíc jedná o konvexní těleso a není třeba před samotným výpočtem určovat, se kterou „půlkou“ kuželové plochy se bude pracovat. V kapitole o metodách vymezení pracovního prostoru jsme zmínili také zmenšení objektu o poloměr robotického ramene za účelem snazších výpočtů. Uvědomme si, že korektní zmenšení kuželové plochy by znamenalo rozpad plochy na dvě tělesa. To by se samozřejmě dalo obejít, avšak opět by to vyžadovalo přepočet koeficientů plochy. Zmenšení kuželu zadaného čtyřmi body je podstatně snazší a bude podrobněji rozebráno v kapitole o implementaci algoritmu. Obdobně je méně náročné transformovat (posunout, rotovat) jednotlivé body kuželu, než koeficienty kvadratické plochy.

5 IMPLEMENTACE PROGRAMU

V této kapitole si podrobně popíšeme praktickou část práce, tedy program, který počítá, zda zadané rameno robota, resp. úsečka zadaná dvěma body v prostoru a poloměrem, leží uvnitř zadaného kužele, resp. komolého kužele. Případ výpočtu komolého kužele se od kužele liší jen velmi nepatrně, proto si celý program popíšeme pouze jednou a na rozdílnosti upozorníme.

Jak bylo dříve zmíněno, úloha bude řešena pomocí analytické geometrie v prostoru. Na obrázku vpravo můžeme vidět obecný vývojový diagram programu. Tato kapitola bude rozdělena na dvě podkapitoly. V první podkapitole si představíme funkce, které prostupují celým programem, a proto je vhodné si je představit hned na začátek. Druhá podkapitola pak bude věnována hlavním funkcím programu tak, jak jsou popsány v obecném vývojovém diagramu.



Na úplný úvod před popisem jakýchkoliv funkcí ještě zmiňme, že jelikož pracujeme s geometrií, je naším základním prvkem bod. Každý bod je reprezentován polem o délce 3, jehož hodnoty jsou s dvojnásobnou přesností (double). Takovéto pole je v programu využíváno velmi často, jak pro výpočty bodů, tak pro počítání s vektory. Proto byl na začátku vytvořen nový univerzální uživatelský datový typ *coord* z anglického coordinates, tedy souřadnice.

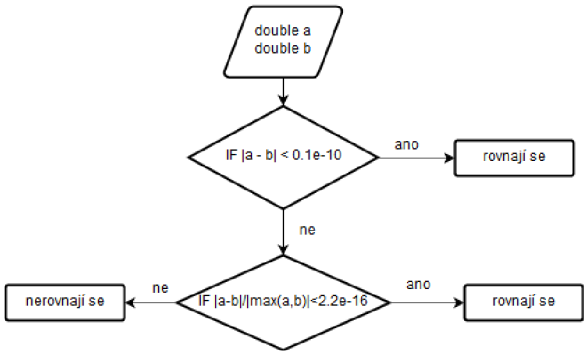
5.1 Základní funkce

Jak bylo již výše zmíněno, za základní funkce považujeme takové, které jsou používány v různých blocích programu a bylo by proto zbytečné představovat je opakovaně, nebo je zahrnovat ke konkrétnímu bloku. Mezi tyto funkce řadíme například funkci porovnávání hodnoty typu double *epsCompar()*, výpočty vzdáleností dvou bodů *distancePP()*, výpočet jednotkového normálového vektoru *calcNormal()* atd.

5.1.1 Porovnávání hodnoty typu double - epsCompar()

V kapitole o problematice nepřesností floatových hodnot jsme se zabývali nepřesností reálných čísel uložených v počítači. Při porovnávání čísel proto na možné nepřesnosti musíme brát ohled. Je třeba zmínit, že v našem programu pro dosažení větší přesnosti počítáme s hodnotami typu double, principiálně se však nic nemění.

Funkce *epsCompar()* je určená právě k rozhodnutí, zda dvě hodnoty typu double můžeme považovat za stejné na základě relativní chyby. Vstupem funkce jsou pouze dvě srovnávané hodnoty, výstup je pak hodnota typu bool, která má hodnotu true, pokud můžeme čísla považovat za stejná a false v případě opačném. Jak bylo v kapitole 3 zmíněno, jelikož v okolí nuly je výpočet relativní odchylky problematický, počítáme i relativní odchylku. Vývojový diagram funkce vidíme na (Obr. 22). Čísla se rovnají, pokud alespoň jedna z odchylek je menší, než nastavené konstanty.



Obr. 22: Vývojový diagram epsCompar()

5.1.2 Vzdálenost dvou bodů distancePP()

Tato funkce počítá primárně vzdálenost dvou bodů, ovšem jako vedlejší produkt generuje i jednotkový směrový vektor mezi těmito body. V programu se této vlastnosti také často využívá. Vstupními parametry jsou tři hodnoty typu coord. První dvě představují dva body, mezi kterými určujeme vzdálenost (A, B). Třetí slouží právě k uložení směrového vektoru *u*. Vzdálenost počítáme standardně pomocí:

$$u_i = A_i - B_i$$

$$|AB| = \sqrt{\sum u_i^2}$$

Jednotlivé prvky směrového vektoru pak dělíme délkou, abychom získali jednotkový směrový vektor.

5.1.3 Normálový vektor calcNormal()

Vstupní hodnoty funkce jsou tři různé body (*X*, *Y*, *Z*) tvořící rovinu, ke kterým spočítáme normálový vektor. Čtvrtou vstupní hodnotou je indikátor chyby typu bool (*err*) indikátor je nastaven na hodnotu true, pokud zadané body leží na přímce a tedy je velikost normálového vektoru nulová.

Funkce k výpočtu normály využívá funkci vektorového součinu (*crossProduct*), kde vstupy jsou body *X*, *Y*, *Z* a výstupem vektor. Následně je spočítána délka vektoru a jednotlivé složky jsou touto délkou vyděleny:

$$l = \sqrt{x^2 + y^2 + z^2}$$

$$n = (\frac{x}{l}, \frac{y}{l}, \frac{z}{l})$$

Přičemž l nesmí být rovna nule. Návratová hodnota funkce odpovídá jednotkovému normálovému vektoru, avšak než vektor zahrneme do dalších výpočtů, musíme vždy zkontrolovat chybový indikátor.

5.1.4 Rovnice roviny planeEqv()

Tato funkce počítá obecnou rovnici roviny resp. její koeficienty (a, b, c, d). Rovina je zadaná třemi body v prostoru (A, B, C). Vstupními parametry jsou tyto tři body a indikátor chyby typu bool (*error*). Výstupem je pak čtyřmístné pole hodnot typu double, které odpovídají koeficientům rovnice. Pomocí funkce na výpočet normálového vektoru dostáváme první tři koeficienty rovnice a zároveň hodnotu indikátoru chyby. Čtvrtý koeficient dopočítáme následovně:

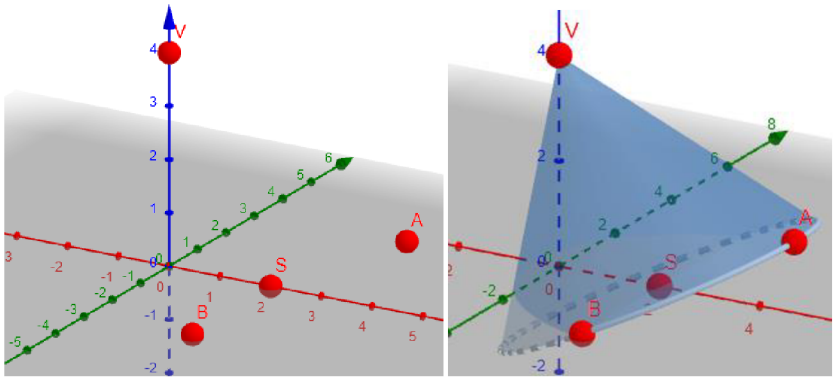
$$\rho = ax + by + cz + d$$
$$d = -(a * A_x + b * A_y + c * A_z)$$

resp: `planeEquation[3]=-(normal[0]*A[0]+normal[1]*A[1]+normal[2]*A[2]);`

5.2 Hlavní funkce programu

5.2.1 Vstupní data kuželu a jejich kontrola

Kužel je v rámci programu jednoznačně určen čtyřmi body, a to vrcholem, středem kružnice podstavy a dvěma body ležícími na této kružnici (Obr. 23). Tyto body nám určují rovinu podstavy kužele, proto nesmí se středem kružnice tvořit přímku. Pokud počítáme komolý kužel, pak jsou vstupy kromě čtyř výše uvedených bodů, také tři body určující řeznou rovinu. Než se však pustíme do výpočtů, je vhodné ověřit, že tyto body jsou správně zadané, resp. že opravdu jednoznačně definují kužel resp. komolý kužel. Pokud by tomu tak nebylo, tak by byly výpočty zcela zbytečné a výsledek chybný, případně by nastala chyba už v průběhu výpočtů.



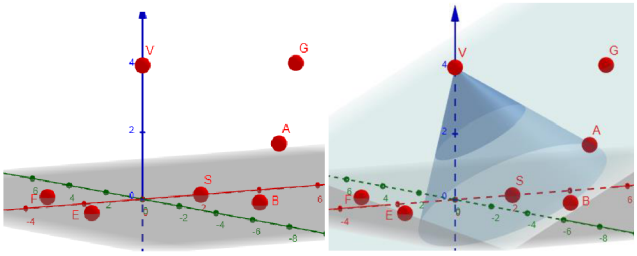
Obr. 23: Určení kuželu pomocí čtyř bodů

Pro kontrolu vstupních bodů kuželu je tedy vytvořena funkce *inputCheck()* („kontrola vstupů“). Funkce má 6 vstupních argumentů, přičemž první čtyři jsou výše zmiňované body v pořadí vrchol V, střed kružnice So a dva body na kružnici X, Y.

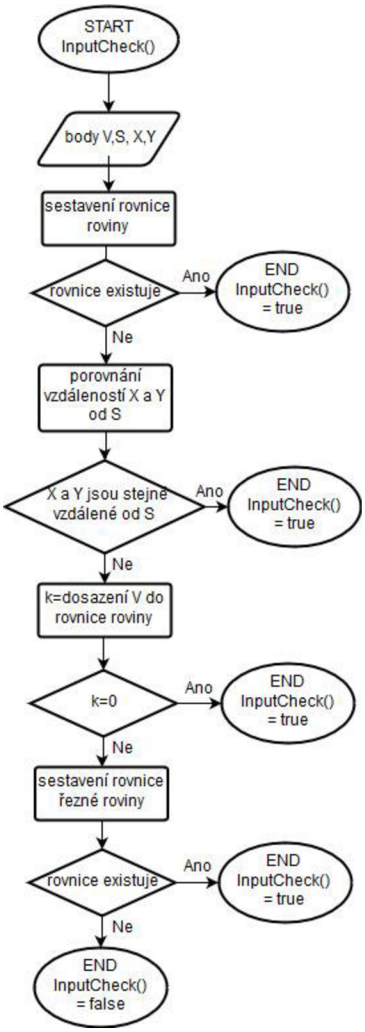
Dalším argumentem je pole o délce 4 a přesnosti double (SXY). Vzorové volání funkce:

```
inputCheck(V, So, X, Y, SXY);
```

Toto pole je vyhrazeno pro parametry obecné rovnice roviny podstavy a je pouze deklarované, inicializace proběhne až v rámci funkce. Posledním vstupem je dvouhodnotová reference typu bool (halfSpace). Ta je taktéž pouze deklarovaná a bude vyjadřovat, ve které polorovině roviny SXY leží vrchol *V*, což je důležité pro další části programu. V programu pro výpočet komolého kuželu jsou ještě další tři body (E,F,G) pro určení druhé roviny (Obr. 24). Funkce *inputCheck()* má návratovou hodnotu typu bool, která odpovídá ve funkci proměnné *err*. Nabývá hodnoty false, pokud zadané body tvoří kužel, hodnoty true v opačném případě. Vývojový diagram funkce *inputCheck* pro komolý kužel vidíme na obrázku vpravo.



Obr. 24: Definice komolého kuželu



Abychom mohli prohlásit, že zadané body tvoří kužel, musíme ověřit tři skutečnosti:

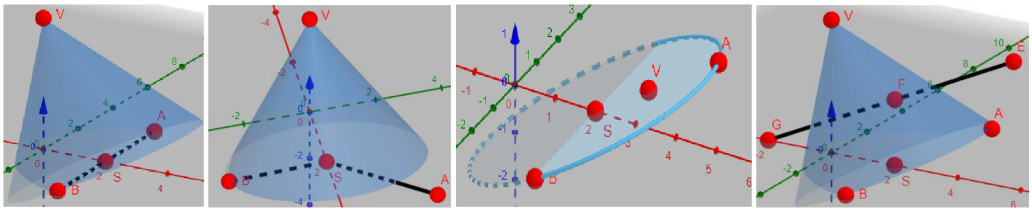
1. Zadané body S,X,*Y* musí tvořit rovinu a tedy nesmí ležet na jedné přímce. To ověříme výpočtem rovnice roviny pomocí *planeEqv()*. Pokud má indikátor chyby hodnotu true, pak *S*, X, *Y* rovinu netvoří a celkový výstup funkce je true, tedy chybná data.
2. Dále je třeba ověřit, že body X a *Y* opravdu leží na jedné kružnici. Proto vypočteme pro každý z bodů vzdálenost od středu pomocí funkce *distPP()*. Jelikož potřebujeme porovnat hodnoty typu double, použijeme funkci *epsCompar()* a vypočtené vzdálenosti pomocí ní srovnáme. V případě, že se nerovnají, resp. je nemůžeme považovat za stejné, výstupem funkce je opět chyba.
3. Poslední ověření spočívá v kontrole, zda vrchol *V* neleží v rovině podstavy. Jelikož již s prvního bodu máme spočítanou rovnici roviny, stačí do ní dosadit bod *V* a zkoumat (opět pomocí *epsCompar()*), zda je výsledek roven nule. V případě že ano, pak je opět výstupem funkce chyba.

Než se posuneme ve výpočtu dál, uložíme si znaménko výsledné

hodnoty po dosazení. Toto znaménko nám totiž určuje poloprostor, ve kterém leží vrchol kuželu. Zjistíme ho pomocí *signbit()*, která nabývá hodnoty true, pokud je znaménko záporné a false, pokud je kladné.

Pokud jsou všechny 3 výše uvedené body v pořádku, pak máme korektně zadaný kužel a můžeme pokračovat ve výpočtu. Pokud počítáme komolý kužel, tak jsou součástí funkce *inputCheck()* ještě:

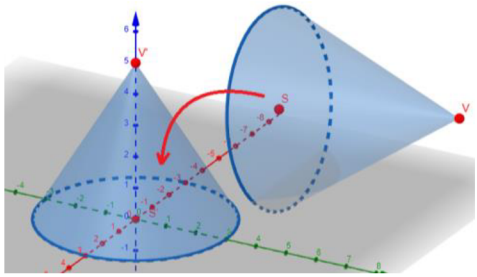
4. Kontrola pomocí *planeEqv()*, zda zadané vstupy, představující řeznou rovinu *EFG* opravdu jednoznačně rovinu určují.
5. Zjištění, zda jsou roviny *EFG* a *SXY* rovnoběžné, různoběžné, případně totožné. V případě, že jsou dvě roviny rovnoběžné, pak mají shodné normálové vektory, které se mohou lišit znaménkem. Pokud navíc do roviny *SXY* dosadíme jeden z bodů určující rovinu *EFG* a výsledek je roven nule, pak jsou roviny dokonce shodné a řešit úlohu jako komolý kužel je zbytečné (funkce vrátí chybu). Navíc, v případě rovnoběžnosti rovin kontrolujeme, zda rovina podstavy neleží v opačné polorovině roviny *SXY* než vrchol *V*. V takovém případě by bylo zřejmě opět bezpředmětné řešit úlohu jako komolý kužel.



Obr. 25: Chybně zadané kužely

5.2.2 Transformace souřadnic

Kužel může být v prostoru zadán prakticky kdekoliv a úloha bude řešitelná. Ovšem snadnějších a především rychlejších výpočtů můžeme dosáhnout při vhodném posunutí, případně natočení souřadného systému. V našem případě volíme takové posunutí a natočení, aby kružnice definující podstavu kuželu ležela v rovině dané osami *XY* a její střed ležel v počátku souřadného systému (Obr. 26). Nejdříve provedeme posunutí středu kuželu do počátku souřadnicového systému. Toho je docíleno tak, že od každého bodu kuželu odečteme souřadnice středu *S*. Dalším krokem je získání rotační matice pro natočení podstavy kuželu do roviny *XY*. V naší implementaci budeme souřadnice transformovat pomocí kvaternionů.



Obr. 26: Transformace souřadnic

Kvaterniony jsou nekomutativním zobecněním komplexních čísel v trojrozměrném prostoru, kdy imaginárními částmi jsou *i, j, k* a platí[34]:

$$i^2 = j^2 = k^2 = ijk = -1$$

Kvaternion může být vyjádřen jako:

$$H = a + bi + cj + dk$$

Pro rotaci v 3D prostoru je kvaternion určen následovně[35]:

$$Q = \cos\left(\frac{\alpha}{2}\right) + i\left(x * \sin\left(\frac{\alpha}{2}\right)\right) + j\left(y * \sin\left(\frac{\alpha}{2}\right)\right) + k\left(z * \sin\left(\frac{\alpha}{2}\right)\right)$$

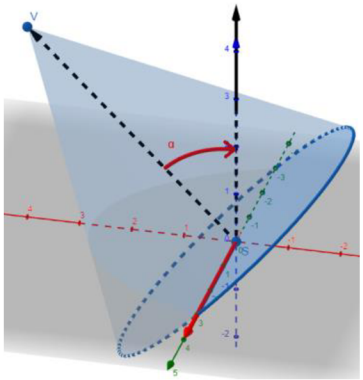
označme:

$$qw = \cos\left(\frac{\alpha}{2}\right), \quad qx = x * \sin\left(\frac{\alpha}{2}\right), \quad qy = y * \sin\left(\frac{\alpha}{2}\right) \quad \text{a} \quad qz = z * \sin\left(\frac{\alpha}{2}\right)$$

kde α je úhel, o který chceme objekt rotovat a *x,y,z* jsou koeficienty vektoru, kolem kterých chceme těleso rotovat. V našem případě může být osou rotace souřadnicová osa *x* nebo *y*. Úhel α je úhel mezi souřadnicovou osou z a vektorem $\overrightarrow{SV}$ (Obr. 27). Matici rotace získáme takto[36]:

$$R = \begin{matrix} 1 - 2 * \mathrm{qy}^2 - 2 * \mathrm{qz}^2 & 2 * \mathrm{qx} * \mathrm{qy} - 2 * \mathrm{qz} * \mathrm{qw} & 2 * \mathrm{qx} * \mathrm{qz} + 2 * \mathrm{qy} * \mathrm{qw} \\ 2 * \mathrm{qx} * \mathrm{qy} + 2 * \mathrm{qz} * \mathrm{qw} & 1 - 2 * \mathrm{qx}^2 - 2 * \mathrm{qz}^2 & 2 * \mathrm{qy} * \mathrm{qz} - 2 * \mathrm{qx} * \mathrm{qw} \\ 2 * \mathrm{qx} * \mathrm{qz} - 2 * \mathrm{qy} * \mathrm{qw} & 2 * \mathrm{qy} * \mathrm{qz} + 2 * \mathrm{qx} * \mathrm{qw} & 1 - 2 * \mathrm{qx}^2 - 2 * \mathrm{qy}^2 \end{matrix}$$

Nyní stačí každý bod, který chceme rotovat (tedy *V*, *S*, X, *Y*), vynásobit zprava nebo zleva maticí *R* (v závislosti na z-ové souřadnici rovnice roviny *SXZ*).



Obr. 27: Rotace okolo osy y o úhel α

5.2.3 Zadávání úseček, kontrola dat a transformace úsečky

Když máme vhodně transformované souřadnice kuželu, začíná iterační část programu, kdy postupně načítáme polohy jednotlivých dílů robota a kontrolujeme, jestli leží uvnitř kuželu. Než však dojde k hlavní výpočtové části, je třeba zadané hodnoty opět zkontrolovat a předpřipravit pro výpočet.

Jak již bylo v druhé kapitole popisováno, robotické rameno pro účely výpočtů aproximujeme jako válec, resp. jako úsečku s poloměrem. Vstupní data budou proto dva krajní body úsečky (*A*, *B*) a poloměr (*ra*). V první řadě

[34] Weisstein, Eric W. "Quaternion." From *MathWorld*--A Wolfram Web Resource. http://mathworld.wolfram.com/Quaternion.html

[35] http://www.euclideanspace.com/maths/algebra/realNormedAlgebra/quaternions/

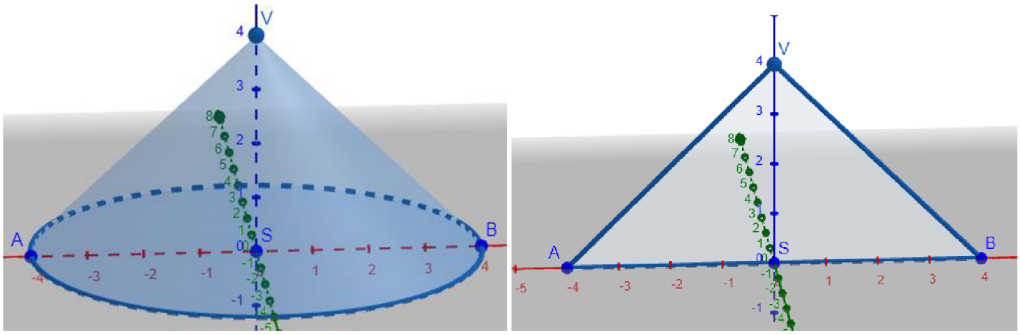
[36] http://www.euclideanspace.com/maths/geometry/rotations/conversions/quaternionToMatrix/index.htm

proběhne kontrola, zda se jedná o dva různé body, tedy že se opravdu jedná o úsečku. Kontrola proběhne opět pomocí funkce *epsCompar()* pro každou dvojici souřadnic *A* a *B*. Dalším krokem bude transformace souřadnic bodů úsečky pomocí funkce rotation() a posunutí o vektor $\overrightarrow{SoS}$.

Nyní, když je zkontrolována platnost vstupních hodnot a úsečka je transformována do posunutých souřadnic kuželu, můžeme pokračovat v hledání vzájemné polohy kuželového prostoru a válce, který úsečka s poloměrem představuje. Jelikož je však obtížné pro každou polohu dopočítávat celý válec a následně klasifikovat polohu každého bodu na válci vůči našemu kuželu, provedeme zmenšení kužele o poloměr.

5.2.4 Zmenšení kužele o poloměr rescale()

Nastává tedy otázka, jak zmenšit celý kužel o poloměr *ra*. Uvědomme si, že nyní jsme ve fázi, kdy máme celý kužel definovaný jeho vrcholem, středem podstavy, který leží v počátku soustavy souřadnic a poloměrem podstavy, která leží v rovině XY. Pokud tedy chceme celý takto nadefinovaný kužel zmenšit o zadané *ra*, pak stačí pouze tyto body vhodně posunout. K tomu však není třeba počítat s celým objemem kužele, ale pouze s jeho rovinným řezem, procházejícím středem, vrcholem a pro snadné počítání body *X* a *Y*, které představují průsečíky podstavové kružnice kuželu s osou x viz (Obr. 28). Ke zmenšení kuželu slouží v programu funkce *rescale()*, jejímiž vstupními argumenty je vrchol *V*, střed *S*, reference nový střed *Sn* a nový vrchol *Vn* a poloměr robotického ramene *ra*. Výstup je prázdný, jelikož potřebné výsledné hodnoty najdeme po běhu funkce uložené v nadeklarovaných proměnných. V případě, že počítáme s komolým kuželem, pak navíc posouváme řeznou rovinu.

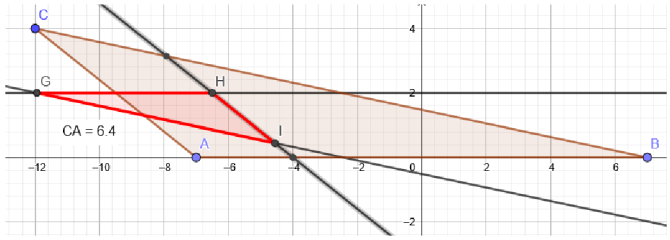


Obr. 28: Rovinný řez kuželu postačující pro zmenšení

Nyní tedy stačí zmenšit trojúhelník definovaný body *X*, *Y* a *V* o poloměr *ra*. Nejdříve však musíme provést kontrolu, zda má zmenšování smysl, resp. není-li poloměr ramene příliš velký na to, aby se s celým ramenem mohlo vůbec jakkoliv pohnout tak, aby nepřesahovalo hranice zadaného kuželu. Kontrola proběhne tak, že pomocí funkce *distPP()* postupně změříme vzdálenosti |XV| a |YV|. Je zřejmé, že vzdálenost |*XY*| je rovna dvojnásobku poloměru r kuželu. Následně vybereme nejmenší z těchto vzdáleností a zkontrolujeme, zda je větší než dvojnásobek poloměru ramene *ra* (dvojnásobek jelikož úsečku zkracujeme o *ra* na každé straně). V případě že ne, další

výpočty nemají smysl, jelikož není možné ramenem pohnout tak, aby neopustilo povolený prostor. Teoreticky by bylo možné připustit i rovnost, kdy by byl pohyb možný pouze po přímce procházející kolmo středem příslušné úsečky. S ohledem na bezpečnost výpočtu však upřednostňujeme ostrou nerovnost.

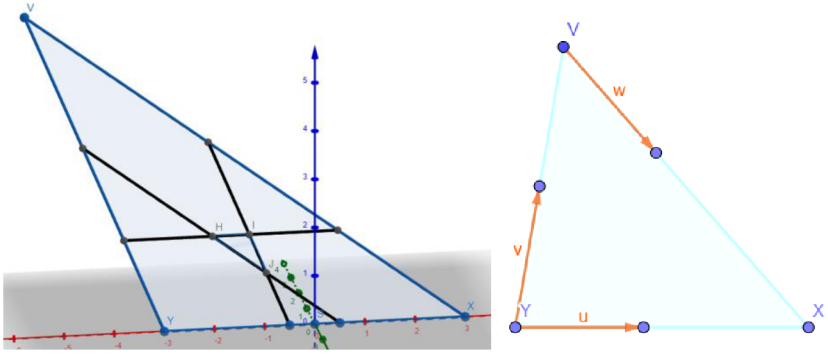
Předešlá kontrola je nutná, avšak ne postačující podmínka k tomu, aby zmenšený trojúhelník vznikl. Podívejme se například na (Obr. 29), kdy každá ze stran trojúhelníku je delší, než 2*poloměr ramene, avšak výsledný prostor pro pohyb ramene je přesto nulový, protože dojde k „překlopení“ zmenšeného trojúhelníku. Tuto skutečnost však budeme ověřovat až po zmenšení trojúhelníku na základě vzájemné polohy původních bodů *S*, *X* a *Y* a jejich nových poloh.



Obr. 29: Nedostatečná podmínka kontroly vzniku zmenšeného kuželu

Zmenšení provedeme tak, že k jednotlivým přímkám tvořící strany trojúhelníku vytvoříme rovnoběžky ležící v rovině trojúhelníku ve vzdálenosti *ra* směrem do středu trojúhelníku. Následně spočítáme vzájemné průsečíky těchto rovnoběžek a tím dostaneme vrcholy zmenšeného trojúhelníku. Máme-li tyto přímky zadané parametricky (tedy bodem a směrovým vektorem), pak posunout přímku znamená posunout bod, kterým je zadána.

V případě funkce *rescale()* si nejdříve pomocí funkce *distancePP()* určíme směrové vektory (*u*,*v*,*w*) mezi vrcholy trojúhelníku. Na rozdíl od použití funkce pouze pro určení vzdálenosti kdy pořadí zadávání bodů bylo nepodstatné, nyní musíme pořadí volit obezřetně s ohledem na směr, který chceme, aby daný vektor představoval. Na (Obr. 30) vidíme, jak jsou vektory prvotně vypočítány. V průběhu výpočtu jsou však i otáčeny, aby vyhovovaly výpočtu.



Obr. 30: Konstrukce zmenšeného kuželu

Přímky jsou definovány následně: (*X*, *w*), (*Y*, *u*) a (*V*, *v*). Nyní je tedy potřeba vypočítat vektory kolmé na jednotlivé přímky, ležící v rovině trojúhelníku a zároveň směřující dovnitř. Ty získáme dvojitým použitím vektorového součinu. Chceme-li posouvat například přímku se směrnicí v, pak uděláme vektorový součin $u \times v = n1$, kde *n1* je normála k ploše trojúhelníku. Následně počítáme $v \times n1 = n1$. Nyní n1 představuje jednotkový vektor kolmý na přímku (*V*, *v*). Vektor n1 nyní vynásobíme poloměrem ramene ra a původní vrchol *V* o tento vektor posuneme, čímž získáváme novou polohu vrcholu (Vn). Obdobně dopočítáme i další vrcholy, nicméně musíme vždy dbát na to, aby vektory, se kterými počítáme, byly orientovány směrem ven z posouvaného vrcholu, tzn., aby směřovaly do vrcholu jiného. V takovém případě totiž, bez ohledu na to, jestli je vrchol *V* v kladné či záporné polorovině osy *z*, vždy posouváme bod směrem dovnitř trojúhelníku a ne ven (ověřitelné pomocí pravidla pravé ruky).

Nyní máme tři parametricky zadané přímky definující zmenšený trojúhelník. Teď stačí pouze najít jejich průsečíky odpovídající novým vrcholům zmenšeného trojúhelníku. K výpočtu průsečíku dvou přímek zadaných parametricky nám poslouží funkce interLL(), kterou použijeme na každou dvojici přímek.

5.2.4.1 Funkce interLL()

Vstupními argumenty funkce jsou po řadě bod *A* a směrnice jedné přímky a bod *B* a směrnice přímky druhé. Výstupem jsou souřadnice průsečíku přímek. Výpočet probíhá na základě určení parametru *s* ze soustavy tří rovnic o dvou neznámých následovně:

$$x: A_x + t * u_x = B_x + s * v_x$$
$$y: A_y + t * u_y = B_y + s * v_y$$

Z první rovnice si vyjádříme t:

$$t = \frac{B_x - A_x + s * v_x}{u_x}$$

Dosadíme do druhé:

$$A_y + \frac{(B_x - A_x + s * v_x)}{u_x} * u_y = B_y + s * v_y$$

Odtud:

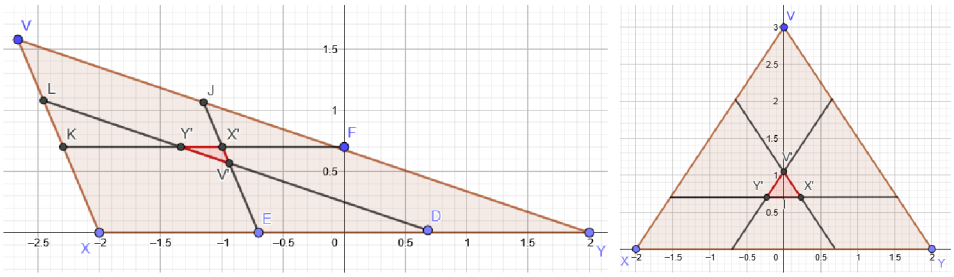
$$\frac{A_y - B_y + \frac{(B_x - A_x)}{u_x} * u_y}{(v_y - \frac{v_x * u_y}{u_x})} = s$$

Souřadnice průsečíku pak dostaneme dosazením s do parametrického zadání druhé přímky. V kódu výpočet vypadá následovně:

```
double s =(A[1]-B[1]+(u[1]/u[0])*(B[0]-A[0])) / (v[1]-(v[0]*u[1]/u[0]));
coord interPoint = {B[0]+s*v[0],B[1] + s*v[1] ,B[2] + s*v[2] };
```

Právě jsme získali souřadnice posunutých vrcholů trojúhelníku. Zbývá tedy ověřit, zda není poloměr ramene *ra* robota příliš velký pro pohyb. Jak jsme zmínili výše, provedeme to pomocí porovnání s původními polohami bodů následovně:

- Na počátku jsme měli body *X* a *Y* nadefinované tak, že *Y* leží v záporné části a bod *X* leží v kladné části osy *x*. Body se mohou v rámci osy x v průběhu výpočtu posunout, nicméně stále musí platit, že x-ová složka nové souřadnice *Xn* musí být větší, než x-ová souřadnice Yn (Obr. 31).
- Dále leží-li vrchol *V* v kladné polorovině osy z (tedy jeho z-ová složka je větší než z-ové složky *X* a *Y*, které jsou samozřejmě nulové), tak i z-ová složka *Vn* musí být větší, než z-ové složky *Xn* resp. *Yn*. Naopak pokud *V* leží v záporné polorovině osy z pak i posunutý vrchol *V* musí ležet tamtéž.



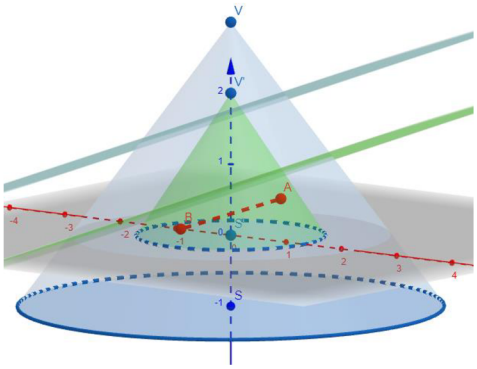
Obr. 31: Nezmenšitelný vs. zmenšitelný kužel

V případě, že nějaká z podmínek není splněna, pak výpočet končí s hodnotou false, tedy že rameno se nenachází v kuželu. Naopak pokud jsou podmínky splněny, můžeme dopočítat nový střed kuželu a nový poloměr:

```
Sn= { Yn[0] + ((Xn[0] - Yn[0]) / 2), Yn[1] + ((Xn[1] - Yn[1]) / 2), Xn[2] };
r = distancePP(Sn, Xn, u);
```

Pokud navíc počítáme úlohu s komolým kuželem, pak posouváme i řeznou rovinu, a to ve směru k původnímu středu. Následně zkoumáme, zda body *A* a *B* zadaného ramene stále leží ve stejné polorovině, jako před posunutím (Obr. 32). Obrázek také demonstruje, že naše verze komolého kuželu není v matematickém smyslu podobná kuželu původnímu, tzn., že není zachován poměr objemů, na které rovina kužel dělí. Připomeňme si proto, že v robotice nehledáme přímo zmenšení původního kuželu, nýbrž množinu bodů, která má od původního komolého kuželu vzdálenost alespoň r, což je rozdíl. Operaci kontroly polohy *A* a *B* vůči posunuté rovině předřadíme zmenšování kuželu, jelikož je rychlejší, takže v případě, že je detekována chyba, ušetříme si větší objem výpočtů. Posunutí provádíme tak, že rovinu přeneseme o r_a-násobek normálového vektoru směrem ke středu *S*. Posunout celou rovinu znamená posunout body *E*, *F*, *G* a spočítat novou rovnici roviny. Nejdříve posuneme jeden bod (například *E*) a zjistíme, zda takto posunutý bod leží ve stejné polorovině roviny *EFG*, jako střed *S*. V případě, že ano, pak posuneme zbylé body tvořící rovinu. Pokud ne, pak počítáme s posunutím na opačnou stranu (vynásobíme normálu -1). Dále ověříme, zda

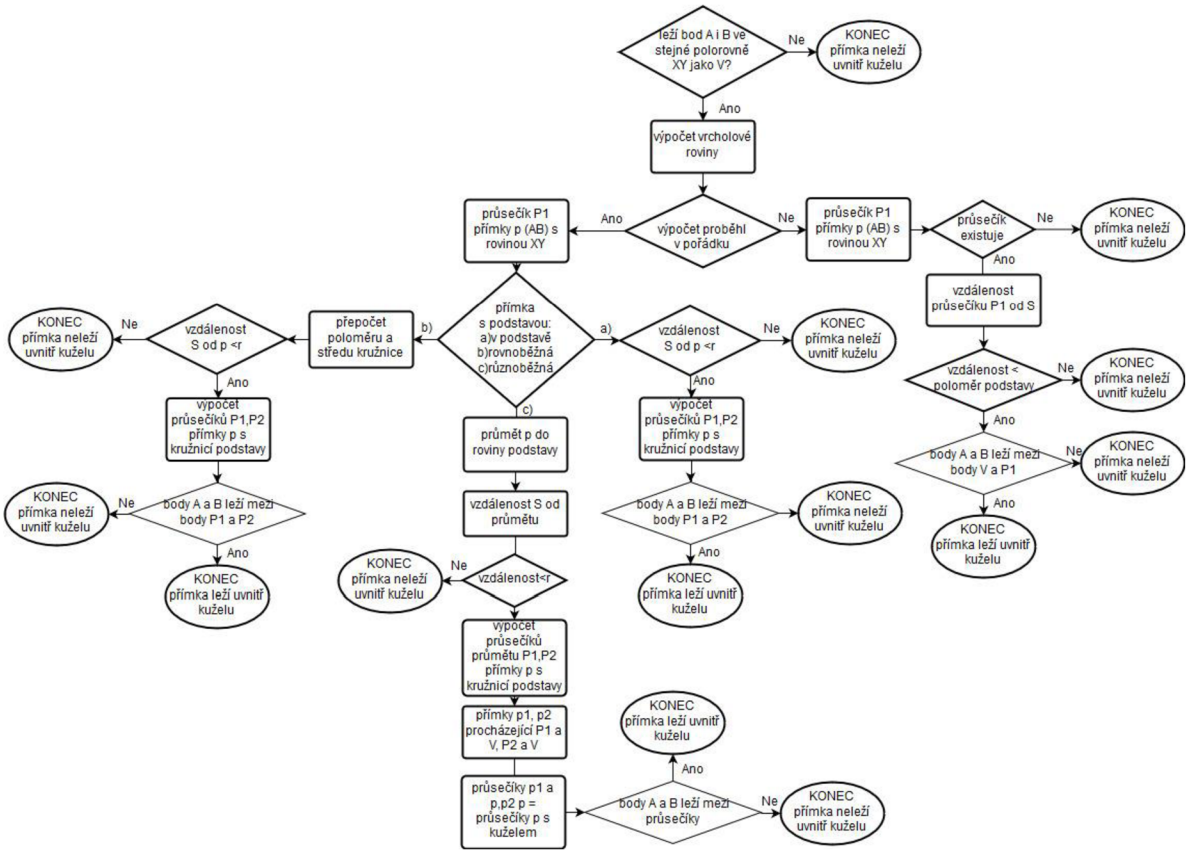
je střed *S* stále ve stejné polorovině *EFG*, jako před posunutím. Pokud ano, pokračujeme zmenšením celého kuželu. Pokud ne, můžeme další výpočty přeskočit.



Obr. 32: Posunutí roviny

Zároveň v této části provedeme kontrolu, zda body *A*, *B* leží ve správné polorovině. Pokud ne výpočet končí chybou. V případě, že ano, pak jsme dořešili úlohu komolého kuželu a pokračujeme shodně s výpočtem klasického kuželu.

Pro snazší počítání opět posuneme střed do počátku souřadného systému *Sn*->*S*. Jelikož jsme rovinu podstavy pouze posouvali, není třeba další natáčení. Zároveň samozřejmě posouváme i všechny používané body (*Vn*, *A*, *B*) tak, že od nich odečteme *Sn*. Nyní již máme hodnoty připraveny a zkontrolovány pro hlavní část programu, která určuje vzájemnou polohu úsečky a kuželu. Vývojový diagram pro výpočet vzájemné polohy vidíme na (Obr. 33).



Obr. 33: Vývojový diagram výpočtu vzájemné polohy úsečky a kuželu

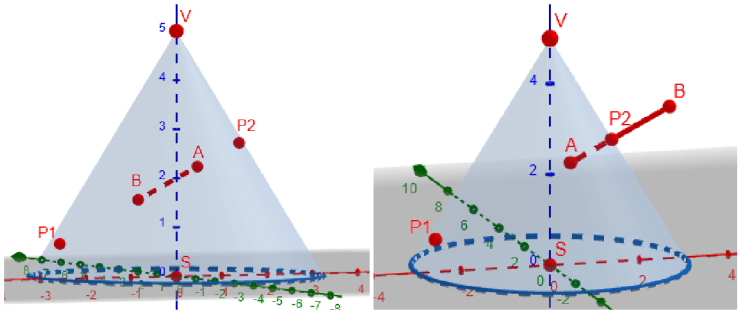
5.2.5 Výpočet vzájemné polohy úsečky a kužele

V prvním kroku si nadefinujeme koeficienty rovnice roviny *XY*: *xyPlane* = {0,0,1,0}, ve které leží podstava kuželu a kterou budeme dále využívat. Začneme tím výpočtově nejjednodušším, tedy ověřením, zda oba body úsečky (*A*, *B*) leží ve stejné polorovině roviny *XY*. Toho docílíme porovnáním znamének hodnot, které vyjdou při dosazení bodů *X*, resp. *Y* do rovnice roviny *XY*. Je tedy zřejmé, že porovnáváme pouze znaménka z-ových souřadnic. Znaménko dosazeného vrcholu *V* máme uložené v proměnné halfSpace. Pokud znaménka v alespoň jednom případě nejsou stejná, pak zadaná úsečka zcela jistě leží minimálně jednou svou částí mimo kužel, a algoritmus končí s chybovým výstupem.

Výpočet pokračuje tak, že hledáme průsečíky přímky procházející body *A* a *B* s kuželem. Následně, pokud oba průsečíky existují, ověřujeme pomocí funkce *comparison()*. Ta stanoví, zda body *A* i *B* leží mezi průsečíky interPoint1, interPoint2.

5.2.5.1 Funkce comparison()

Vstupními hodnotami jsou dva průsečíky přímky procházející body *A* a *B* s kuželem a samotné body úsečky *A*, *B*. Výstupem je pak hodnota typu bool, která nabývá hodnoty true, pokud *A* i *B* leží mezi zadanými průsečíky. Funkce pracuje tak, že pro každou souřadnici zvlášť nejdříve zjistí, který z průsečíků v ní nabývá vyšší hodnoty, a poté srovnává, zda jsou dané souřadnice *A* i *B* menší nebo rovny souřadnici průsečíku s vyšší hodnotou a vyšší nebo rovny souřadnici průsečíku s nižší hodnotou (Obr. 34). Rovnost samozřejmě porovnáváme pomocí funkce epsCompar(). V případě, že alespoň jediná souřadnice bodu A nebo B neleží mezi souřadnicemi průsečíků, pak úsečka neleží celá uvnitř kuželu a výpočet polohy končí s chybou (úsečka leží mimo kužel).



Obr. 34: Úsečka leží vs. úsečka neleží mezi dvěma průsečíky

5.2.5.2 Funkce intersection()

Další důležitou funkcí pro hledání průsečíků přímky s kuželem je funkce pro najití průsečíku přímky a roviny *intersection()* s následujícími vstupními parametry: dva body (*A*, *B*) definující přímku, rovnice roviny (*planeEquation*), se kterou chceme najít průsečík, neinicializovaná proměnná typu integer (*position*), do které později uložíme informaci o tom, jakou má přímka s rovinou vzájemnou polohu a bod (*interPoint*), do kterého chceme uložit souřadnice průsečíku, pokud existuje. Výstup je prázdný, jelikož potřebné hodnoty ukládáme do vstupně-výstupních proměnných.

Nejdříve z bodů *A* a *B* určíme směrový vektor přímky. Následně uděláme skalární součin směrového vektoru přímky a normálového vektoru roviny. Výsledek uložíme do proměnné *par*. V případě, že je par roven 0 (standardně porovnáváme pomocí *espComp()*), pak jsou na sebe vektory kolmé a tedy je přímka s rovinou rovnoběžná, nebo v ní leží. Dosadíme tedy do rovnice roviny bod *A* a opět pomocí *epsCompar()* zjišťujeme, zda se rovná nule. Pokud ano, pak přímka leží v rovině a do proměnné *position* ukládáme hodnotu 0. V opačném případě je přímka s rovinou pouze rovnoběžná a nastvíme *position=-1*.

Pokud je parametr *par* různý od nuly, pak je přímka s rovinou různoběžná a můžeme hledat průsečík. Máme tedy přímku zadanou parametricky: $p_i = A_i + t * u_i$ a hledáme parametr *t* následovně:
Dosadíme jednotlivé složky parametrického zadání přímky do obecné rovnice roviny:

$$a * (A_x + t * u_x) + b * (A_y + t * u_y) + c * (A_z + t * u_z) + d = 0$$

Odtud vyjádříme neznámý parametr t:

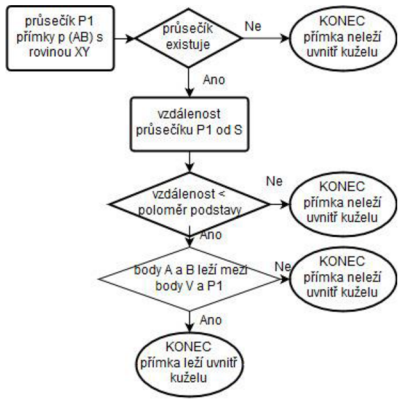
$$t = -\frac{a * A_x + b * A_y + c * A_z + d}{a * u_x + b * u_y + c * u_z}$$

Parametr zpětně dosadíme a získáváme průsečík. Do proměnné position ukládáme 1.

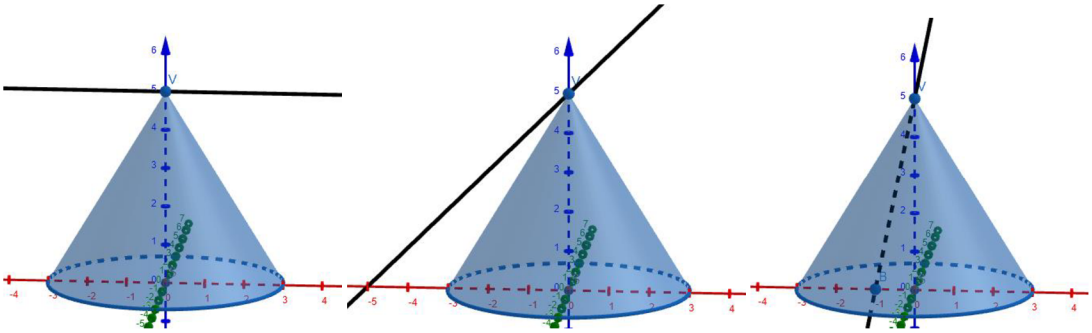
Pojďme si tedy ukázat, jak průsečíky s kuželem můžeme získat, případně určit, že neexistují. Při výpočtu může nastat pět různých situací. Začněme sestrojením vrcholové roviny, tvořené body *A*, *B*, *V*, pomocí funkce planeEqv(). Tímto se nám celá úloha dělí na dvě podúlohy. První podúloha, kdy *planeEqv()* vrátí indikátor chyby *error* s hodnotou true a tedy zkoumáme případ, kdy přímka p prochází jak body úsečky, tak vrcholem. Druhý zkoumaný případ je, když vrcholová rovina vznikne.

A) Body A, B a V leží na přímce

V případě, že body A,B,V leží na jedné přímce, mohou nastat celkem dva případy, tedy že přímka prochází kuželem a tedy má průsečík s rovinou XY uvnitř kruhové podstavy nebo leží zcela mimo viz(Obr. 35). Polohu zjistíme pomocí výpočtu průsečíku přímky s rovinou pomocí funkce *intersection()*. Funkce *intersection()* může vrátit hodnotu $pos \in \{-1,1\}$. Hodnotu 0 vrátit zřejmě nemůže, jelikož to by znamenalo, že přímka leží v rovině podstavy, a tedy že i vrchol leží v rovině podstavy, což je případ, který jsme podchytili již v rámci kontroly vstupních dat funkcí *inputCheck()*. Pokud vrátí hodnotu -1, tedy že je přímka rovnoběžná s rovinou, je zřejmé, že další průsečík s kuželem neexistuje a program proto vrací hodnotu false. Třetí hodnotu, kterou může *intersection()* vrátit, je 1: přímka je s rovinou podstavy různoběžná a našli jsme jejich průsečík. Můžeme nyní ověřit, zda přímka prochází kuželem, což uděláme tak, že porovnáme vzdálenost nalezeného průsečíku od středu *S* podstavy. Jestliže je vzdálenost bodů menší nebo rovna poloměru podstavy,



našli jsme druhý průsečík přímky s kuželem. Můžeme navázat funkcí *comparison()*, která ověří, zda body *A* a *B* leží mezi nalezenými průsečíky. Odtud dostáváme výsledek, zda úsečka leží uvnitř kuželu, či nikoliv. Pakliže je vzdálenost středu od průsečíku větší, než poloměr, je zřejmé, že bez dalších výpočtů víme, že přímka, a tedy i zadaná úsečka, leží mimo kužel.



Obr. 35: Možné polohy přímky pro A,B,V ležící na jedné přímce

B) Body A,B,V tvoří vrcholovou rovinu

V tomto případě můžeme postupovat tak, jak bylo zmiňováno v kapitole 4. Sestrojíme pomocný bod *R* na přímce *p*. Dále stejně jako v předešlém případě hledáme průsečík přímky p s rovinou *XY*. Tentokrát mohou nastat všechny tři případy hodnot $pos \in \{-1,0,1\}$, protože přímka p neprochází vrcholem *V*, a proto může ležet i v rovině podstavy. V předchozím případě jsme měli jeden průsečík s kuželem daný a pro zjištění existence, případně polohy druhého nám stačila funkce *distancePP()* pro výpočet vzdálenosti dvou bodů. Tato podúloha je však nepatrně složitější a vyžaduje funkci pro výpočet vzdálenosti přímky od bodu a průsečíku přímky s kružnicí. Než tedy budeme pokračovat v popisu řešení, tyto funkce si krátce představíme.

5.2.5.3 Funkce pro výpočet vzdálenosti přímky od bodu distanceLP()

Argumenty funkce jsou tři body. První dva jsou body (L1, L2) definující přímku, třetí je bod *S*, jehož vzdálenost od této přímky chceme zjistit. Funkce vrací hodnotu double, představující hledanou vzdálenost. Výpočet probíhá tak, že ze zadaných bodů přímky vypočteme směrový vektor u. Dále hledáme bod *X* na přímce, jehož směrový vektor $\overrightarrow{XS}$ je kolmý na přímku. Bod *X* můžeme vyjádřit pomocí parametrické rovnice přímky: $p_i = L_i + t * u_i$. Využijeme vlastnosti, že dva na sebe kolmé vektory mají skalární součin roven nule: $(S - X) * u = 0$. Do rovnice dosadíme bod *X* z parametrického vyjádření přímky a hledáme hodnotu parametru t:

$$(S_x - (L_x + t * u_x)) * u_x + (S_y - (L_y + t * u_y)) * u_y + (S_z - (L_z + t * u_z)) * u_z = 0$$

odtud:

$$t = \frac{(S_x - L_x) * u_x + (S_y - L_y) * u_y + (S_z - L_z) * u_z}{u_x^2 + u_y^2 + u_z^2}.$$

Dosazením t do parametrické rovnice přímky dostáváme bod *X*. Vzdálenost |*XS*| poté spočítáme pomocí funkce *distancePP()*.

5.2.5.4 Funkce pro výpočet průsečíku přímky s kružnicí interLR()

Vstupními parametry funkce jsou body *A*, *B*, do nichž v průběhu funkce uložíme nalezené průsečíky, střed kružnice *S* a poloměr *r*. Výstup je prázdný, jelikož požadované hodnoty ukládáme rovnou do vstupně výstupních proměnných. Prvním krokem je výpočet směrového vektoru u, čímž získáme parametrické vyjádření přímky: $p_i = A_i + t * u_i$. Jak si zmíníme ještě později, zadaná přímka a kružnice jsou v rovině *XY*, resp. rovině s ní rovnoběžné, proto můžeme zanedbat z-ovou souřadnici. Toto vyjádření dosadíme za x a y do rovnice kružnice:

$$(x - S_x)^2 + (y - S_y)^2 = r^2$$

Dosazení:

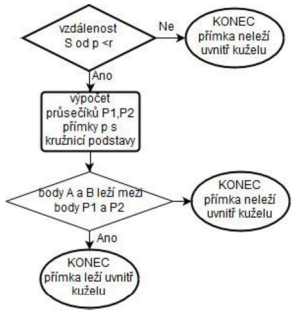
$$(A_x + t * u_x - S_x)^2 + (A_y + t * u_y - S_y)^2 = r^2$$

rovnici převedeme na kvadratickou rovnici tvaru $a * t^2 + b * t + c = 0$, přičemž hodnoty koeficientů a, b, c:

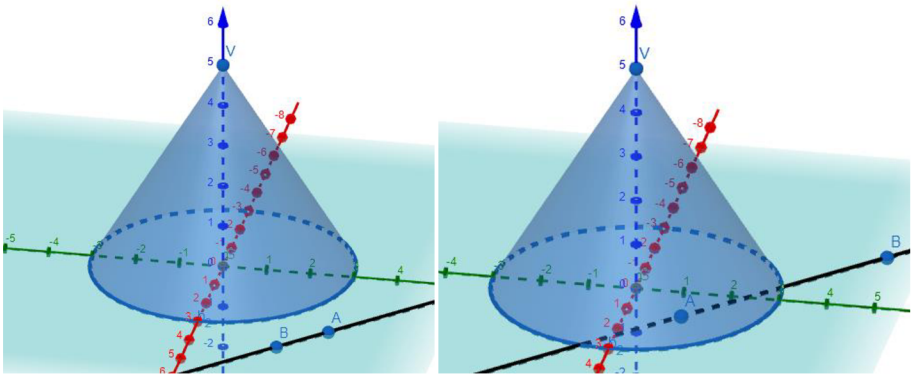
$$a = u_x^2 + u_y^2$$
$$b = 2 * (A_x - S_x) * u_x + 2 * (A_y - S_y) * u_z$$
$$c = (A_x - S_x)^2 + (A_y - S_y)^2 - r^2$$

uložíme do proměnné typu *coord* coef. Dále pomocí diskriminantu vypočteme dvě hodnoty parametru *t1* a *t2*. Dosazením parametrů do původní parametrické rovnice přímky získáme dva průsečíky s kružnicí, které ukládáme namísto původních vstupních hodnot *A*, *B*. Poznamenejme ještě, že funkce počítá pouze s řešením, kdy přímka protíná kružnici ve dvou bodech. Případy, kdy je přímka tečnou ke kružnici nebo ji zcela míjí, jsou podchyceny před voláním funkce *interLR()* pomocí výpočtu vzdálenosti přímky od bodu *distanceLP()*.

B1) Přímka v rovině podstavy



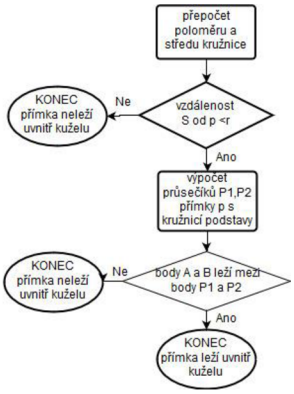
Nyní se můžeme vrátit k možnostem, které mohou nastat při hledání průsečíku přímky *p* s rovinou *XY*. Nejjednodušším případem bude když přímka bude ležet v rovině podstavy (par=0) (Obr. 36). Pak počítáme funkcí *distanceLP()* vzdálenost přímky od středu podstavy. Pokud je tato vzdálenost větší, nebo rovna poloměru podstavy, pak úsečka *AB* leží mimo kužel. V opačném případě počítáme pomocí funkce *interLR()* její průsečíky s kružnicí a na závěr porovnáme, zda úsečka *AB* leží mezi nalezenými průsečíky. Pokud ano, pak celé robotické rameno leží v povoleném prostoru, v opačném případě končí cyklus s hodnotou false.

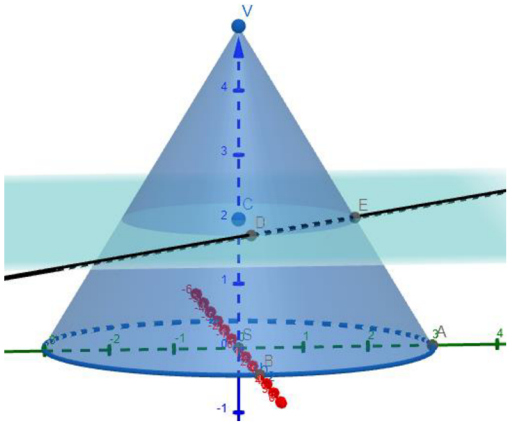


Obr. 36: Přímka p v rovně podstavy

B2) Přímka rovnoběžná s rovinou podstavy

V případě že zadaná přímka je rovnoběžná s rovinou podstavy, je třeba provést řez kuželem na úrovni přímky. Tímto nám vznikne nová podstava, u níž musíme dopočítat nový střed a poloměr (Obr. 37).





Obr. 37: Přímka rovnoběžná s rovinou podstavy

Začneme výpočtem nového středu *Sn*, který musí ležet mezi původním středem *S* a vrcholem *V*, vznikne tedy posunutím bodu *S* o k-násobek jednotkového směrového vektoru $\overrightarrow{SV}$, který si označíme jako u:

$$S_n = S + k * u$$

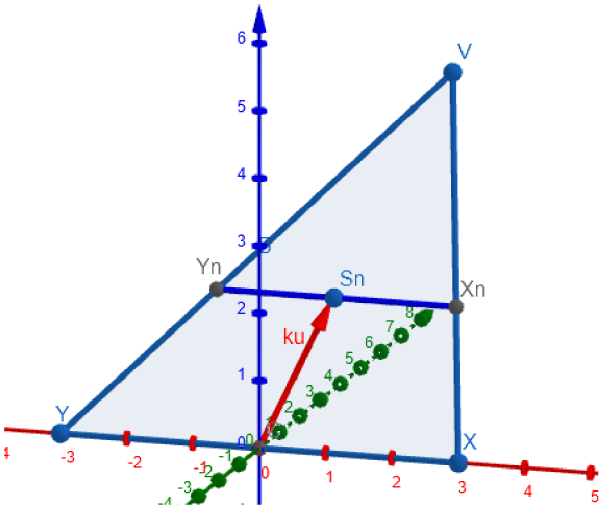
Koeficient k zjistíme ze znalosti velikosti posunutí v ose *z*, které odpovídá z-ové souřadnici bodu *A*, resp. *B*. Dále máme původní *S* v počátku souřadnic, tedy jej při výpočtu koeficientu k nemusíme uvažovat. Dostáváme tedy:

$$k = \frac{A_z}{u_z}$$

Dosazením dostaneme souřadnice nového středu. Velikost nového poloměru nyní můžeme spočítat pomocí podobnosti trojúhelníků, viz (Obr. 38):

$$r_n = \frac{r * |S_n V|}{|SV|}$$

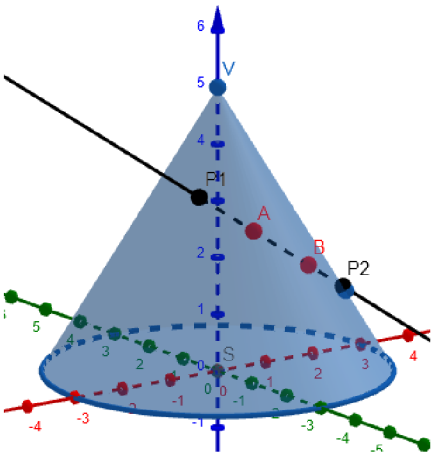
Nyní, když máme potřebné hodnoty S_n a r_n, stačí vypočítat vzdálenost přímky p od nového středu S_n a pokud je tato vzdálenost $<r_n$, najít průsečíky této přímky a nové kružnice. Následuje porovnání, jestli body A, B leží mezi nalezenými průsečíky. Pokud ano, pak se celé rameno nachází v povoleném kuželovém prostoru. Jakákoliv jiná možnost znamená, chybu.



Obr. 38: Přepočet posunutého středu a poloměru kružnice

B3) Přímka různoběžná s rovinou podstavy

Pokud je přímka p různoběžná s rovinou podstavy (par=1) (Obr. 39), pak kromě průsečíku přímky p s rovinou podstavy, hledáme i průsečík přímky q, procházející body V a R, s rovinou XY. Vzniklý průsečík reprezentuje půdorysný stopník a pokud jím a průsečíkem přímky p s podstavou proložíme přímku r, pak tato přímka odpovídá průmětu přímky p do roviny XY. Stejně jako v předchozích případech, na základě vzdálenosti r od středu podstavy S určíme, zda má smysl hledat průsečíky s kružnicí podstavy. Pokud ne, cyklus standardně končí, avšak pokud jsou průsečíky nalezeny, výpočet dále pokračuje. Tyto průsečíky jsou pouze průměty skutečných průsečíků do roviny podstavy. Skutečné průsečíky nalezneme tak, že sestavíme 2 přímky dané jedním z průsečíků a vrcholem V. Pak jsou průsečíky těchto přímek s přímkou p hledanými průsečíky přímky p s kuželem. Průsečíky spočítáme pomocí funkce *interLL()*, kterou jsme si vysvětlili již v části o zmenšování kuželu. Opět následuje porovnání *comparison()*, pro zjištění, zda body A, B leží mezi nalezenými průsečíky.



Obr. 39: Přímka různoběžná s rovinou podstavy

Odtud následuje další cyklus, začínající načtením nové úsečky s poloměrem.

6 ZHODNOCENÍ A DISKUZE

Výše popsané algoritmy byly úspěšně testovány pro různě zadané kužely a komolé kužely. Nicméně takovéto testování je velmi náročné, jelikož každý testovaný kužel musel být pro kontrolu zadán do grafického softwaru (grafická kalkulačka Geogebra), je tedy možné, že některý specifický případ při testování unikl. Algoritmus však bude dále převeden do softwaru Automation Studio firmy B&R Automation, který má vizualizační prostředím napojené přímo na vstupní a výstupní hodnoty algoritmu. Zde pak mohou být algoritmy objemněji testovány a případné chyby odladěny.

Testování algoritmů proběhlo na běžném počítači s procesorem Intel řady i5 (i5-201M, 3,6GHz). Dosažené časy lze považovat za objektivní vzhledem k tomu, že industriální počítače, které řídí robotické aplikace, jsou vybaveny rovněž procesory Intel řady i3, i5 či i7. Všechny tyto procesory mají výkon srovnatelný s procesorem použitým pro testování. Běžná délka cyklu běhu robotické knihovny je $\sim 4ms$. Naměřené délky jednoho cyklu algoritmu pro kužel: $\sim 50\mu s$ a pro komolý kužel $\sim 250\mu s$. Což jsou však stále akceptovatelné hodnoty i pro nejméně výkonné PLC.

7 ZÁVĚR

Práce byla zaměřena na problematiku monitorování pracovních prostorů robota. V teoretické části bylo hlavním cílem shrnout používané postupy a algoritmy monitorování pracovních prostorů.

V první kapitole jsme si představili, co máme pod pojmem pracovní prostor na mysli. Dále jsme si představili základní algoritmy pro detekce kolizí v prostoru. Zmínili jsme voxelové mřížky, oktalové stromy, binární dělení prostoru a konstruktivní geometrii pevných těles. Následně jsme si představili přístup k detekci kolizí ve firmě B&R Automation.

Druhá kapitola již byla teoretickou přípravou k implementaci algoritmů, kdy jsme si představili nepřesnosti výpočtů spojené s užíváním proměnných s plovoucí desetinnou čárkou. Řekli jsme si, jak měřit relativní a absolutní chybu a ukázali jsme si, jak tyto hodnoty použít při porovnávání dvou čísel s plovoucí desetinnou čárkou.

Poslední teoretická kapitola představila základní teorii kuželů. Rozdělili jsme si možné reprezentace kuželů na analyticky zadané kužely a kužely jako kvadratické plochy. Dále jsme si zmínili, jak by u každé reprezentace vypadalo hledání průsečíků s přímkou, a zhodnotili, která reprezentace je pro náš algoritmus výhodnější.

V praktické části jsme popsali dva implementované algoritmy pro určení, zda část robotického ramene, určená dvěma body a poloměrem, leží v pracovním prostoru tvaru kuželu nebo komolého kuželu. Algoritmy jsme popisovali současně, jelikož rozdíly mezi nimi jsou minimální.

Při popisu jednotlivých implementací jsme si nejdřív představili obecné funkce, které byly potřeba k různým výpočtům při běhu algoritmu. Jednalo se například o porovnávání hodnot typu double, výpočet normálových vektorů, vzdálenost dvou bodů nebo funkce pro výpočet rovnice roviny.

V hlavní části jsme se zaměřili na popis hlavních kroků algoritmů. Nejdříve jsme vysvětlili, jak je zadáván kužel a jak je kontrolována správnost vložených dat. Následovala transformace kuželu tak, aby jeho podstava ležela v rovině XY, což usnadní další výpočty. Dále je popsaná cyklická část programu, kdy jsou načítána jednotlivá ramena robota, která jsou nejdříve zkontrolována. Poté jsou body transformovány, aby jejich poloha odpovídala nové poloze kuželu. Kužel je zmenšen o poloměr části robotického ramene a je vypočtena vzájemná poloha zadané úsečky a kuželu.

8 SEZNAM POUŽITÉ LITERATURY

[1] FUNKHOUSER, Thomas. Solid Modeling. In: *Princeton University* [online]. 2000 [cit. 2018-05-27]. Dostupné z: https://www.cs.princeton.edu/courses/archive/fall00/cs426/lectures/solid/sld001.htm

[2] STRACHOTA, Pavel. *Modelování těles* [online]. In: . [cit. 2018-05-27]. Dostupné z: http://saint-paul.fjfi.cvut.cz/base/sites/default/files/POGR/POGR2/05.modelovani_teles.pdf

[3] NEVALA, Eric. Introduction to Octrees. In: *GameDev* [online]. 20.ledna 2014 [cit. 2018-05-27]. Dostupné z: https://www.gamedev.net/articles/programming/general-and-gameplay-programming/introduction-to-octrees-r3529/

[4] DOUG, James. Spatial Data Structures. In: *Carnegie Mellon University: School of computer science* [online]. 6.11.2013 [cit. 2018-05-27]. Dostupné z: http://www.cs.cmu.edu/~djames/15-462/Fall03/notes/19-spatial.pdf

[5] Constructive Solid Geometry CSG. In: *Netgen/NGSolve* [online]. [cit. 2018-05-27]. Dostupné z: https://ngsolve.org/docu/nightly/netgen_tutorials/define_3d_geometries.html

[6] HEGDE, Shriram. *Solid Modeling Techniques: Constructive Solid Geometry (CSG)* [online]. In: . [cit. 2018-05-27]. Dostupné z: http://web.iitd.ac.in/~hegde/cad/lecture/L32_solidmcsg.pdf

[7] Scientific Notation. In: *Department of Chemistry Texas A&M University* [online]. [cit. 2018-05-27]. Dostupné z: https://www.chem.tamu.edu/class/fyp/mathrev/mr-scnot.htm

[8] Significant Figures. In: *Department of Chemistry Texas A&M University* [online]. [cit. 2018-05-27]. Dostupné z: https://www.chem.tamu.edu/class/fyp/mathrev/mr-sigfg.html

[9] NULL, Linda. a Julia. LOBUR. *The essentials of computer organization and architecture* [online]. Sudbury, Mass.: Jones and Bartlett Publishers, c2003 [cit. 2018-05-27]. ISBN 07-637-0444-X.

[10] KEARFOTT, Ralph Baker. *The IEEE 754-2008 Floating Point Standard and its Pending Revision* [online]. In: . [cit. 2018-05-27]. Dostupné z: https://interval.louisiana.edu/preprints/2015-Fall-UL-Applied-Math-Seminar-754.pdf

[11] GOLDBERG, David. *What Every Computer Scientist Should Know About Floating-Point Arithmetic* [online]. In: . březen 1991 [cit. 2018-05-27]. Dostupné z: https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html#799

[12] *Rounding Errors* [online]. In: . [cit. 2018-05-27]. Dostupné z: http://floating-point-gui.de/errors/rounding/

[13] An introduction to different rounding algorithms. In: *EETimes* [online]. 4.1.2006 [cit. 2018-05-27]. Dostupné z: https://www.eetimes.com/document.asp?doc_id=1274485

[14] DAWSON, Bruce. Comparing Floating Point Numbers, 2012 Edition. In: *Random ASCII* [online]. 25.2.2012 [cit. 2018-05-27]. Dostupné z:

https://randomascii.wordpress.com/2012/02/25/comparing-floating-point-numbers-2012-edition/

[15] HAŠEK, Roman a Pavel PECH. *Kvadratické plochy a jejich prezentace v programu Maple*. V Českých Budějovicích: Jihočeská univerzita, 2010. ISBN 978-80-7394-271-7.

9 SEZNAM PŘÍLOH

CD