En Simpel Robotsimulator

Rasmus Friis Kjeldsen
razzle@diku.dk

Gunni Rode
delerium@diku.dk

Vejleder – Jens Damgaard Andersen
jda@diku.dk

1. november 2004

Indhold

Figurer

1 Indledning

Dette projekt omhandler simulatorer, der kan simulere *bil-agtige* robotter med et begrænset sæt af fysiske egenskaber i planen[1]. Vi ønsker, at en simuleret robot skal kunne løse følgende opgave, som er delt i to faser:

1. Find den korteste rute mellem to positioner A og B under forudsætning af, at ruten ikke indeholder nogen forhindringer, og at robotten kun kan bevæge sig i fremadrettet retning.
2. Følg den fundne rute bedst muligt.

En *position* defineres her som triplen bestående af de cartetiske koordinater i planen samt robottens orientering, dvs. (x, y, θ).

Vi opstiller en præcis matematisk model for en sådan robot, og hvorledes den kan simuleres. Vi redegør for, hvordan den korteste rute mellem to positioner i planen findes effektivt og beskriver, hvordan vores udviklede simulationsframework kan simulere en mængde af sådanne robotter, som kan finde og følge en optimal rute. Selve simulationsframeworket indeholder definitioner for forskellige brugergrænseflader, og vi har implementeret to simulatorapplikationer: Én der kan afvikles fra en konsol, og én med en komplet grafisk brugerflade, der bl.a. tillader animation af simulationen, mens den afvikles. Endvidere har vi succesfuldt implementeret dele af programkoden for en simuleret robot på en fysisk robot.

1.1 Motivation

Ved deltagelse i den årlige RoboCup turnering (2004), se [rob, 2004], på Danmarks Tekniske Universitet kunne det observeres, at flere af de bil-agtige robotter valgte overraskende ruter i visse situationer. Det der – for det menneskelige øje – måske så ud som den mest optimale rute, var det ofte ikke alligevel, når karakteristika for omgivelserne og robotten blev taget med i betragtning.

Et eksempel, der illustrerer dette, er en robot med forhjulsstyring, som skal køre fra p_1 til p_2, hvor p_2 er placeret umiddelbart til højre for p_1; for eksempel $p_1 = (0,\ 0,\ 0)$ og $p_2 = (0,\ 0.1,\ \pi)$. Med mindre robotten er meget lille, eller kan dreje meget skarpt, vil et enkelt højresving ikke kunne bringe robotten fra p_1 til p_2. En strategi kunne være først at dreje en smule til venstre, efterfulgt af et længere højresving,

[1] Se afsnit 2.1 på side 5 for en definition og matematisk model af en bil-agtig robot som anvendt i dette projekt.

og afslutte med et kort venstresving. Dette giver sammenlagt en større drejeradius, men samtidigt en længere rute. Ved RoboCup 2004 var netop et lignende tilfælde aktuelt. Problemet blev løst empirisk, dvs. ved gentagen *trial and error*, hvilket var en temmelig tidskrævende proces.

Det er dog langt fra optimalt at udvikle og afprøve strategier direkte på en robot. Udviklingstiden er længere end normalt: Programkoden skal overføres til robotten; der skal ventes på, at robotten rent fysisk udfører programmet; det er sjældent muligt at *debugge* direkte på robotten; der kan være flere end én type robot der udvikles til; osv. Det kan endog være meget svært at fejlfinde en løsning på en fysisk robot, også fordi der for eksempel kun er begrænsede skærmstørrelser på robotten til rådighed.

1.2 Mål

Vores primære mål er at udvikle en simulator, der kan simulere en en simpel *non-holonomic*, autonom (bil-agtig) robot. En non-holonomic robot er begrænset i sine bevægeretninger, idet der i et non-holonomic system er begrænsninger på et objekts hastigheder i planen. For vores robot betyder det, at den kun kan køre forlæns/baglæns, ikke sidelæns. Med andre ord kan en non-holonomic robot ikke følge vilkårlige kurver i planen. Robotten skal kunne finde den afstandsmæssigt korteste rute mellem to punkter, inspireret af [Reeds and Shepp, 1990], og herefter forsøge at følge den fundne rute optimalt, under forudsætning af at dens fysiske begrænsninger overholdes, såsom for eksempel hvor hurtigt den kan dreje.

Det er endvidere vigtigt, at simulatoren er nem at bruge, både bruger- og udviklingsmæssigt, idet vi forestiller os, at der skal udvikles en mængde forskellige simulationsscenarier – hvilket jo er pointen med en simulator. Således må både robotten og simulatoren kunne konfigureres. Brugermæssigt vil det være optimalt, hvis simulatoren kan tegne robotten på dens færd, så det også grafisk kan illustreres, hvor god en given bil-agtig robot er til at løse sin opgave. Det vil intuitivt give en bedre forståelse for, hvordan robotten opfører sig sammenlignet med kun at betragte simulationsdata. Simulatoren skal i sagens natur altså gøre det lettere at udvikle og fejlfinde løsninger, som senere vil kunne bruges på en virkelig robot.

Vi definerer endvidere en bil-agtig robot som altid havende følgende egenskaber:

- Fire hjul;
- En motor til fremdrift;
- Forhjulsstrying, baghjulenes vinkel er fast;

- Tachometer (“omdrejningstæller”) monteret på hvert baghjul;
- Skal kunne fungere uden (simuleret) Global Positioning System (GPS).

Disse egenskaber svarer til en delmængde af de egenskaber, som den robot, den ene af forfatterne til denne rapport deltog med i RoboCup 2004. Denne robot vil i resten af rapporten blive refereret til som “Murphy”; dens karakteristika kan ses i appendiks A. Ideen på længere sigt er, at implementere programkode udviklet på vores simulerede robotter på denne eller andre fysiske robotter.

Idet vi vælger at kigge på den optimale rute i afstandsmæssig forstand, må robotten kunne bestemme sin position. Ved at kende den tilbagelagte afstand for hvert baghjul kan dette lade sig gøre.

For at opnå vores primære mål defineres nu følgende delmål:

1. At opstille og implementere en matematisk model for robottens bevægelse i planen;
2. At finde og implementere en algoritme, som løser opgaven med at finde den korteste afstand mellem to positioner under hensyntagen til modellen;
3. At finde og implementere mindst en algoritme til at følge en valgt rute, optimal eller ej;
4. At designe og implementere en simulator, der ud fra delmål 1-3, kan afvikle simulationer af forskellige konfigurationer af bil-agtige robotter. Det ønskes endvidere, at simulatoren som minimum kan rapportere, hvor godt robotten løser sin opgave i form af afvigelse mellem robottens virkelige og estimerede position. Endeligt er det ønskværdigt, at simulatoren skal kunne animere robotten mens den løser sin opgave med at følge en rute.

1.3 Rapportens opbygning

Vi begynder med at beskrive, hvorledes vi matematisk modellerer en bil-agtig robot i kapitel 2. Umiddelbart herefter følger kapitel 3, hvori det forklares, hvorledes den opstillede model, udtrykt i differentialligninger, løses. Dernæst beskriver kapitel 4, hvordan den korteste rute mellem to positioner i planen findes, samt hvorledes den fundne rute kan følges af en bil-agtig robot; der gennemgås flere metoder, som alle er implementeret. Det næste kapitel, 5, omhandler designet og implementeringen af de ovenfor nævnte fire kapitler, samt selve simulatorene, og om hvordan simulationer afvikles via den udviklede programkode. Kapitel 6 redegør dernæst for, at de

udviklede simulatorer kan afvikle simulationer for bil-agtige robotter, og at de udviklede metoder er særdeles effektive (korrekte) til for eksempel at finde og følge en optimal rute mellem to positioner. Dernæst præsenteres kort forslag til yderligere arbejde i kapitel 7, endeligt efterfulgt af en konklusion i kapitel 8.

Den udviklede programkode samt simulationseksempler er at finde på den vedlagte cd-rom, som beskrevet i næste afsnit.

Engelske fraser og specielle begreber fremtræder i *kursiv tekst* første gang de optræder i teksten. Programkode og filnavne i teksten fremstår altid med `fast bredde`.

Hvor intet andet er nævnt, så er det en simuleret robot med Murphys dimensioner, som bruges til afprøvning.

Det forventes, at læseren er bekendt med OOA og OOP – Objekt Orienteret Analyse og Programmering – samt programmeringssproget Java (1.4.2).

1.4 Indhold af den vedlagte cd

Til denne rapport hører en cd med følgende struktur:

- `/rapport/robotsim.pdf`: Denne rapport.
- `/programkode/*`: Java-kildekoden.
- `/dokumentation/*`: Genereret JavaDoc for kildekoden; startsiden for dokumentationen er `index.html`.
- `/simulator/*`: Indeholder simulatorapplikationerne. Under Windows kan `shell_simulator.bat` og `swing_simulator.bat` køres; på Unix/Linux maskiner kan `shell_simulator.sh` og `swing_simulator.sh` køres. Alternativt kan en simulator startes direkte i dette bibliotek med kommandoen `java dk.diku.robotsim.simulator.[Shell|Swing]Simulator`. Det kræves i alle tilfælde som minimum, at Java JRE 1.4.2 er installeret, for at simulatoren kan køre (se eventuelt afsnit 5).
- `/simulator/resources/*`: Eksempler på input til simulatorer og simulationer.
- `/simulationer/*`: Eksempler på output genereret af simulatorene.
- `/artikler/*`: Kopier i pdf- eller postscriptformat af de fleste af de artikler, der refereres til i teksten.

2 Modellering

Dette kapitel omhandler den matematiske modellering af en robot til brug i simulationen. Kapitlet indledes med en formel definition af en bil-agtig robot, som er den type robot, vi beskæftiger os med, samt identificering af en bil-agtig robots fysiske begrænsninger.

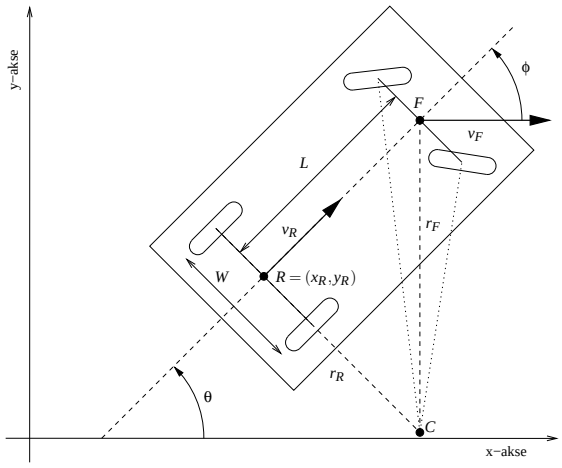
Som angivet i [Meriam and Kraige, 1998] ifølge [Kongsbak et al., 2002, side 15] er en brugbar model til simulering af bil-agtig robot en *dynamisk model*. En dynamisk model består af to undermodeller: En *kinetisk* og en *kinematisk* model. Den kinetiske model beskriver energien og kræfterne, der får robotten til at bevæge sig, hvorimod den kinematiske model beskriver robottens orientering og bevægelseskurve. Disse modeller forklares i henholdsvis afsnit 2.2 og 2.3, og en samlet model præsenteres herefter i afsnit 2.4.

For at holde modellen relativt simpel, har vi valgt kun at se på tilfældet, hvor robottens hjul ruller perfekt på underlaget. Herved kan vi undlade at iberegne krafterne mellem hjul og underlag i bevægelsesligningerne. Endvidere modellerer vi robottens motor meget simpelt, så vi alt i alt helt kan undgå at regne med fysiske kræfter i den samlede model. For en modellering af en bil-agtig robot baseret på kræfter mellem en robot og dens omgivelser refereres til [Shiller and Gwo, 1991].

2.1 Matematisk beskrivelse af robottens fysiske begrænsninger

Vi definerer en bil-agtig robot som havende fire hjul. De to bagerste hjul sidder fastgjort på robottens chassis med en akse gennem deres centrum på en måde, så de kun kan rotere om denne akse. De to forreste hjul sidder også fastgjort med en akse gennem deres centrum, men på en måde så de både kan rotere om aksen og omkring en lodret akse. Se figur 1. Modellen på figuren ses at have Ackermannstyring. Ackermannstyring lader forhjulene under et sving følge cirkelbuer med lidt forskellig radius, så forhjulene ruller uden at skride ud, jævnfør [Lund, 2000, side 46]. Forhjulsstyringen er *modelleret* som et enkelt forhjul monteret midt på foraksen. Når robotten skal tegnes, tegnes den som havende to forhjul, som på figuren, og deres vinkler beregnes ud fra det ene modellerede hjuls vinkel ϕ. Et køretøj med et enkelt forhjul og et køretøj med Ackermannstyring vil følge de samme kurvebaner, og derfor er denne modellering ikke mindre generel.

På figuren er R midtpunktet af den bagerste akse og v_R hastigheden af dette punkt. ϕ er styrehjulenes vinkel i forhold til robottens længdeakse. For fast $\phi \neq 0$ kører robotten langs en cirkel med centrum i punktet C. L betegner afstanden mellem for-



Figur 1: Model af en bil-agtig robot. Løst tegnet efter [Sekhavat and Hermosillo, 2001, figur 4].

og bagakserne, og W er afstanden mellem baghjulene. θ er robottens orientering (rotation) i forhold til x-aksen.

Vi antager endvidere, at robotten har en motor til at sikre fremdrift, men antager intet om, hvordan motorkraften overføres til hjulene.

2.2 Kinetisk model

I den kinematiske model i afsnit 2.3 har vi brug for at opstille nogle ligninger, som beskriver, hvorledes robottens aktuatorer påvirker robotten. Som på en almindelig bil, kan robotten styre forhjulenes vinkel, samt styre sin acceleration (svarer til motor/bremser på en almindelig bil).

2.2.1 Motor

For at forenkle ligningerne, har vi valgt en ganske simpel modellering af robottens motor. Vi vælger, at robotten direkte kan styre robottens acceleration i vektoren v_R's retning, jævnfør figur 1:

$$\dot{v}_R = a \tag{1}$$

Her er a så accelerationen, som robotten kan kontrollere. Denne model modellerer slet ikke de interne forhold i en fysisk motor, men den er valgt så simpelt som

muligt, da en avanceret modellering ikke er en del af foremålet med opgaven.

2.2.2 Forhjulsstyring med servo

Modelleringen af forhjulsstyringen er bygget på en modellering af den servomotor, som er brugt på Murphy-robotten til at styre forhjulene. Robotten kan her sende en vinkel til servomotoren, og den vil herefter dreje forhjulene til den ønskede vinkel, betegnet ϕ_w, og derefter aktivt holde vinklen. Vi vælger at modellere drejehastigheden, betegnet v_ϕ, som konstant, dvs. at ignorere start- og slutacceleration:

$$\dot{\phi} = \begin{cases} v_\phi & \text{hvis } \phi < \phi_w \\ -v_\phi & \text{hvis } \phi > \phi_w \\ 0 & \text{hvis } \phi = \phi_w \end{cases} \tag{2}$$

2.3 Kinematisk model

En kinematisk model af robotten på figur 1 kan udledes som beskrevet i det følgende. Det forudsættes, at hjulene på robotten ruller på underlaget uden at skride ud.

Ud fra figur 1 kan drejeradiussen r_R for midtpunktet R af bagaksen udledes ved trekantsbetragtning:

$$\tan\phi = \frac{L}{r_R} \quad \Rightarrow \quad r_R = \frac{L}{\tan\phi} \tag{3}$$

Den tidsafledte af θ kan nu findes. For en cirkel med radius r_R gælder, at vinklen er proportional med buelængden l som:

$$\theta = \frac{l}{r_R} \tag{4}$$

Der gælder så også, at ændringen af vinklen er proportional med ændringen af længden:

$$\Delta\theta = \frac{\Delta l}{r_R} \tag{5}$$

Sker denne ændring i tidsrummet Δt haves:

$$\frac{\Delta\theta}{\Delta t} = \frac{\Delta l}{\Delta t}\frac{1}{r_R} = \frac{\Delta l}{\Delta t}\frac{\tan\phi}{L} \tag{6}$$

I grænsen for $\Delta t \to 0$ haves:

$$\dot{\theta} = \dot{l}\frac{\tan\phi}{L} \tag{7}$$

Da $\dot{l}$ er hastigheden af punktet R i tangentretningen for cirklen med radius r_R, er $\dot{l} = v_R$, jævnfør definitionen på v_R på figur 1. Altså haves:

$$\dot{\theta} = v_R\frac{\tan\phi}{L} \tag{8}$$

Bevægelsesligningerne for $\dot{R} = (\dot{x}_R, \dot{y}_R)$ findes let ud fra θ, da de blot er v_R projekteret ind på x- og y-aksen:

$$\dot{x}_R = v_R\cos\theta \tag{9}$$
$$\dot{y}_R = v_R\sin\theta \tag{10}$$

I alt er den kinematiske model givet ved følgende differentialligninger, jævnfør de i [Sekhavat and Hermosillo, 2001, formel 6] postulerede, men ikke udledte, ligninger:

$$\dot{x}_R = v_R\cos\theta \tag{11}$$
$$\dot{y}_R = v_R\sin\theta \tag{12}$$
$$\dot{\theta} = v_R\frac{\tan\phi}{l} \tag{13}$$

Da en bil-agtig robots konstruktion typisk vil begrænse styrehjulenes hjulvinkel ϕ, vil vi her kun beskæftige os med tilfældet, hvor $0 \leq |\phi| \leq \phi_{max} < \frac{\pi}{2}$, og hvor ϕ_{max} er styrehjulenes maksimale vinkel. Den ovenstående model gælder naturligvis også kun for $-\frac{\pi}{2} < \phi < \frac{\pi}{2}$, idet $\tan\frac{\pi}{2} = \pm\infty$.

Denne begrænsning af styrehjulenes maksimale vinkel medfører naturligvis, at robotten også har en minimum drejeradius, jævnfør formel 3:

$$\rho_{min} = \frac{L}{\tan\phi_{max}} \tag{14}$$

2.3.1 Modellerede sensorer

Da robotten også skal simulere to tachometre, et på hvert baghjul som nævnt i afsnit 1.2, må modellen også beregne, hvor langt hvert baghjul til ethvert tidspunkt har kørt. Først modelleres, hvor langt punktet R har bevæget sig:

$$\dot{d}_R = |v_R| \tag{15}$$

For at beregne hvor langt et baghjul har kørt, beregnes først hvor langt punktet, som baghjulets centrum befinder sig i, har bevæget sig. Hvis robotten blot kører ligeud ($\phi = 0$), gælder så:

$$\dot{d}_{left} = \dot{d}_R \tag{16}$$
$$\dot{d}_{right} = \dot{d}_R \tag{17}$$

Hvis robotten ikke kører ligeud ($\phi \neq 0$), så må $\dot{d}_{left}$ og $\dot{d}_{right}$ beregnes ud fra $\dot{x}$ og $\dot{y}$:

$$\dot{d}_{left} = \sqrt{\left(\dot{x}\frac{r_{left}}{r_R}\right)^2 + \left(\dot{y}\frac{r_{left}}{r_R}\right)^2} \tag{18}$$
$$\dot{d}_{right} = \sqrt{\left(\dot{x}\frac{r_{right}}{r_R}\right)^2 + \left(\dot{y}\frac{r_{right}}{r_R}\right)^2} \tag{19}$$

Her er r_R drejeradiussen ved hjulvinklen ϕ udregnet efter formel 3, og r_{left} og r_{right} er drejeradiussen for henholdsvis det venstre og det højre hjul, jævnfør figur 8. Disse kan udregnes som:

$$r_{short} = r_R - \frac{W}{2} \tag{20}$$
$$r_{long} = r_R + \frac{W}{2} \tag{21}$$
$$r_{left} = \begin{cases} r_{long} & \text{hvis } \phi < 0 \\ r_{short} & \text{hvis } \phi > 0 \end{cases} \tag{22}$$
$$r_{right} = \begin{cases} r_{short} & \text{hvis } \phi < 0 \\ r_{long} & \text{hvis } \phi > 0 \end{cases} \tag{23}$$

I det ovenstående betegner W afstanden mellem baghjulene.

Når nu d_{left} og d_{right} kan beregnes, kan tachometerne simuleres. Hvert tachometer giver et heltal, q, ud fra antallet af omdrejninger et hjul har foretaget, samt ud fra tachometerets opløsning q_{res} (ticks pr. omdrejning). For hvert hjul kan antallet af omdrejninger, o, udregnes ud fra, hvor langt hjulets centrum har bevæget sig, repræsenteret ved d i formlen svarende til henholdsvis d_{left} og d_{right}, samt hjulets radius, r_{wheel}:

$$o = \frac{d}{2\pi r_{wheel}} \tag{24}$$

Ud fra o og q_{res} kan tachometerets tilsvarende værdi q beregnes:

$$q = \lfloor o q_{res} \rfloor \tag{25}$$

2.4 Samlet model

Den samlede model af robotten består af den kinetiske og den kinematiske model:

$$\dot{v}_R = a \tag{26}$$

$$\dot{\phi} = \begin{cases} v_\phi & \text{hvis } \phi < \phi_w \\ -v_\phi & \text{hvis } \phi > \phi_w \\ 0 & \text{hvis } \phi = \phi_w \end{cases} \tag{27}$$

$$\dot{\theta} = v_R \frac{\tan\phi}{l} \tag{28}$$

$$\dot{x}_R = v_R \cos\theta \tag{29}$$

$$\dot{y}_R = v_R \sin\theta \tag{30}$$

$$\dot{d}_R = |v_R| \tag{31}$$

$$\dot{d}_{left} = \begin{cases} \sqrt{\left(\dot{x}\frac{r_{left}}{r_R}\right)^2 + \left(\dot{y}\frac{r_{left}}{r_R}\right)^2} & \text{hvis } \phi \neq 0 \\ \dot{d}_R & \text{hvis } \phi = 0 \end{cases} \tag{32}$$

$$\dot{d}_{right} = \begin{cases} \sqrt{\left(\dot{x}\frac{r_{right}}{r_R}\right)^2 + \left(\dot{y}\frac{r_{right}}{r_R}\right)^2} & \text{hvis } \phi \neq 0 \\ \dot{d}_R & \text{hvis } \phi = 0 \end{cases} \tag{33}$$

I det overstående er de frie variable, som robotten direkte kan ændre: Accelerationen a og den ønskede vinkel ϕ_w.

3 Simulator

Den i kapitel 2 opstillede model af robotten består af en række differentialligninger. For at kunne anvende modellen, må disse ligninger løses. Dette kan gøres eksakt analytisk, eller også kan en tilnærmet løsning findes, for eksempel ved numerisk løsning af modellen. En analytisk løsning af modellen ikke er mulig, da robottens *controller* (dvs. det program, som kører på den simulerede robot) påvirker både robotmotorens acceleration og styrehjulenes vinkelacceleration på en måde, som det ikke er muligt at opstille analytiske formler for. Vi vil derfor søge en numerisk løsning.

3.1 Numerisk løsning af modellen

Der findes en række former for numeriske løsninger af differentialligninger, som giver mere eller mindre præcise tilnærmelser til det rigtige resultat. Vi har valgt, at gøre den anvendte numeriske løsningsmetode uafhængig af den implementerede model, så løsningsmetoden kan vælges på køretidspunktet. De implementerede løsningsmetoder, samt deres fordele og ulemper, er kort beskrevet herunder. De udledes ikke.

Afsnit 5.3 på side 54 beskriver implementationen af nedenstående løsningsmetoder.

3.1.1 Eulers metode

Eulers metode er nok den lettest forståelige metode, og den er tilsvarende let at implementere, hvilket gør den velegnet under udviklingen af programmet. Til gengæld er den ikke særlig præcis.

Eulers metode, jævnfør [Burden and Faires, 1997, side 259-261], giver en numerisk approksimation af differentialligningen:

$$\frac{dy}{dt} = f(t,y),\ a \leq t \leq b,\ y(a) = \alpha$$

Her er α starttilstanden, og t er tiden.

Metoden er iterativ, og for hver iteration fremskrives resultatet af sidste iteration med en konstant h, kaldet skridtstørrelsen, multipliceret med værdien af funktionen f i sidste tidsskridt:

$$w_0 = \alpha \tag{34}$$
$$w_{i+1} = w_i + hf(t_i, w_i), \quad i = 0, 1, ... \tag{35}$$

3.1.2 Runge-Kutta metoder

Runge-Kutta-metoderne er en familie af iterative metoder, som er noget mere regnetunge end Eulers metode, men som til gengæld kan tilbyde en meget større nøjagtighed. Vi vil ikke udlede de generelle formler her, men blot angive den metode, vi har anvendt i simuleringen. For mere om Runge-Kutta-metoderne, se [Burden and Faires, 1997, side 276-286] og [Ljung and Glad, 1991, p. 312-313].

Metoden, vi har anvendt i simulationen, er en Runge-Kutta-metode af orden 2, jævnfør [Burden and Faires, 1997, side 276-286]:

$$w_0 = \alpha \tag{36}$$
$$k_1 = f(t_i, w_i) \tag{37}$$
$$k_2 = f(t_i + \frac{h}{2}, w_i + h\frac{k_1}{2}) \tag{38}$$
$$w_{i+1} = w_i + hk_2, \quad i = 0, 1, ... \tag{39}$$

Igen er konstanten h skridtstørrelsen.

4 Robotten

Anvendelsen af den udviklede robotsimulation er beskrevet i dette kapitel. Som nævnt i indledningen vil vi gerne finde en algoritme, som kan få en bil-agtig robot til at køre fra en start- til en slutposition, og helst på en, i en vis henseende, optimal måde. Dette er i litteraturen bedre kendt som *position-til-position bevægelsesproblemet*.

4.1 Position-til-position bevægelsesproblemet

Dette kapitel omhandler løsningen af *The Point-to-Point Motion Task* som beskrevet i [Luca et al., 1999, side 172] – eller med andre ord position-til-position bevægelsesproblemet for den bestemte klasse af bil-agtige robotter som er illustreret på figur 1. Opgaven i dette problem er at finde en algoritme, som kan bevæge en robot fra en startposition p_1 til en slutposition p_2. Vi definerer en *position* som triplen af de cartetiske koordinater (x,y) og robottens orientering θ, dvs. (x,y,θ). Vi betragter kun bevægelser i planen.

Den første halvdel af problemet er at finde en *rute* fra p_1 til p_2, som det er fysisk muligt for robotten at følge. En sådan rute kan betragtes som en sammenhængende kurve γ. Dette beskrives i afsnit 4.2. I afsnit 4.3 beskrives, hvorledes den korteste rute fra p_1 til p_2 findes.

For at robotten kan følge en rute, skal den kende sin position. Dette problem behandles i afsnit 4.4.

Den anden halvdel af problemet er at finde en algoritme, som kan få robotten til at følge den fundne kurve fra start til slut. Dette vil blive beskrevet i afsnit 4.5.

4.2 Mulige bevægelseskurver en bil-agtig robot kan følge

Ud fra den kinematiske model i afsnit 2.3 kan det ses, at en bil-agtig robot, hvis bevægelsesligninger er beskrevet ved modellen, ikke kan følge enhver kurve i planen. Specielt kan det ses, at kun kurver med en begrænset krumning κ_{max}, eller tilsvarende mindste krumningsradius ρ_{min}, kan følges. Er ϕ_{max} styrehjulenes maksimale vinkel, gælder det jævnfør formel 3 og [Fabricius-Bjerre, 1977, side 21] at:

$$\rho_{min} = \frac{L}{\tan\phi_{max}} \qquad \kappa_{max} = \frac{1}{\rho_{min}} \tag{40}$$

Den løsning på position-til-position bevægelsesproblemet vi søger, må altså ikke resultere i kurver med en mindre krumningsradius end ρ_{min}.

4.3 Den korteste kurve mellem to positioner med begrænset mindste krumningsradius

Vi vil i de følgende afsnit gøre rede for, hvorledes den korteste kurve γ_{min} mellem to punkter med en begrænset mindste krumningsradius ρ_{min} kan findes. [Dubins, 1957] behandler dette problem og viser, at γ_{min} altid kan konstrueres ved hjælp af tre cirkelbuer eller en cirkelbue, en linie og en cirkelbue. Linie- eller buestykkerne skal naturligvis ligge i forlængelse af hinanden. For alle cirkelbuernes vedkommende gælder, at deres radius er givet ved ρ_{min}. Resultatet gælder kun for fremadrettet bevægelse. Tilfældet, hvor robotten tillades at bakke, behandles ikke her, men se [Reeds and Shepp, 1990]. Dette er lidt mere kompliceret, men grundideen er den samme.

Afsnit 4.3.1 herunder giver en introduktion til den i de efterfølgende afsnit anvendte notation. Herefter følger en kort beskrivelse af de efterfølgende afsnit, som er flyttet dertil, da notationen må forklares, før meningen med de efterfølgende afsnit kan forstås.

4.3.1 Notation

Ifølge Lester Dubins [Dubins, 1957], jævnfør [Reeds and Shepp, 1990], kan den afstandsmæssigt korteste rute mellem to punkter[2] beskrives ved et *ord* bestående af tre bogstaver fra alfabetet $\{l, r, s\}$, hvor det kræves, at robotten har en (måske forskellig) orientering i start- og slutpunktet. Et punkt inklusiv orientering kaldes en position. Et ord, der beskriver den optimale rute, vil altid have tre bogstaver, og hvert bogstav beskriver en del af ruten. En sådan rute kan betragtes som en kurve γ, og den optimale rute svarer således til γ_{min}.

De to første bogstaver, l (*"go left"*) og r (*"go right"*) beskriver, at robotten skal følge en cirkelbue med radius ρ_{min} henholdsvis til venstre og til højre, dvs. mod uret og med uret. Det sidste bogstav, s (*"go straight"*), beskriver en lige linie mellem to punkter, som robotten skal følge. Ordet lsr, for eksempel, beskriver da, at robotten skal starte med at køre (forlæns) mod uret, derefter køre ligeud, for til sidst at køre med uret. Men da en given rute således er sammensat af tre uafhængige dele, er det nødvendigt at kunne angive, hvor lang en given del er for at kunne identificere én specifik rute; et givet ord repræsenterer derfor en *klasse* af ruter, der dikterer

[2] Engelsk: *geodesic*

en given bevægemåde for robotten. For hver sådan klasse er der netop én optimal rute, men der kan godt være flere klasser, der repræsenterer den korteste vej mellem to positioner, idet der godt kan være flere korteste veje mellem to positioner.

For at identificere én specifik rute ud fra et startpunkt og orientering, tilknyttes længder til de tre dele af ruten, nemlig t, u og v for henholdsvis den første, anden og tredie del; for eksempel $l_t s_u r_v$. Indtil nu har vi antaget, at robotten kun kan køre forlæns. Såfremt det ikke er tilfældet, skal notationen også tage højde for dette (selvom vi ikke beskæftiger os med sådanne situationer). Dette gør den ved at angive et plus (+), hvis robotten skal køre forlæns, og et minus (-), hvis robotten skal køre baglæns. Eksempel: l_t^+ betyder, at robotten skal dreje til venstre, mens den kører t enheder forlæns, hvorimod l_t^- betyder, at den skal dreje til venstre, mens den kører t enheder baglæns. To efterfølgende bogstaver med forskellig retningsangivelse betyder derfor, at robotten på et tidspunkt er nødt til at stoppe.

Endeligt viser Dubins også, at den korteste rute altid tager formen CCC eller CSC, hvor C er enten l eller r, og S svarer til s. Det lader til, at det ikke er muligt rent matematisk at opstille en enkelt ligning for den korteste rute, men da det vides, at den optimale rute antager formen CCC eller CSC fås derved et begrænset sæt af seks ruteklasser. Den optimale rute vælges som den korteste rute fra de ruteklasser, der har en løsning for de pågældende punkter og orientering [Reeds and Shepp, 1990, side 367-368]. Disse seks mulige klasser er: lrl, lsl, lsr, rlr, rsr samt rsl, jævnfør [Dubins, 1957, side 515].

[Kongsbak et al., 2002] prøver at finde den korteste rute alene for CSC-klasserne. Dette var den artikel vi blev inspireret af oprindeligt, men det lykkedes dem rent faktisk ikke at udregne den korteste rute! Efter at have læst [Dubins, 1957] og [Reeds and Shepp, 1990], kan vi se at, hvis man følger algoritmen beskrevet i [Kongsbak et al., 2002, afsnit 4.1] findes en ikke-optimal rute. Det på side 30 i artiklen beskrevne valg af startcirkel giver ikke den korteste rute, for eksempel modbevist af tilfældet, hvor der skal køres fra $(0,0,\frac{\pi}{2})$ til $(1,3,0)$ ved krumningsradius $\rho = 1$. Den beskrevne algoritme vil finde en rute langs de i artiklen benævnte cirkelcentre $1M$ $2N$ med længde $\frac{5\pi}{2} = 7.85$. Men den korteste rute går langs $1N$ $2N$ med længde $\frac{3\pi}{2} + \sqrt{2 \cdot 2^2} = 7.54$, jævnfør afsnit 4.3.4, 4.3.5 og 4.3.6 nedenfor.

For at kunne finde den korteste rute skal t, u og v selvsagt kunne findes for hver af de seks ruteklasser, såfremt de har en løsning. Løsningerne fremkommer alle ved relativ simpel geometri. Nogle af klasserne kan fremkomme med nul eller to ruter, som bud på en løsning (lrl og rlr), andre nul eller én (lsr og rsl), og andre vil altid fremkomme med én løsning (lsl og rsr). Der vil derfor maksimalt være otte løsninger, hvoriblandt den optimale rute skal findes.

At finde en rute fra positionen (x_1, y_1, θ_1) til positionen (x_2, y_2, θ_2) kan forenkles til

at finde en rute fra positionen $(0,0,0)$ til en position (x,y,θ) uden at løsningen er mindre generel. Denne normering beskrives i afsnit 4.3.2. Grundet en *spejlingsegenskab*, som beskrevet i afsnit 4.3.3 på side 18, kan vi tillige nøjes med at finde løsninger for de første tre ruteklasser lrl, lsl og lsr. Vi kan derefter bruge spejlingsegenskaben til at finde de tilsvarende løsninger for de sidste tre klasser rlr, rsr og rsl. Afsnittene 4.3.4, 4.3.5 og 4.3.6 begyndende på side 18 beskriver, hvordan vi henholdsvis finder løsninger for klasserne lrl, lsl og lsr. Vi har valgt at udlede formlerne i denne opgave, da vi fandt op til flere fejl i [Reeds and Shepp, 1990], som ganske vist postulerer løsningerne (delvist fejlagtigt), men som ikke udleder dem. Kapitlet afsluttes med en delkonklusion i afsnit 4.3.7.

4.3.2 Normering

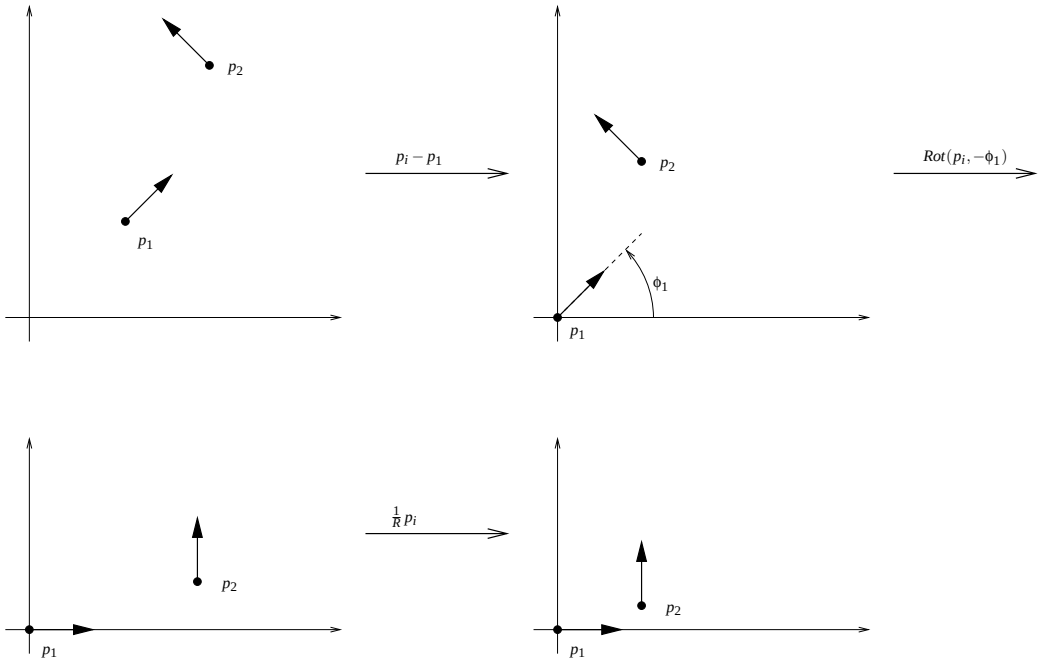
For at forenkle udledningen af kurveformlerne, kan udledningen af den korteste kurve fra positionen (x_1,y_1,θ_1) til positionen (x_2,y_2,θ_2) reduceres. Man kan nemlig nøjes med at finde den korteste kurve fra $(0,0,0)$ til et punkt (x,y,θ), hvor kun $\rho_{min} = 1$ betragtes, uden at gøre formlerne mindre generelle. Dette beskrives ikke i [Reeds and Shepp, 1990], som ikke kigger på det generelle tilfælde, men vi begrunder det herunder.

At normeringen ikke gør formlerne mindre generelle grunder i, at der til enhver normering af to positioner findes en invers normering af den udledte kurve mellem de normerede positioner. Normeringen består af understående transformationer af start- og slutpositionen p_1 og p_2, illustreret på figur 2. pt_i er de transformerede positioner:

$$pt_i = p_i - p_1 \tag{41}$$
$$pt_i = Rot(pt_i, -\phi_1) \tag{42}$$
$$pt_i = \frac{1}{R} pt_i \tag{43}$$

Her betegner $Rot(p,\alpha)$ roteringen af positionen $p = (p_x, p_y, p_\phi)$ α radianer omkring punktet $(0,0)$, inklusiv rotation af orienteringen, dvs. α adderes til p_ϕ. R betegner radius af kurvens cirkelbuer, hvor $R = \rho_{min}$ for robotten. I formel 41 og 43 berøres kun x- og y-koordinaten, ikke orienteringen.

Når en kurve γ er udregnet (se nedenstående) efter den ovenstående normering, går den fra positionen $(0,0,0)$ til positionen (x,y,θ). Den inverse transformation af kurven ligger delvist i dens repræsentation, som består af en udgangsposition



Figur 2: Normering af startpunktet p_1 og slutpunktet p_2. Pilene udtrykker orienteringen i punktet; længden af pilene har ingen betydning.

(dvs. et punkt i planen og en orientering), samt en sammenhængende række af bue- og liniestykker, hvor buestykkerne har en implicit cirkelradius $r = 1$. Hvert bue- eller liniestykkes (herefter *segment*) startposition er givet ved det forrige segments slutposition; selvfølgelig med undtagelse af det første, som har den samlede kurve γ's startposition.

Den inverse transformation af γ består af:

- Multiplicer segmentlængder med R og sæt $r = R$.
- Sæt kurvens startposition til den oprindelige startposition (x_1, y_1, θ_1).

Efter den inverse transformation går γ nu fra positionen (x_1, y_1, θ_1) til positionen (x_2, y_2, θ_2).

4.3.3 Spejling

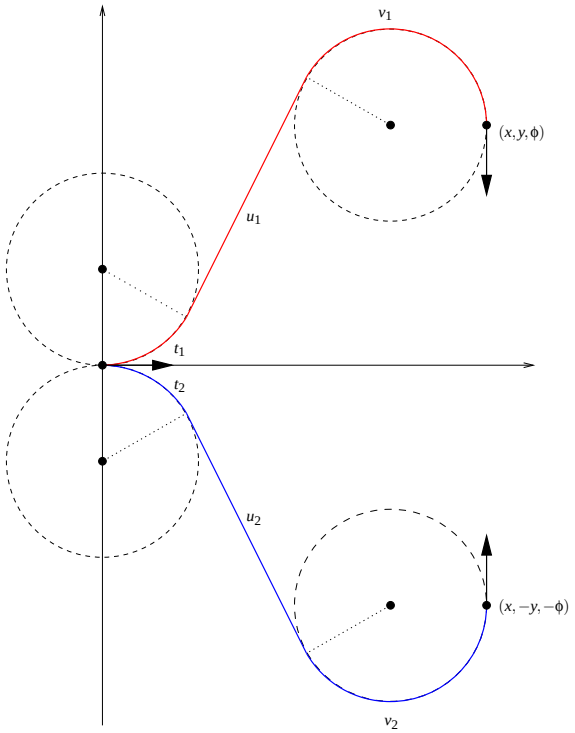
Spejlingsegenskaben beskrives bedst ved et eksempel og en illustration; se figur 3. Kendes for eksempel formlen for udregningen af den korteste kurve $l^+_{t_1} s^+_{u_1} r^+_{v_1}$ fra $(0,0,0)$ til et vilkårligt position (x,y,ϕ) (jævnfør 4.3.6), udregner denne formel også kurven $r^+_{t_2} s^+_{u_2} l^+_{v_2}$ fra $(0,0,0)$ til punktet $(x,-y,-\phi)$. Som det ses af illustrationen, gælder nemlig:

$$\begin{aligned} t_1 &= t_2 \\ u_1 &= u_2 \\ v_1 &= v_2 \end{aligned}$$

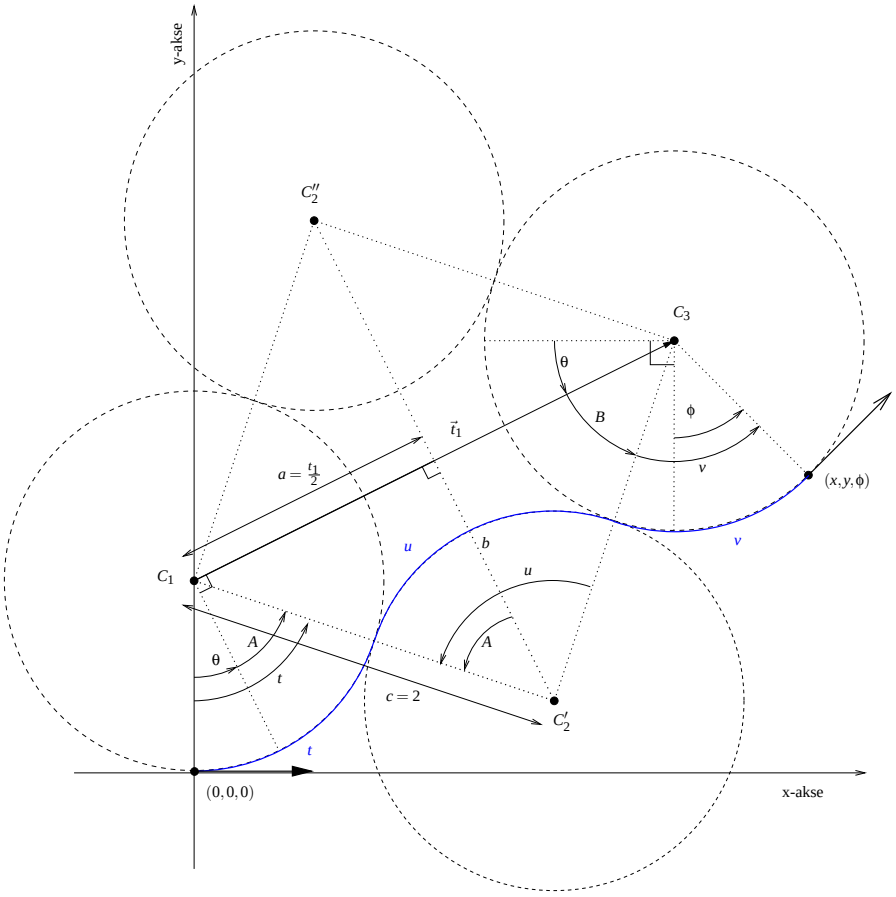
Ved hjælp af spejlingsegenskaben kan formlerne beskrevet i afsnit 4.3.4 (lrl), 4.3.5 (lsl) og 4.3.6 (lsr) altså også bruges til at udregne kurverne for henholdsvis rlr, rsr og rsl klasserne.

4.3.4 Udledning af formler for kurven $l^+_t r^+_u l^+_v$

Ud fra startpositionen $(0,0,0)$ og slutpositionen (x,y,ϕ) ønskes længderne t, u og v for kurven $l^+_t r^+_u l^+_v$ udledt. Udledningen er analog til [Reeds and Shepp, 1990, side 390], formel 8.3, som dog udregner længderne for kurven $l^+_t r^-_u l^+_v$, og som desuden indeholder en del fejl. Længderne udledes geometrisk ved hjælp af figur 4.



Figur 3: Spejlingsegenskaben: Med rødt: Kurven $l^+_{t_1} s^+_{u_1} r^+_{v_1}$ fra $(0,0,0)$ til (x,y,ϕ). Med blåt: Kurven $r^+_{t_2} s^+_{u_2} l^+_{v_2}$ fra $(0,0,0)$ til $(x,-y,-\phi)$.



Figur 4: Geometrisk udledning af længderne t, u og v for kurven $l_t^+ r_u^+ l_v^+$ med startposition $(0,0,0)$ og slutposition (x,y,ϕ).

Til brug i det følgende, samt i afsnit 4.3.5 og 4.3.6, defineres to hjælpefunktioner R og M. Først defineres funktionen R: Når der skrives $(r,\theta) = R(x,y)$ menes transformationen til polære koordinater, dvs. $r\cos\theta = x$, $r\sin\theta = y$, $r \geq 0$, $0 \leq \theta < 2\pi$. Funktionen M defineres nu så $\phi = M(\theta)$, hvis $\phi \equiv \theta \bmod 2\pi$, $0 \leq \phi < 2\pi$.

I [Reeds and Shepp, 1990, side 390] findes analoge definitioner for R og M, dog med undtagelse af intervallerne $0 \leq \theta < 2\pi$ og $0 \leq \phi < 2\pi$. De er i artiklen defineret som henholdsvis $-\pi \leq \theta < \pi$ og $-\pi \leq \phi < \pi$. Dette er vi helt sikre på er en fejl, da det vil føre til negative længder (svarende til at robotten skal bakke) i de udledte formler i artiklen, som **kun** medtager positive længder; Det drejer sig bl.a. om de for denne rapport interessante formler 8.1 og 8.2, og til dels også 8.3, fremført på side 390 i artiklen.

Betragt igen figur 4. Med definitionen af funktionerne R og M kan de ønskede længder nu udledes. Først findes C_3, som er centrum for den sidste cirkel, som ruten følger:

$$\begin{aligned} \xi &= x - \sin\phi, \quad \eta = y + \cos\phi, \\ C_3 &= (\xi,\eta) \end{aligned} \tag{44}$$

Dernæst udregnes længden og vinklen af vektoren $\vec{t}_1$, som er en vektor fra $C_1 = (0,1)$ til C_3:

$$(t_1,\theta) = R(\xi,\eta-1) \tag{45}$$

Hvis $t_1 > 4$ findes ingen løsning. Da er centrum af startcirklen C_1 nemlig så langt fra centrum af slutcirklen C_3, at en mellemliggende cirkel ikke vil kunne forbinde start- og slutcirklen. Hvis $t_1 \leq 4$ findes vinklen A ud fra den retvinklede trekant med siderne a, b og c på figur 4:

$$\begin{aligned} \sin A &= \frac{a}{c}: \\ \sin A &= \frac{\frac{t_1}{2}}{2} = \frac{t_1}{4} \quad \Rightarrow \\ A &= \arcsin(\frac{t_1}{4}) \quad \vee \quad A = \pi - \arcsin(\frac{t_1}{4}) \end{aligned} \tag{46}$$

Da $t_1 > 0 \Rightarrow 0 \leq \arcsin(\frac{t_1}{4}) \leq \frac{\pi}{2}$. Hvis løsningen med $0 \leq A \leq \frac{\pi}{2}$ anvendes, findes ruten som følger cirklen med centrum i C_2', som er den rute, der er indtegnet på figuren. Hvis løsningen med $\frac{\pi}{2} \leq A \leq \pi$ anvendes, findes ruten som følger cirklen

med centrum i C_2''. Denne løsning nævner [Reeds and Shepp, 1990] at de aldrig har observeret som værende optimal, men de beviser det ikke. Vi medtager den for en god ordens skyld. Af figuren ses nu:

$$\begin{aligned} t &= M(A+\theta) && (47)\\ u &= 2A && (48)\\ v &= M(\phi+(\frac{\pi}{2}-\theta-B)) \\ &= M(\phi+(\frac{\pi}{2}-\theta-(\frac{\pi}{2}-A))) \\ &= M(\phi+A-\theta) && (49) \end{aligned}$$

Her er M-funktionen anvendt, så længderne t og v kommer til at ligge i intervallet $[0;2\pi[$.

Som nævnt har artiklen et par fejl: Betingelsen, at hvis $t_1 > 4$ så findes ingen løsning, er i artiklen noteret som "$u_1^2 > 4$". Det er oplagt en fejl, og sikkert et levn fra artiklens formel 8.2. Herefter ses hurtigt, at i resten af formel 8.3 i artiklen skal alle steder, hvor der står "u_1^2" erstattes af "u_1". Desuden findes længderne u, v og w, hvor det var længderne t, u og v som skulle findes. Der er nok tale om en simpel trykfejl. Sidst kan nævnes, at udregningen i artiklen af længden u foretages noget indviklet, og resultatet er ikke altid korrekt. Artiklen udregner som nævnt længden u for kurven $l_t^+ r_u^- l_v^+$, gengivet nedenfor ved formel 50 hentet fra artiklen, og læg mærke til at her skal længden u være negativ:

$$(T,u) = R(2-\xi\sin t+\eta\cos t,\ \xi\cos t+\eta\sin t) \qquad (50)$$

Denne formel giver ikke altid det rigtige resultat, og kan desuden formuleres meget enklere, som det kan ses af figur 4, og helt analogt til formel 48:

$$u = 2\pi - 2A \qquad (51)$$

At formel 50 ikke altid giver det rigtige resultat kan ses på for eksempel kurven fra $(0,0,0)$ til $(3,0,\frac{3\pi}{4})$. På figur 5 ses situationen med de to rigtige løsninger indtegnet. Indsættes tallene i ligningerne 47, 51 og 49 fås:

$$\begin{aligned} \xi &= 2.2929 \\ \eta &= -0.7071 \\ A &= 0.7961 \end{aligned}$$

$$
\begin{aligned}
t_1 &= 0.1561 \\
t_2 &= 1.7055 \\
u_1 &= -4.6909 \\
u_2 &= -1.5922 \\
v_1 &= 3.7923 \\
v_2 &= 5.3417
\end{aligned}
$$

Indsættes tallene derimod i formel 50 i stedet for formel 51, fås:

$$
\begin{aligned}
u_1 &= 1.5922 \\
u_2 &= 4.6909
\end{aligned}
$$

Formel 50 giver i dette tilfælde altså positive u, svarende til at robotten skal køre fremad, frem for at bakke!

4.3.5 Udledning af formler for kurven $l_t^+ s_u^+ l_v^+$

Længderne t, u og v ønskes udledt for kurven $l_t^+ s_u^+ l_v^+$ med startposition $(0,0,0)$ og slutposition (x,y,ϕ). Udledningen giver samme resultat som vist i formel 8.1. i [Reeds and Shepp, 1990, side 390]. Længderne udledes geometrisk ved hjælp af figur 6.

Vi starter med at finde t og u som henholdsvis længden og vinklen af vektoren $\vec{u}$, som går fra C_1 til C_2:

$$(u,t) = R(x - \sin\phi, y - 1 + cos\phi) \tag{52}$$

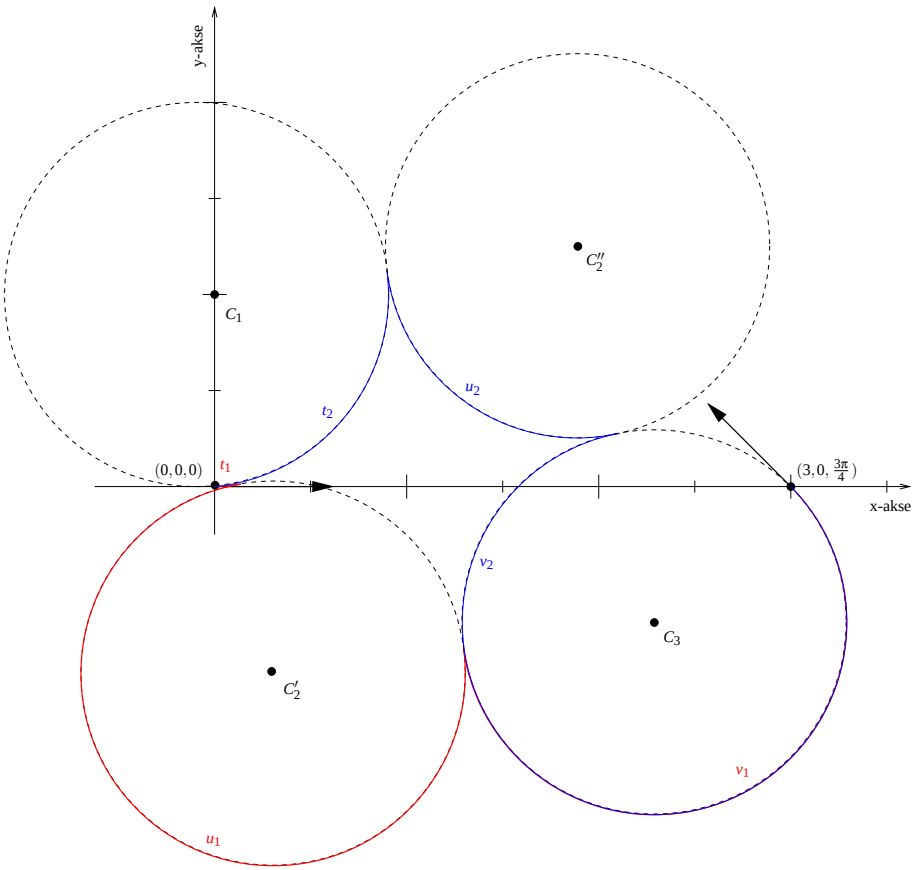
Af figuren ses også:

$$v = M(\phi - t) \tag{53}$$

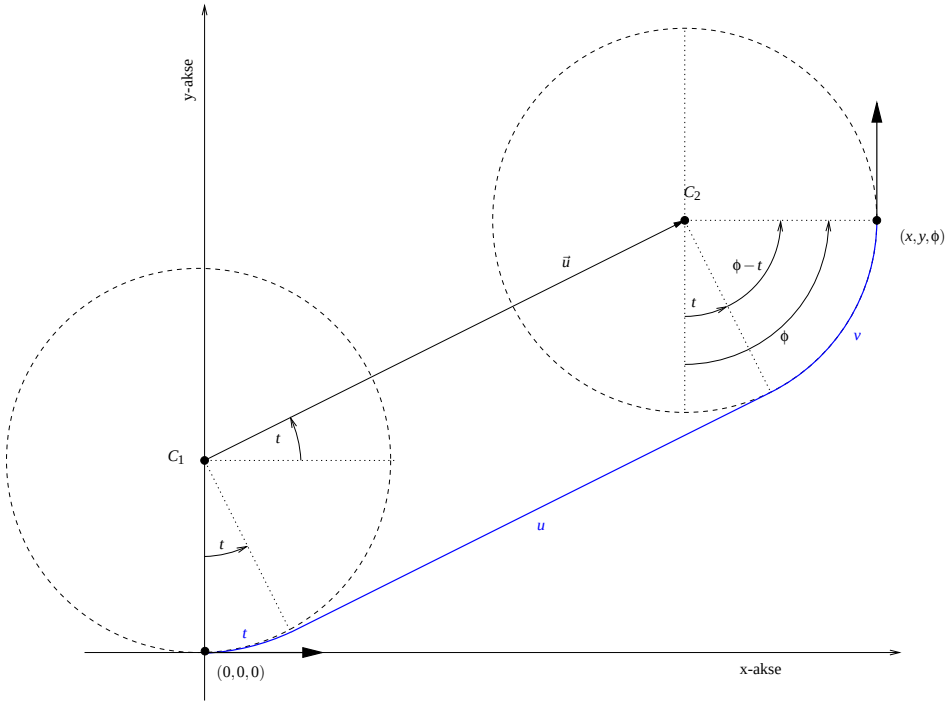
4.3.6 Udledning af formler for kurven $l_t^+ s_u^+ r_v^+$

Igen ønskes længderne t, u og v udledt, denne gang for kurven $l_t^+ s_u^+ r_v^+$, med startposition $(0,0,0)$ og slutposition (x,y,ϕ). Udledningen giver samme resultat som formel 8.2 i [Reeds and Shepp, 1990, side 390]. Længderne udledes geometrisk ved hjælp af figur 7.

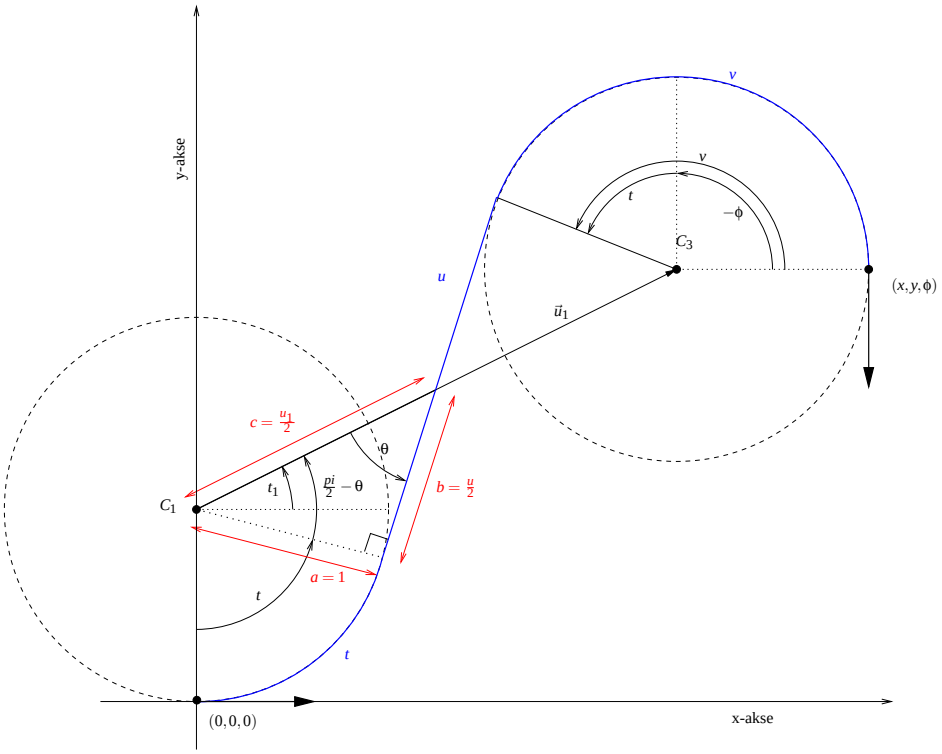
Først findes $C_3 = (\xi, \eta)$, som er centrum for den sidste cirkel på ruten, som følger:



Figur 5: Eksempelkurve fra $(0,0,0)$ til $(3,0,\frac{3\pi}{4})$. Der findes to løsninger, markeret henholdsvis med rødt og blåt.



Figur 6: Geometrisk udledning af længderne t, u og v for kurven $l_t^+ s_u^+ l_v^+$ med startposition $(0,0,0)$ og slutposition (x,y,ϕ).



Figur 7: Geometrisk udledning af længderne t, u og v for kurven $l_t^+ s_u^+ r_v^+$ med startposition $(0,0,0)$ og slutposition i (x,y,ϕ).

$$\begin{aligned} \xi &= x+\sin\phi, \quad \eta = y-\cos\phi, \\ c_3 &= (\xi,\eta) \end{aligned} \tag{54}$$

Dernæst udregnes længden og vinklen af vektoren $\vec{u}_1$, som er en vektor fra $C_1 = (0,1)$ til C_3:

$$(u_1,t_1) = R(\xi,\eta-1) \tag{55}$$

Her optræder endnu en fejl i artiklens formel 8.2. Den skriver formel 55 som $(u_1,t) = R(\xi,\eta-1)$, svarende til at længden t allerede udregnes her. Det er dog blot en trykfejl, da t_1 refereres i et senere udtryk, som rent faktisk udregner t.

Hvis $u_1 < 2$ findes ingen løsning; da overlapper de to cirkler hinanden, og kan ikke forbindes med en linie så den ønskede kørselsretning overholdes. Ellers findes længden u ud fra den retvinklede trekant med siderne a, b og c på figur 7 ved hjælp af Pythagoras sætning:

$$\begin{aligned} a^2 &= b^2+c^2: \\ \left(\frac{u_1}{2}\right)^2 &= \left(\frac{u}{2}\right)^2+1^2 \quad \Rightarrow \\ \frac{u_1^2}{4} &= \frac{u^2}{4}+1 \quad \Rightarrow \\ u_1^2 &= u^2+4 \quad \Rightarrow \\ u &= \sqrt{u_1^2-4} \end{aligned} \tag{56}$$

Ud fra den fundne længde u kan t nu findes. Først findes vinklen θ i den retvinklede trekant med siderne a, b og c på figur 7 ved hjælp af den trigonometriske funktion $\tan\theta = \frac{a}{b}$:

$$\begin{aligned} \tan\theta &= \frac{1}{\frac{u}{2}} \quad \Rightarrow \\ \theta &= \arctan\frac{2}{u} \end{aligned} \tag{57}$$

Artiklen udregner θ som:

$$(T, \theta) = R(u, 2) \tag{58}$$

Her er T blot en "dummy-variabel". Formel 58 giver naturligvis den samme værdi af θ som formel 57, jævnfør definitionen af R i afsnit 4.3.4; det er blot noget mindre gennemskueligt, hvordan resultatet bliver udledt.

Nu kan t findes ud fra θ ved igen at betragte figur 7:

$$\begin{aligned} t + (\frac{\pi}{2} - \theta) &= t_1 + \frac{\pi}{2} \quad \Rightarrow \\ t &= t_1 + \theta \end{aligned} \tag{59}$$

Da t ønskes at ligge i intervallet $[0; 2\pi[$, så der hverken køres baglæns eller mere end én gang rundt på cirklen, vælges:

$$t = M(t_1 + \theta) \tag{60}$$

Ligeledes ønskes længden $v \in [0; 2\pi[$, og ud fra figuren ses:

$$v = M(t - \phi) \tag{61}$$

4.3.7 Beregning af den korteste kurve

Dette afsnit giver en oversigt over resultaterne fra afsnit 4.3.2 til afsnit 4.3.6. Når den korteste rute fra startpositionen $p_1 = (x_1, y_1, \phi_1)$ til slutpositionen $p_2 = (x_2, y_2, \phi_2)$ med mindste drejeradius ρ_{min} skal beregnes, udføres følgende trin:

1. p_1 og p_2 normeres, som beskrevet i afsnit 4.3.2. Den søgte rute kan nu beregnes ved at beregne ruten fra startpositionen $(0, 0, 0)$ til slutpositionen $p = (x, y, \phi)$, med mindste drejeradius $\rho_{min} = 1$.

2. Længderne t, u og v beregnes for kurverne $l_t^+ r_u^+ l_v^+$, $l_t^+ s_u^+ l_v^+$ og $l_t^+ s_u^+ r_v^+$ som beskrevet i afsnit 4.3.4, 4.3.5 og 4.3.6, samt for kurverne $r_t^+ l_u^+ r_v^+$, $r_t^+ s_u^+ r_v^+$ og $r_t^+ s_u^+ l_v^+$ ved hjælp af spejlingsegenskaben beskrevet i afsnit 4.3.3. Dette kan resultere i op til otte kurvemuligheder, da kurven $l_t^+ r_u^+ l_v^+$ og $r_t^+ l_u^+ r_v^+$ begge kan give op til to resultater.

3. Af de op til otte fundne normerede ruter udvælges den korteste. Længden af hver kurve beregnes som summen af længderne t, u og v: $l = t + u + v$.

4. Ved hjælp af den inverse normering beskrevet i afsnit 4.3.2 transformeres kurvens længder t, u og v, så den nu beskriver den korteste mulige kurve, γ_{min}, fra den oprindelige startposition, $p_1 = (x_1, y_1, \phi_1)$, til den oprindelige slutposition, $p_2 = (x_2, y_2, \phi_2)$.

Den endelige korteste kurve kan repræsenteres som en startposition (x_1, y_1, ϕ_1), samt de 3 fundne længder t, u og v, da slutpositionen ligger implicit i denne repræsentation: Kurven ender nemlig i den oprindelige slutposition (x_2, y_2, ϕ_2).

4.4 Positionsbestemmelse

For at kunne følge en kurve, er det nødvendigt for robotten at kende dens position. Vi har valgt at implementere to forskellige positionsbestemmelsesmetoder, nemlig et simuleret Global Positioning System (GPS) og *Dead Reckoning*. Disse vil blive beskrevet i de følgende afsnit.

4.4.1 Global Positioning System

Da robotten bliver simuleret, og simulationen naturligvis kender robottens position, kan robotten let få sin position bestemt: Simulationen skal blot sørge for at sætte den aktuelle position på et simuleret GPS-modul.

Man kan tænke sig, at det på en ikke-simuleret fysisk robot svarer til et fysisk GPS-modul. Denne form for positionsbestemmelse har den robot som har givet inspiration til denne opgave ikke, jævnfør indledningen. Men det er en meget praktisk funktionalitet at have under implementeringen af simulationen, da robottens positionsbestemmelse herved ikke giver anledning til nogen fejl. Ligeledes er denne positionsbestemmelse også anvendt ved evalueringen af rute-følge-rutinerne; se afsnit 4.5.

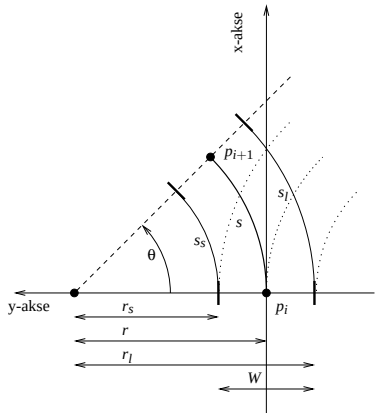
4.4.2 Dead Reckoning

Som nævnt har den robot, som opgaven er inspireret af, ikke noget GPS-modul. Til gengæld har den et tachometer (svarer til en omdrejningstæller) på hvert baghjul. For hver gang et baghjul har kørt en vis fast distance, inkrementeres tachometerets register. En inkrementering kaldes også et “tick”. Ved at aflæse tachometerne kan robottens position bestemmes, som forklaret herunder.

Når robotten i et givet tidsrum har kørt et vist stykke, kan dette stykke tilnærmes med en cirkelbue, som vist på figur 8. For et udsnit på θ radianer af en cirkel med

radius r, er buelængden s givet ved:

$$s = r\theta \tag{62}$$



Figur 8: Dead Reckoning. Bemærk at x-aksen peger opad og y-aksen mod venstre! p_i er startpositionen af midtpunktet af bagaksen, p_{i+1} er det samme midtpunkts slutposition. W er afstanden mellem hjulene. s_s er den strækning, som det hjul der har kørt kortest, har kørt; s_l betegner den strækning, som det hjul der har kørt længst, har kørt.

Vi har altså, jævnfør figur 8:

$$s_s = r_s\theta, s_l = r_l\theta \quad \Rightarrow \quad \frac{s_s}{r_s} = \frac{s_l}{r_l} = \theta \tag{63}$$

Her betegner s_s den strækning, som det hjul der har kørt *kortest*, har kørt; s_l betegner den strækning, som det hjul der har kørt *længst*, har kørt.

Da der også gælder at $r_l = r_s + W$, hvor W er afstanden mellem hjulene, haves:

$$\frac{s_s}{r_s} = \frac{s_l}{r_s + W} \quad \Rightarrow \tag{64}$$

$$s_s r_s + s_s W = r_s s_l \quad \Rightarrow \tag{65}$$

$$r_s(s_l - s_s) = s_s W \quad \Rightarrow \tag{66}$$

$$r_s = \frac{s_s W}{s_l - s_s} \tag{67}$$

Drejeradiussen r kan nu findes for eksempel ud fra r_s som:

$$r = r_s + \frac{W}{2} \tag{68}$$

Den drejede vinkel $\Delta\theta$ kan findes ud fra r_s som i formel 63:

$$\Delta\theta = \frac{s_s}{r_s} \tag{69}$$

Den nye position findes som:

$$p_{i+1} = p_i + (\Delta x,\ \Delta y,\ \Delta\theta) \tag{70}$$

Ved at betragte figur 8 kan $(\Delta x,\ \Delta y)$ findes som:

$$\Delta x = r\sin\Delta\theta \tag{71}$$
$$\Delta y = r(1-\cos\Delta\theta) \tag{72}$$

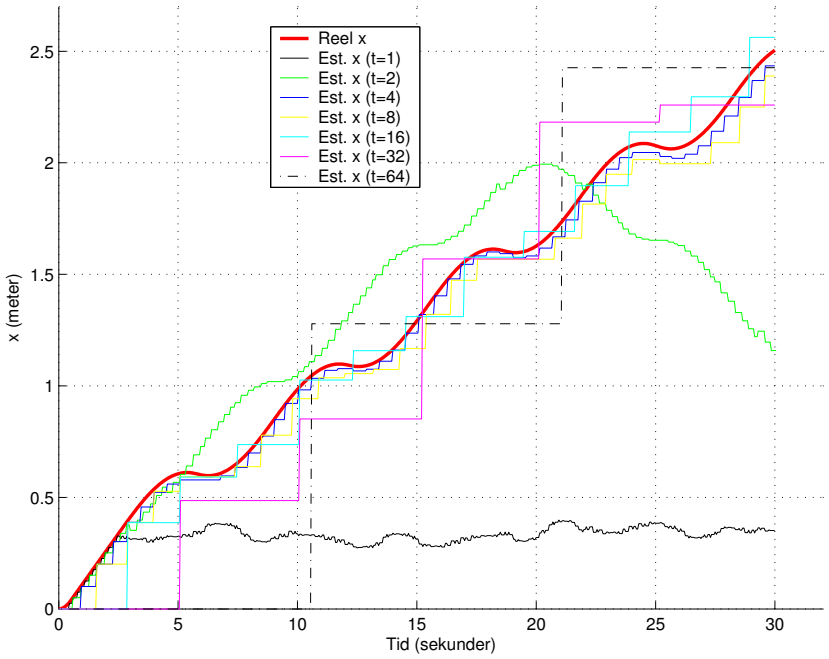
På figuren er det tilfælde illustreret, hvor det venstre hjul kører kortere end højre. I modsat fald skal fortegnet på Δy i formel 72 vendes.

Ved denne form for positionsbestemmelse er valget af, hvor langt der går mellem opdateringen af positionen en afvejning: Opdateres for ofte, vil tachometernes diskretisering af den kørte afstand få meget stor betydning. Opdateres for sjældent, vil mindre sving under kørslen ikke blive opdaget. Ved at afprøve forskellige intervalstørrelser empirisk er vi kommet frem til, at opdatere positionen, hver gang begge robottens hjul er kørt mindst 16 ticks siden sidste opdatering. Dette gælder for Murphy-robotten, som har en tachometeropløsning på 40 ticks pr. omdrejning, og en hjulradius på 4 cm.

Figur 9 viser forskellige afprøvede intervalstørrelser ved en tachometeropløsning på 10 ticks pr. omdrejning, og med en hjulradius på 4 cm. Vi har valgt at mindske tachometeropløsningen i denne og de følgende grafer i forhold til Murphy for at tydeliggøre afvigelserne.

Herunder præsenteres en skitse af positionsbestemmelsen som pseudokode. Funktionen `ticksToPos` udregner formel 71 og 69. `p` er den ønskede position og `THRESHOLD` er intervalstørrelsen:

```
oldLeft = oldRight = 0;
p = new Position(0, 0, 0);

while(true) {
  left = getLeftTicks(), right = getRightTicks();
  if (left - oldLeft < THRESHOLD || right - oldRight < THRESHOLD) {
    continue;
  }
  p_delta = ticksToPos(left - oldLeft, right - oldRight);
  p = p + rotate(p_delta, p.orientation);
  oldLeft = left, oldRight = right;
}
```



Figur 9: Afprøvning af forskellige opdateringsintervaller. Den reelle x-position og den estimerede x-position ved intervaller på henholdsvis 1, 2, 4, 8, 16, 32 og 64 ticks. Som det ses, giver en intervallængde på omkring 4-16 en rimelig præcision. Robotten kører i et siksak mønster af alternerende venstre- og højresving.

Læg mærke til rotationen `rotate(p_delta, p.orientation)` af p_{delta}: Her tages højde for, at orienteringen af robotten ved starten af det kørte interval i det generelle tilfælde ikke er 0. Orienteringen er vist som 0 på figur 8.

Robottens position bliver således kun opdateret, hver gang hele det valgte interval er nået. Dette er dog temmelig uhensigtsmæssigt, når robotten skal følge en kurve, som beskrevet i afsnit 4.5. Her er det nødvendigt med meget finere opdateringsintervaller for at sikre en glat og korrekt kørsel. Heldigvis kan der let rettes op på denne uhensigtsmæssighed.

Herunder er vist pseudokode for en finere positionsbestemmelse, som ved hvert eneste nye tick udregner en ny position – det er blot en udvidelse af den før viste pseudokode, som genberegner positionen for hvert nyt tick, men den overordnede præcision er den samme som for den forrige pseudokode:

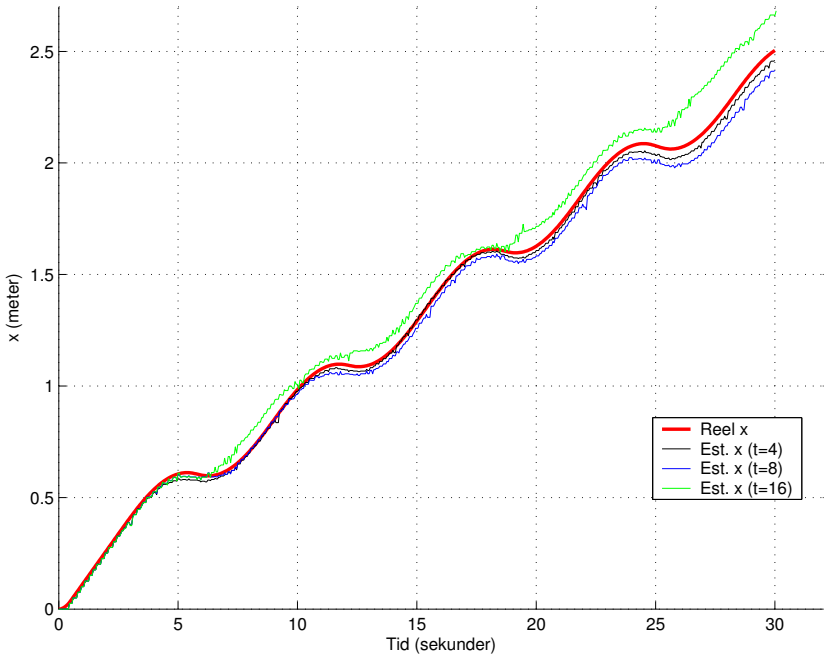
```
oldLeft = oldRight = 0;
Position p        = new Position(0, 0, 0);
Position p_coarse = new Position(0, 0, 0);

while (true) {
  left = getLeftTicks(), right = getRightTicks();
  Position p_delta = ticksToPos(left - oldLeft, right - oldRight);
  p_delta = rotate(p_delta, p_coarse.orientation);

  if (left - oldLeft < THRESHOLD || right - oldRight < THRESHOLD) {
    p = p_coarse + p_delta;
  } else {
    p_coarse += p_delta;
    p = p_coarse;
    oldLeft = left, oldRight = right;
  }
}
```

Figur 10 og 11 viser en afprøvning af den finere positionsbestemmelse. Som det ses, opnås en væsentligt mindre afvigelse fra den reelle position, end ved den førstnævnte positionsbestemmelse. Figur 11 viser, at ved færre og mindre skarpe sving betyder valgte af intervalstørrelsen mindre.

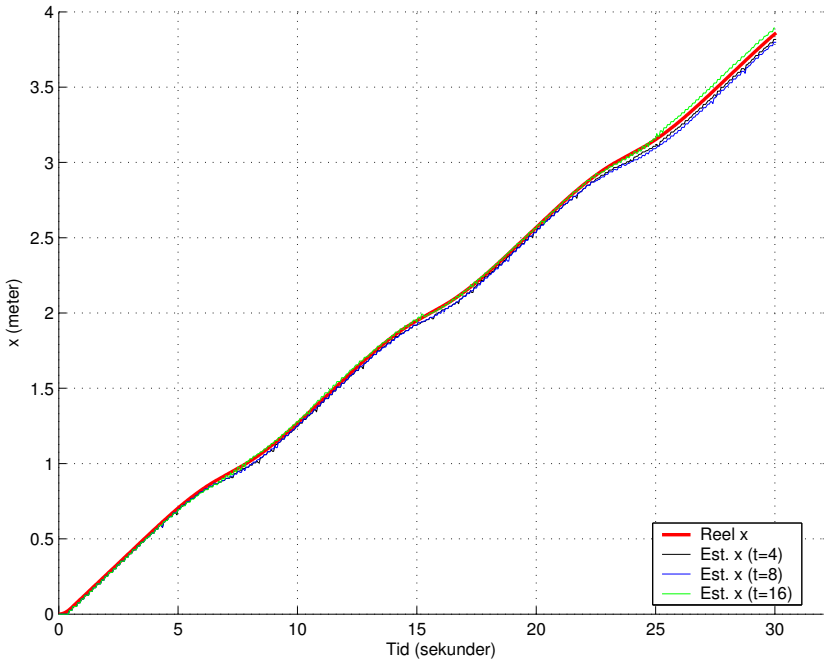
4.5 At følge en rute

Første del af position-til-position bevægelsesproblemet, beskrevet i afsnit 4.1, finder en rute, repræsenteret ved en sammenhængende kurve γ. Følger robotten kurven γ, vil den bevæge sig fra startpositionen til slutpositionen som ønsket.

Anden del af problemet er nu at finde en algoritme, som får en bil-agtig robot til at følge den fundne kurve.



Figur 10: Samme kurve som i figur 9 følges, men denne gang med den finere positionsbestemmelse. Intervalstørrelser på 4, 8 og 16 er afprøvet.



Figur 11: Den finere positionsbestemmelse afprøvet på en kurve, hvor der køres i et siksak mønster af gentagne: højresving, ligeud, venstresving, ligeud. Svingene er halvt så skarpe, som den fulgte kurve på figur 9 og figur 10.

Da der i følge vores erfaring ikke findes en egentlig optimal algoritme til løsning af dette problem, vil der herunder blive foreslået tre mulige, men (formentligt) ikke optimale løsninger.

Alle tre løsninger antager, at robottens hastighed bliver styret eksternt af en hastighedsregulering, jævnfør afsnit 2.2.1, som ikke er under algoritmernes kontrol. Specielt kontrollerer algoritmerne ikke den afledte af positionen $(\dot{x},\dot{y})$ i slutpunktet, altså robottens hastighed i slutpunktet. Derfor gives der ingen garantier om, at robotten *stopper* i slutpunktet. En løsning på dette skal nok findes i en mere avanceret hastighedsregulering, som er et problem vi har valgt at se bort fra i denne opgave.

4.5.1 Den naive løsning

γ er en sammenhængende kurve konstrueret af cirkelbuer og rette linier (segmenter), og tangenten af startpunktet af γ er parallel med robottens orientering i startpositionen. Derfor kan γ følges ved, segment for segment, at køre segmentets længde, opmålt ved hjælp af robottens tachometer. Er det aktuelle segment en ret linie, sættes forhjulenes vinkel til 0°, og er det aktuelle segment en cirkelbue, sættes forhjulenes vinkel til en vinkel ϕ, som kan udledes af formel 3 på side 7 ved at isolere ϕ.

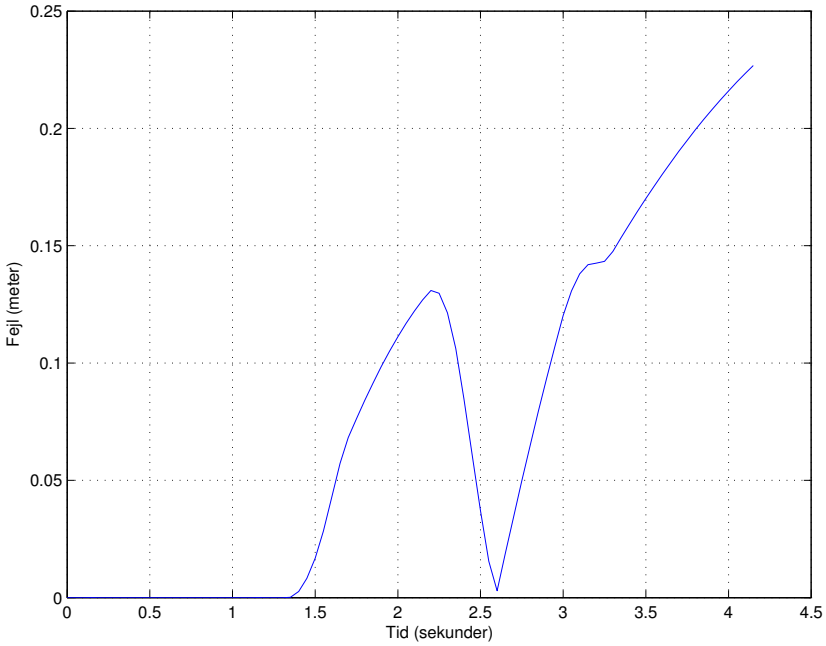
Denne løsning har dog to store problemer:

1. Vinkelaccelerationen som robottens forhjul kan dreje med må nødvendigvis være begrænset for en reel fysisk robot. Dvs. at forhjulene er et stykke tid om at dreje fra en vinkel til en anden, og dette vil føre til en afvigelse fra kurven γ, med mindre robotten stopper helt op, hver gang hjulvinklen ændres.
2. Det andet problem er, at algoritmen er en *åben-sløjfe*-styring, jævnfør [Ole Jannerup, 2000, side 12], idet der ikke er nogen feedback af den reelle position medregnet i algoritmen. Derfor vil unøjagtigheder i styringen, som for eksempel slør i forhjulene, føre til ikke-opdagede afvigelser fra kurven.

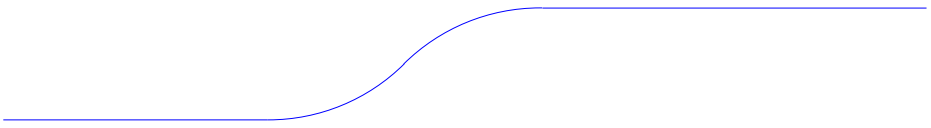
På figur 12 ses positionsafvigelsen, når en robot følger testkurven på figur 13 med den naive løsning. Som det ses, afviger robottens position fra testkurven. Værst er det, at afvigelsen er akkumulerende over tid.

4.5.2 P-regulering – en ad-hoc løsning

Problemet at følge en rute er en del af udfordringen ved RoboCup-konkurrencen, jævnfør [rob, 2004]. Et hovedelement af den er nemlig at følge en hvid tapestreg



Figur 12: Robotten følger testkurven i figur 13, med en fart på 0,75 m/s, ved hjælp af den naive kurvefølgningsalgoritme.



Figur 13: Testkurve. Kurven består af 4 segmenter: En 1,0 meter lang linie, to cirkler med radius $\frac{\pi}{4}$ meter og længde 0,8 meter, og en 2,0 meter lang linie.

("ruten") på et mørkegråt underlag. Under udviklingen af en robot til konkurrencen blev problemet løst ved, at en liniesensor blev placeret ved robotten forhjulsaksel. Liniesensoren giver, efter en passende kalibrering, afstanden regnet med fortegn mellem centrum af robottens forhjulsaksel og centrum af stregen, herefter benævnt e.

Da det ikke var oplagt, hvorledes hjulvinklen skulle stilles i forhold til e, blev det afprøvet blot at sætte hjulvinklen proportionalt med afstanden:

$$\phi = -ke \tag{73}$$

Ved eksperimentelt at finde en passende k, samt at afskære ϕ ved meget store e, kom metoden til at virke tilfredsstillende ved ikke for høje hastigheder. Metoden svarer til en P-regulering, jævnfør [Ole Jannerup, 2000, side 266], med de dertil hørende problemer, især oversving, stationær fejl og stig-tid.

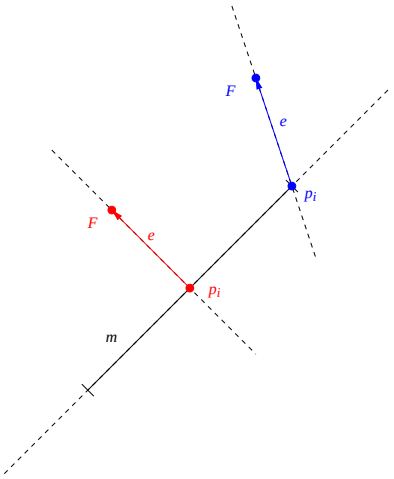
Da denne metode har vist sig at fungere i praksis, er det oplagt at overføre den til ruteføl gningsproblemet, hvilket er forholdsvist enkelt. e vælges nu som den korteste vektor fra et punkt på ruten γ til centrum af forhjulsakslen (punktet F), svarende til [Moret, 2003, side 23]. Da γ består af segmenter, nemlig liniestykker og cirkelbuer, findes den korteste vektor til γ som ved først at finde den korteste vektor til hvert segment, og derefter udvælge den korteste vektor af disse:

- Er segmentet et liniestykke m, findes skæringspunktet p_i mellem (den matematiske, uendeligt lange) linie som m ligger på og linien, som går igennem F, vinkelret på m. Ligger p_i på liniestykket m, udregnes e vektoren fra p_i til F. I modsat fald sættes e til den korteste af vektorerne fra endepunkterne af m til F. Se figur 14.

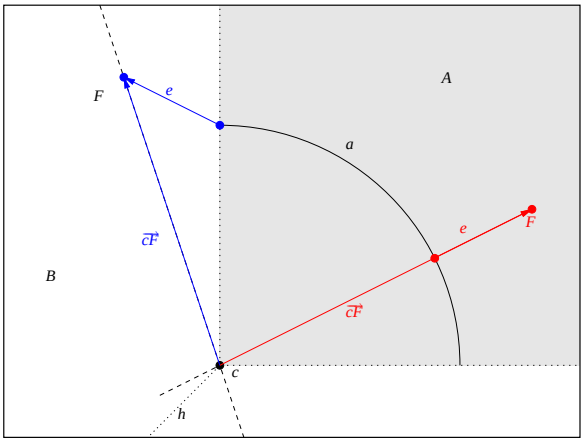
- Er segmentet et buestykke a, deles planen først op i to dele. Den ene del A består af de punkter, som kan nås af en vektor fra punktet c, hvor vektorens vinkel ligger inden for cirkelbuen a. Den anden del B består af resten af planen. Se figur 15.

  Ligger F i delmængden A, vil den matematiske linie, som går gennem c og med samme vinkel som $\overrightarrow{cF}$, skære cirkelbuen a. e sættes så til at være vektoren fra skæringspunktet til F. Ligger F derimod i delmængden B, vil det nærmeste punkt være et af endepunkterne af a, og e sættes til den korteste af vektorerne fra endepunkterne af a til F.

  Den ovenstående definition af en korteste vektor fra et punkt til en cirkelbue har dog et par småproblemer. Ligger F nemlig lige i c, vil afstanden fra



Figur 14: På figuren ses to eksempler på, hvor punktet F kan ligge i forhold til liniestykket m.



Figur 15: På figuren ses to eksempler på, hvor punktet F kan ligge i forhold til cirkelbuen a. Mængden A er markeret med grå baggrund, og bemærk at på figuren ses kun et udsnit af A og B, idet de begge er uendelig store.

F til ethvert punkt på cirkelbuen være lige stor. Ligger F på "vinkelhalveringslinien", kaldet h på figur 15, er afstanden fra F til begge endepunkter af cirkelbuen lige store, og samtidig den mindste afstand til et punkt på cirkelbuen. Begge problemer løses ved at sætte den korteste vektor e til vektoren fra F til *endepunktet* af cirkelbuen.

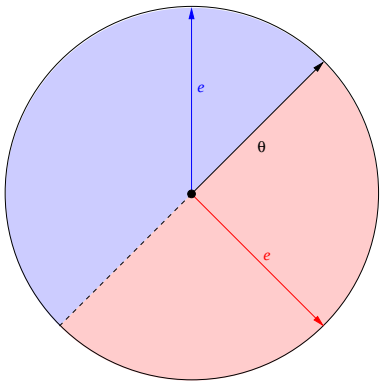
Ud fra den korteste vektor e fra ruten γ, kan følgende omskrivning af formel 73 nu anvendes til hjulstyringen:

$$\phi = \text{sign}(e, \theta) k |e| \tag{74}$$

Her er θ orienteringen af robotten, og funktionen $\text{sign}(e, \theta)$ er defineret som:

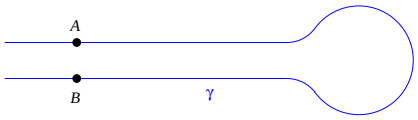
$$\text{sign}(e, \theta) = \begin{cases} 1 & \text{hvis vinklen af } e \text{ er mindre end } \theta \\ 0 & \text{hvis vinklen af } e \text{ er den samme som } \theta \\ -1 & \text{hvis vinklen af } e \text{ er større end } \theta \end{cases}$$

Se også figur 16.



Figur 16: Med rødt: Vinklen af e er mindre end θ, sign(e,θ) = 1, og der skal drejes til højre. Med blåt: Vinklen af e er større end θ, sign(e,θ) = -1, og der skal drejes til venstre.

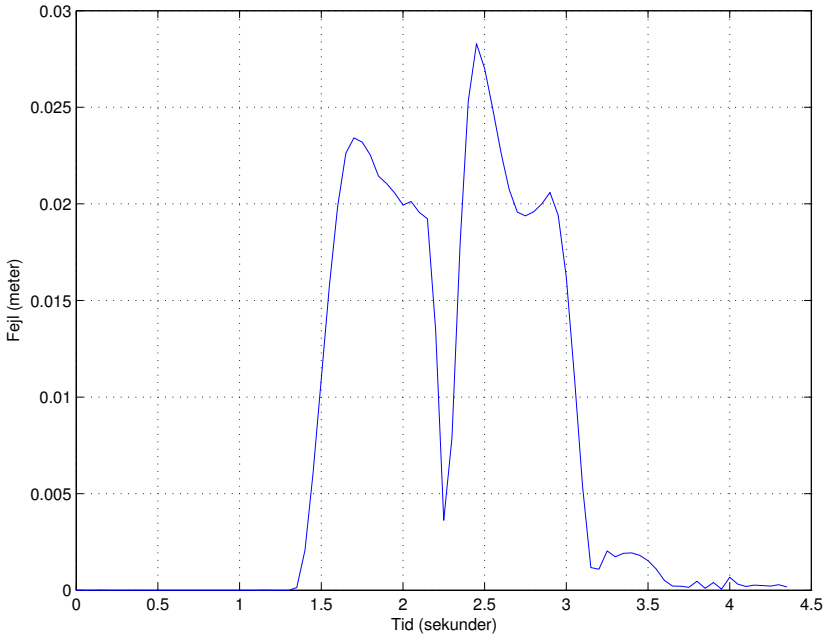
Denne metode giver en stabil styring under de fleste "rimelige" forhold, dvs. når robotten ikke kører for hurtigt, samt når "kurven ikke kommer meget tæt på sig selv". Det sidste grunder i, at kun den korteste afstand til et punkt på kurven betragtes. Metoden giver problemer hvis to punkter i planen ligger tæt på hinanden, men samtidigt er placeret langt fra hinanden på kurven som vist på figur 17. Dette kan der rettes op på ved at gøre metoden mere avanceret, men det har vi valgt ikke at



Figur 17: Punkterne A og B ligger langt fra hinanden regnet langs kurven γ, men ligger euklidisk tæt på hinanden i planen.

gøre; langt de fleste kurver, som genereres af metoden til at finde kortest afstand mellem to punkter, som beskrevet i afsnit 4.3, vil nemlig ikke være problematiske.

På figur 18 ses positionsafvigelsen, når en robot følger testkurven på figur 13 med P-regulering. Som det ses, afviger robottens position meget mindre fra testkurven end med den naive løsning, og fejlen er ikke akkumulerende over tid. Til gengæld kan robotten gå i "selvsving", dvs. en stigende oscillering omkring kurven, hvis kurven har kraftige knæk (steder med meget lille krumningsradius).



Figur 18: Robotten følger testkurven i figur 13, med en fart på 0,75 m/s, ved hjælp af P-regulering. Bemærk y-aksens skala, den er omkring en faktor 10 mindre end i figur 12!

PID-regulering P-reguleringen kan let udviddes til en PID-regulering som beskrevet i [Ole Jannerup, 2000, side 292-296]. Her er især *D*-leddet interessant, idet

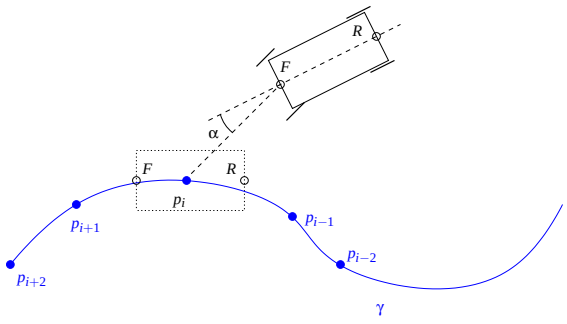
det øger stabiliteten af styringen, og *I*-leddet mindsker afvigelsen fra kurven (mindsker den stationære fejl). En PID-regulering har vi også implementeret, men den vil ikke blive yderligere gennemgået her, idet dens funktionalitet er stort set den samme som P-reguleringens. Dog er styringens stabilitet som sagt større, hvilket naturligvis er en ønskværdig egenskab.

Fælles for både P- og PID-reguleringen er, at de introducerer konstant(er) (k_p for begge, og k_i, k_d for PID), som kan finjusteres. For en nærmere udregning af, hvorledes dette gøres, se [Ole Jannerup, 2000, side 292-296]. Her skal det blot nævnes, at de er finjusteret empirisk.

4.5.3 Successiv punkt-styring

For at perspektivere metoderne i de to foregående afsnit, som begge er "hjemmegjorte", har vi valgt også at implementere en metode fra en artikel, nemlig [Egerstedt et al., 1997, afsnit 6]. Artiklen omhandler ligesom denne rapport position-til-position bevægelsesproblemet, dog med nogle noget anderledes forudsætninger, men artiklen behandler også kurve-følge-problemet. Artiklens løsning summeres herunder.

- Vælg et punkt på kurven. Styrevinklen ϕ sættes til vinklen mellem orienteringen af bilen og retningen af linien fra F til p_i. Dette er vinklen α på figur 19.
- Når bilen er "tæt nok" på p_i, vælges et nyt punkt på kurven, som der så styres efter.

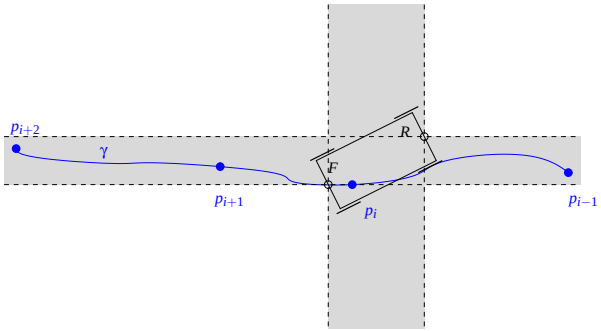


Figur 19: Kurvefølgning ved at køre efter på-hinanden-følgende punkter på kurven. Figuren er baseret på figur 6 i [Egerstedt et al., 1997].

At robotten er ”tæt nok” på et punkt p defineres i artiklen som at den har ”passeret” punktet, hvilket igen defineres som:

$$(x_F - x_p)(x_R - x_p) < 0 \quad \vee \quad (y_F - y_p)(y_R - y_p) < 0 \tag{75}$$

Den ovenstående definition af ”passeret” er dog problematisk. Den tillader at vilkårligt mange punkter vilkårligt langt fra robotten regnes som passerede, som vist på figur 20. Vi har valgt at definere et punkt som passeret, hvis det er inden for en vis afstand fra centrum af robotten. Denne afstand har vi valgt til længden L, som er afstanden mellem robottens for- og bagaksel, jævnfør figur 1 på side 6.



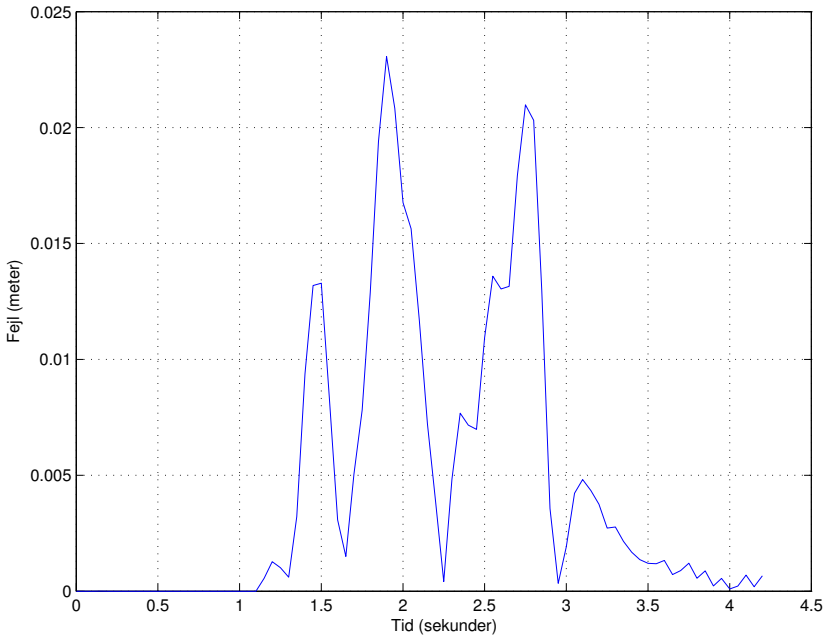
Figur 20: Problemet med artiklens definition af ”passeret”. Med gråt er markeret de punkter i planen, som regnes som passeret. I denne situation vil alle punkterne på γ blive regnet som passeret, selvom det rent faktisk kun er p_i og p_{i-1}, der er passeret.

Artiklen specificerer ikke, hvordan man vælger et nyt punkt på kurven. Vores løsning er, at hver gang et punkt p_i på kurven er passeret, vælges det næste punkt p_{i+1} som det punkt på kurven, som befinder sig robottens længde (L) længere ude på kurven. Dette giver en rimelig fornuftig, men dog lidt slingrende, kørsel, idet der opstår en diskontinuitet i styrevinklen, hver gang et nyt styrepunkt vælges.

På figur 21 ses positionsafvigelsen, når en robot følger testkurven på figur 13 med successiv punktstyring. Som det ses, er resultatet sammenligneligt med resultatet fra P-reguleringen. Dog er successiv punktstyring mere stabil: Den går ikke i ”selvsving”, men til gængæld følges ikke-lige dele af kurven mindre præcist, pga. de gentagne valg af nye styrepunkter.

4.5.4 Valget af kurvefølge-algoritme

Når der skal vælges en algoritme til at følge en kurve, er det en afvejning af flere parametre: Hvor nøjagtigt ønskes det at kurven følges, hvor hurtigt skal robotten



Figur 21: Robotten følger testkurven i figur 13, med en fart på 0,75 m/s, ved hjælp af successiv punktstyring.

køre undervejs og hvor tolerant overfor afvigelser skal algoritmen være. Generelt kan siges, at den naive algoritme beskrevet i afsnit 4.5.1 er for begrænset i sin virkemåde til at fungere i praksis. Valget falder derfor på enten P(ID)-reguleringen, jævnfør afsnit 4.5.2, eller på den successive punktstyring, jævnfør afsnit 4.5.3. Her er P(ID)-reguleringen den mest præcise, men det er begrænset, hvor hurtigt robotten må køre, når kurven følges. Den successive punktstyring er mindre nøjagtig, til gengæld er den ikke så følsom over for robottens hastighed eller overfor udefrakommende påvirkninger, som kan få robotten til at afvige fra kurven.

4.6 Hastighedsregulering

Grundet den meget simple modellering af robottens fremdrift (motor), jævnfør afsnit 2.2.1, er en hastighedsregulering meget enkel at konstruere. Ønskes en vis hastighed, v_w, sættes robottens acceleration a som:

$$a = \begin{cases} a_v & \text{hvis } v_R < v_w \\ -a_v & \text{hvis } v_R > v_w \\ 0 & \text{hvis } v_R = v_w \end{cases} \tag{76}$$

Her er a_v den acceleration man ønsker robotten skal accelerere med, og v_R er robottens hastighed.

Havde robotten været en fysisk robot, ville modelleringen i afsnit 2.2.1 bestemt ikke have været tilstrækkelig; den er blot en praktisk grov tilnærmelse, og hastighedsreguleringen beskrevet ved formel 76 ville slet ikke være tilstrækkelig. Her kunne så i stedet med fordel vælges en PID-regulering, se [Ole Jannerup, 2000, side 292-296].

4.7 Løsningen af position-til-position bevægelsesproblemet

Position-til-position bevægelsesproblemet er blevet løst i dette kapitel. Først findes den korteste kurve mellem de to positioner, som det er muligt for robotten at følge, som beskrevet i afsnit 4.2 og 4.3. Ud fra positionbestemmelsen beskrevet i afsnit 4.4 kan kurven følges, ved hjælp af en af kurvefølgningsalgoritmerne beskrevet i afsnit 4.5.

En enkelt, men vigtig, detalje mangler dog: Valget af ρ_{min} ved udregningen af den korteste kurve. Vælger man at bruge ρ_{min} som beregnet ved formel 40 har kurvefølgningsalgoritmen ikke noget plads at styre indenfor. Når robotten når kurvens første sving, vil det tage forhjulene lidt tid at dreje til den maksimale vinkel, hvilket vil føre til, at robotten kører lidt ved siden af kurven. Dette vil kurvefølgningsalgoritmen ikke kunne rette op på før svinget er færdigt. Er dette det sidste sving på kurven vil robotten derfor ikke nå den ønskede slutposition.

Heldigvis kan der let rettes op på dette problem. Man kan blot bruge en lidt større ρ_{min} til kurveberegningen end den ved formel 40 beregnede. For eksempel kan vælges en 10% større ρ_{min}. Robotten følger så ikke den absolut korteste rute, men til gengæld ender robotten i den ønskede slutposition. For en afprøvning af denne løsning se kapitel 6.1.

En noget mindre hensigtsmæssig løsning er at lade robotten stoppe helt op hver gang kurven ændrer krumning, for derefter at lade den rette forhjulene til den nye krumning. Dette er ikke en realistisk løsning, da robotten vil bruge alt for meget energi på konstant at bremse helt op, for blot derefter at accelerere igen. Desuden vil der være meget mere friktion end ved kørsel, når hjulene drejer i den virkelige verden.

5 Implementering

Dette afsnit omhandler implementeringen af vores løsning. Vi har brugt en del tid på at udvikle et *simulator-framework*, som understøtter flere forskellige typer af simulatorer, der alle kan foretage simulationer af bil-agtige robotter. Afsnittet er ret teknisk orienteret, og forklarende nærmere end analyserende. Vi føler dog, det er vigtigt, at vi forklarer, hvordan vi har designet og implementeret de forskellige dele grundigt, og derfor kan dette afsnit måske forekomme en smule tungt og teknisk. De forskellige dele gennemgås så vidt muligt i samme rækkefølge som de tidligere er blevet introduceret i denne rapport.

Programkoden er skrevet i Java 1.4.2 (J2SE)[3], og alle essentielle dele (klasser) er fuldt ud dokumenteret via JavaDoc[4]. Der er udviklet over 180 klasser/interfaces! Såvel programkoden som dokumentation er på engelsk, og kan forefindes på den denne rapport vedlagte CD under henholdvis `/kildekode` og `/dokumentation`. Dette afsnit forudsætter kendskab til OOA/OOP samt Java. Det kan for læseren meget vel være en god ide at have JavaDoc'en ved hånden, når dette afsnit læses. Den giver for eksempel et godt overblik over pakke- og klassehierarkiet.

Applikationen er udviklet i *open-source* værktøjet Eclipse 3[5]. Således kan både udviklingen og afviklingen af applikationen foregå som minimum under både Windows og Unix/Linux. Der kræves ingen trediepartssoftware for at afvikle applikationen, men Java JRE 1.4.2 skal være installeret på maskinen, hvor applikationen skal afvikles.

5.1 Pakkeoversigt

Tabel 1 viser den overordnede pakkestruktur for Java-kildekoden i alfabetisk rækkefølge; underpakker er ikke inkluderet i oversigten, men kan ses på den den vedlagte CD; for eksempel oversigten genereret af JavaDoc. De enkelte pakker vil blive forklaret nærmere i afsnittene refereret i yderste højre kolonne.

Pakke	Beskrivelse	Afsnit
`dk.diku.robotsim.configuration`	Konfigurering af simulator, simulation, robot samt simulationslogger.	5.7

[3] Se `http://java.sun.com/j2se/1.4.2/index.jsp`.
[4] Se `http://java.sun.com/j2se/javadoc/index.jsp`.
[5] Se `http://www.eclipse.org/`.

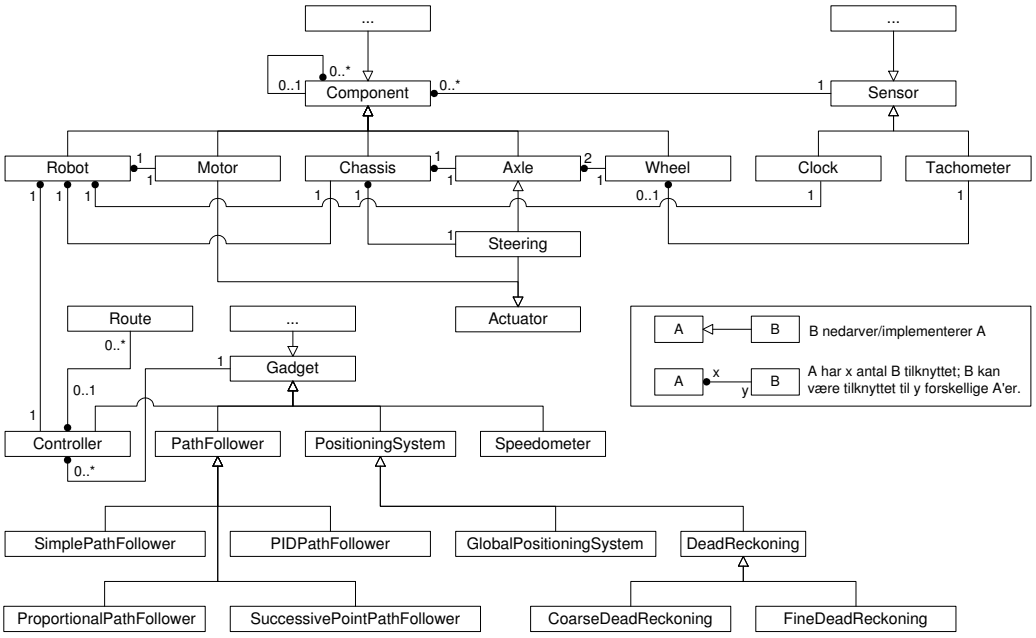
`dk.diku.robotsim.implementation.*`	Implementering af de dele af modellen som er uafhængig af simulationen, men som bruges af robotten. For eksempel implementering af *stifølgere*, *speedometer*, *positionsbestemmelsessystemer*, etc.	5.2.4
`dk.diku.robotsim.logger.*`	Generering, formattering og opbevaring af simulationsdata.	5.4
`dk.diku.robotsim.model.*`	Modellen af en bil-agtig robot og dens komponenter. Dette udgør det offentlige interface til og fra omverden, og en robot kender kun til dette.	5.2
`dk.diku.robotsim.route`	Løsning af korteste rute problemet.	5.5
`dk.diku.robotsim.simulation.*`	Implementering af de dele af modellen, der direkte indgår som en del af simulationen, for eksempel robotten, samt løsningsmodeller til at løse den fysiske model.	5.2.5, 5.3
`dk.diku.robotsim.simulator.*`	Simulatorer og grafiske brugergrænseflader til samme.	5.6
`dk.diku.robotsim.util`	Diverse hjælpeklasser og komponenter.	–
`/resources`	Standardresursebibliotek, hvor konfigurationsfiler læses fra.	–

Tabel 1: Tabellen viser den overordnede pakkeoversigt for den udviklede Java kildekode.

5.2 Modellen

Designet skiller klart mellem klasser, der bruges til at definere den fysiske model af robotten, og dem der bruges i simulationen. Modellen er defineret ved brug af *interfaces*, og har således ingen offentlig (eng: public) implementering eller tilstand. Figur 22 viser modellen, og de forskellige dele og tilknytninger forklares i det følgende.

Modellen definerer de fysiske *komponenter*, der som minimum er en del af en bil-agtig robot “i den virkelige verden”, såsom chassis, motor, hjul, aksler osv., samt hvordan de er koblet sammen. Eksempelvis har robotten et chassis, der har en motor og to aksler, og hver aksel har tilknyttet to hjul.



Figur 22: Overordnet oversigt for modellen som den er defineret i Java. En komponent er en fast, fysisk del af en robot eller robotten selv, og kan have andre komponenter tilknyttet. For eksempel har chassiset to aksler tilknyttet, som hver har to hjul. En komponent er repræsenteret ved `dk.diku.robotsim.model.Component` model-interfacet. En komponent kan klassificeres som en aktuator, såfremt den kan påvirke robotten med en kraft. En aktuator er repræsenteret ved `dk.diku.robotsim.model.Aktuator` klassen. Robotten har for eksempel tilknyttet en motor, som er en aktuator. En komponent kan have sensorer tilknyttet, som leverer input til robottens controller. En sensor er repræsenteret ved `dk.diku.robotsim.model.Sensor` klassen. Robotten har et ur tilknyttet, og hvert hjul på akslen har et tachometer tilknyttet. Controlleren styrer robotten, og bruger dertil gadgets, såsom speedometer, stilfølgere, etc; se eventuelt afsnit 5.2.4. En controllers mål vil ofte være at følge en rute, måske bestående af flere del-optimale kurver. En sådan rute er repræsenteret ved `dk.diku.robotsim.route.Route` klassen (se afsnit for 5.5 uddybning). En gadget er repræsenteret ved `dk.diku.robotsim.model.Gadget` interfacet, og controlleren er også en gadget. Der kan implementeres nye gadgets (samt komponenter og sensorer) “efter behov”; figureren viser vores implementeringer af speedometer, positionsbestemmelsessystemer og stifølgere. Endvidere er en mængde af controllere blevet implementeret, men de er ikke vist på figuren.

Modellen klassificerer endvidere specifikke komponenter som *aktuatorer*. Aktuatorer er komponenter, der kan påvirke robotten med en fysisk kraft, for eksempel en motor.

En komponent kan have tilknyttede *sensorer*. Disse er ikke klassificeret som komponenter, men som en selvstændig gruppe. En sensor kan forsyne robotten med input fra omgivelserne; det være sig et ur (på robotten), et tachometer (på et hjul) eller noget helt tredie.

Komponenterne og sensorerne er velkendte af robotten via de faste interfaces i modellen, som dog jo naturligvis varierer fra komponent/sensor til komponent alt efter dens fysiske egenskaber. Men modellen definerer yderligere to vigtige dele: *controllere* og *gadgets*. En controller modsvarer robottens styringslogik i den fysiske verden, dvs. det er controlleren, der aflæser sensorer, bearbejder data og på bagrund af dette træffer beslutninger om, hvad robotten skal fortage sig, for eksempel ved at bruge sine aktuatorer til at bevæge sig. Controlleren dækker over både CPU (hardware) og robottens program (software). En gadget kan vel bedst oversættes med noget i stil med ”en elektronisk dims”, hardware såvel som software, og kan bruges af controlleren til at uddelegere arbejde (faktisk er controlleren selv en gadget i modellen). Gadget er altså et generisk begreb og har et fast interface, som skal implementeres af specifikke implementeringer Dette sikrer, at modellen ikke nødvendigvis behøver at kende alle gadgets, idet de tilgås via det samme interface altid, og de kan derfor tilføjes eller fjernes efter den specifikke controllers behov. Eksempler på implementerede gadgets er alle controllere, *stiføIgere*, positionsbestemmelsessystemer, speedometer, osv. Det betyder samtidigt, at gadgets er fuldt ud implementerede (og strengt taget ikke en del af den egentlige model), og controlleren kan tilgå gadgets via gadget-interfacet (igennem modellen) eller via den specifikke implementering, idet de rent sprogteknisk er tilknyttet controlleren som en attribut.

5.2.1 Komponenter

Der er oprettet en interfacespecifikation for hver komponent, som alle nedarver fra `dk.diku.robotsim.model.Component`. Komponenter skal registreres hos deres forældrekomponent. Det gør det for eksempel muligt at hente hele eller dele af “komponenttræet” for en given komponent (for eksempel robotten), hvilket gør det nemt for simulatoren at præsentere information om, hvordan robotten er opbygget, bestemme hvilke komponenter der skal tegnes, etc. Men `Component`-klassen specificerer ikke decideret komponentfunktionalitet, da den selvsagt varierer meget fra komponent til komponent.

Vi har endvidere implementeret standard abstrakte klasser, som implementerer

det mest trivielle fra `Component`, så en reel implementering kan nedarve fra disse klasser og bekymre sig om den virkelige komponentfunktionalitet.

5.2.2 Aktuatorer

Motor Motoren skal implementere ligning 1 ved simple "getter"- og "setter"-metoder, dvs. at accelerationen kan sættes og aflæses på klassen. Motoren er repræsenteret ved interfacet `dk.diku.robotsim.model.component.Motor`.

Servo Chassiset har tilknyttet to aksler, og de skal implementeres ved interfacet `dk.diku.robotsim.model.component.Axle`. Den forreste aksel styrer robotten, mens den bagerstes hjul indeholder tachometerne. Styringen er defineret ved interfacet `dk.diku.robotsim.model.component.Steering` som nedarver fra `Axle`. Drejningshastigheden, som angivet i ligning 2, kan sættes og aflæses fra klassen.

5.2.3 Sensorer

Vi har kun udviklet to sensorer indtil videre: et ur og et *tachometer*. Sensorer tilknyttes og kan senere hentes via generiske metoder i `Component`-interfacet.

Tachometer Interfacet `dk.diku.robotsim.model.sensor.Tachometer` definerer et tachometer, og kan returnere antal ticks og den tilbagelagte afstand svarende til ligningerne henholdsvis 24 og 25.

5.2.4 Gadgets

Som sagt, så er det controllerens opgave at styre robotten. En controller er repræsenteret ved interfacet `dk.diku.robotsim.model.Controller`. Det betyder samtidigt, at den samme robot kan bruge forskellige controllere, som igen kan bruge forskellige gadgets, og derved opnå forskellige adfærdsmønstre. En gadget er repræsenteret ved klassen `dk.diku.robotsom.model.Gadget`, som `Controller` nedarver fra. Gadgets skal registreres til controlleren. Det er vigtigt at fastslå, at controlleren kun kender til model-interface- specifikationerne samt implementerede gadgets, men ikke til en specifik implementering af modellen, og således ikke ved om den opererer i en simulereret verden eller ej. I princippet kan en controller implementeret til at styre en simuleret robot uden ændring også styre en fysisk robot.

Der er dele af modellen, der er implementeret som abstrakte klasser, så en ny implementering ikke behøver at bekymre sig om de mere trivielle ting; men det har ingen betydning for controlleren.

De vigtigste metoder for enhver gadget er `initialise` og `update`. Den første initialiserer den pågældende gadget (eller controller), hvor den anden sørger for at "udføre sit arbejde" og opdatere sin interne tilstand. Begge metoder kaldes automatisk fra simulationen på "de korrekte tidspunkter", jævnfør afsnit 5.6.3.

Vi har implementeret en "standardcontroller" som let kan nedarves via klassen `dk.diku.robotsim.implementation.controller.DefaultController`, hvorved man kun behøver at skrive få liniers kode for at have en fuldt funktionsdygtig controller. Faktisk er det nok at implementere `Controller`-metoden `update`[6], men det kan også være nødvendigt at foretage specifik initialisering. Vi har endvidere tilknyttet nogle faste gadgets til standardcontrolleren. Disse har vi implementeret, idet vi har fundet dem yderst nyttige for en vilkårlig controller:

- Et positionsbestemmelsessystem ved klassen `PositioningSystem`, som er den abstrakte super-klasse, der bruges af alle vores positionsbestemmelsessystemer;
- En stifølger ved den abstrakte klasse `PathFollower`, som er super-klassen for alle vores stifølgere; samt
- Et speedometer ved den fuldt implementerede `Speedometer`-klasse.

Alle tre klasser findes i pakken `dk.diku.robotsim.implementation.gadget`. Idet positionsbestemmelsessystemet og stifølgeren er abstrakte, kan der tilknyttes vilkårlige implementeringer af dem til en (standard) controller. Vi har implementeret samtlige metoder nævnt i afsnit 4.4, begyndende på side 29, som forskellige typer positionsbestemmelsessystemer, og endvidere alle metoder nævnt i afsnit 4.5, begyndende på side 33, som forskellige typer stifølgere.

Nedenstående programkode er et eksempel, der viser en controller, som kan følge en given rute bestående af X antal punkter, hvor den korteste vej mellem to på hinanden følgende punkter følges, indtil det sidste punkt er nået. Når det er tilfældet, vil controlleren rapportere til robotten via sin interne tilstand, at den har afsluttet sin opgave. Controlleren starter med at accelerere indtil den har nået 0.35 m/s, hvorefter den holder denne fart.

```
public class SimpleController extends DefaultController {
```

[6] Faktisk nedarver `Controller`-metoden fra `Gadget`.

```
  private boolean accelarate;
  private int index;

  public SimpleController() {
    super(new PIDPathFollower(new FineDeadReckoning()));
  }

  public void initialise() {
    super.initialise();
    this.accelerate = true;
    this.index = 0;
  }

  public void update() {
    // Accelarate?
    if (this.accelarate) {
      this.accelarate = false;
      this.robot.getMotor().setAcceleration(0.5);
    }
    // Has cruise speed been reached?
    if (this.speedometer.getSpeed() > 0.35) {
      this.robot.getMotor().setAcceleration(0);
    }

    // Is the path follower done with the current path
    // between two points?...
    GadgetState state = this.pathFollower.getState();
    if (state == GadgetState.COMPLETED) {
      if (++this.index < this.route.size()) {
        // Get next optimal path between the next set of points...
        this.pathFollower.setPathSolutionSet(this.route.get(this.index));
      } else {
        // Controller is done...
        this.state = GadgetState.COMPLETED;
      }
    }
  }

}
```

Controlleren vil naturligvis være blevet initialiseret med de angivne punkter via simulationen, som derved også har sat et `Route`-objekt på controlleren repræsenterende ruten, som i standardimplementeringen kan tilgås via den nedarvede attribut `route`. Alternativt kan controlleren i sin `initialise`-metode selv specifikt bestemme den eller de ruter, den vil følge.

Der er ingen begrænsning på, hvor mange gadgets en controller kan have tilnyttet, eller af hvilke typer. Det er dog hensigtsmæssigt kun at have for eksempel én aktiv stifølger tilknyttet ad gangen. Det tillader nemlig, at simulatoren kan forespørge, om controlleren har en aktiv gadget af en given type tilknyttet, og i så fald for eksempel udskrive dens offentlige tilstand. Eksempel:

```
// Get active simulation...
Simulation simulation = Simulator.getInstance().getSimulation();
// Fetch controller...
```

```
Controller controller = simulation.getRobot().getController();
// See if at least one active positioning system is attached...
List systems = controller.getGadget(PositioningSystem.class);
for (int i = 0; i < systems.size(); i++) {
  PositioningSystem ps = (PositioningSystem)systems.get(i);
  if (ps.getState() == GadgetState.ACTIVE) {
    // Got it...
  }
}
```

Dette tillader endvidere at en controller kan skifte mellem stifølgere alt efter robottens hastighed, for eksempel.

5.2.5 Simulering

For at kunne foretage simuleringer er en implementering af modellen nødvendig, idet modellen kun er defineret som interfaces. Det er som sagt kun disse interface robotten og controlleren (og gadgets) kan gøre godt med, hvori mod simulatoren og simulationen kender den reelle implementering af modellen, og således også kan forespørge eller sætte tilstanden for hele eller dele af implementeringen. Under pakkerne `dk.diku.robot.simulation.model.*` er modellen implementeret til brug i vores simulationer. Model-interfacet `dk.diku.robotsim.model.Robot` implementeres af `dk.diku.robotsim.simulation.model.SimulatedRobot`, chassis-komponenten `dk.diku.robotsim.model.component.Chassis` implementeres af `dk.diku.robotsim.simulation.model.component.Chassis`, osv.

Fysiske komponenter samt sensorer er defineret som offentlige, faste (`final`) attributter af specifik implementeret type, som dog kun kan tilgås af den fysiske robot via det generiske modelinterface. Det betyder, at simulationen har mere information til rådighed end modellen, idet den kender de implementerede klasser, og ikke kun modelinterfacene. Eksempel:

```
// In controller (robot = Robot)
Steering steering = getRobot().getChassis().getSteering()
..

// In simulation (robot = SimulatedRobot)
SimulatedSteering steering = robot.chassis.steering;
// or using model interface...
SimulatedSteering sterring = (SimulatedSteering)robot.getChassis().getSteering();
```

Via modellen har en robot et chassis ved typen `Chassis` tilknyttet, som igen har et `Steering`-objekt tilknyttet. Robotten er implementeret ved `SimulatedRobot`, chassiset ved `SimulatedChassis`, og styringen ved `SimulatedSteering`. Den implementerede robot har altså implementeringen af chassiset tilknyttet som en offentlig, fast attribut, idet dette er gældende for alle robotter; chassiset har styringen

tilknyttet som en offentlig, fast attribut, idet det er gældende for alle chassiser, etc. Modelimplementeringerne beskytter derudover alle sine attributter (ikke-offentlige), og de skal derfor tilgås som normalt via “gettere” og “settere”.

Implementeringen sørger naturligvis også for, at den tilstand som er tilgængelig via modellen, er indeholdt i de implementerede klasser. Andre nødvendige interfaces implementeres ligeså, så som for eksempel interfacet, der dikterer, at et objekt kan simuleres ud fra et sæt af tilknyttede differentialligninger (se næste afsnit), samt interfaces, der bruges af en simulator til at rendere, dvs. tegne, komponenten.

5.3 Numeriske løsningsmetoder

Som beskrevet i afsnit 3.1 på side 11, så løser vi modellen, dvs. de opstillede differentialligninger, numerisk. Et *objekt* med et sæt af sådanne differentialligninger tilknyttet, er specificeret ved `dk.diku.robotsim.simulation.solver.Solvable` interfacet; det betyder naturligvis, at *implementeringen* af robotten implementerer dette interface. Interfacet dikterer, at det er muligt at forespørge på den nuværende tilstand (`getStateVector`) samt på den afledede tilstand til et givet tidspunkt (`getDerivedStateVector`). Endvidere er det muligt at sætte objektets interne tilstand (med metoden `setStateVector`). Endelig har interfacet en `update`-metode og ved kald til denne, skal implementeringen give robottens controller lov til at opdatere sin interne tilstand, og for eksempel de variable, som beskriver robottens aktuatorer. Disse er variablerne a og ϕ_w for den bil-agtige robot.

Den i afsnit 2.4 beskrevne model af den bil-agtige robot, som ønskes simuleret, er implementeret ved `dk.diku.robotsim.simulation.model.SimulatedRobot`, som følgeligt implementerer `Solvable`-interfacet. Robottens tilstand er beskrevet ved de tilstandsvariable, hvis afledede er bestemt af ligningerne fra afsnit 2.4. De kan eksporteres til en tilstandsvektor med metoden `getStateVector`:

$$(t, v_R, \phi, \theta, x_R, y_R, d_R, d_{left}, d_{right}) \tag{77}$$

Her betegner t tiden. Metoden `getDerivedStateVector` udregner og eksporterer den afledte tilstandsvektor ved at implementere ligningerne fra afsnit 2.4:

$$(\dot{v_R}, \dot{\phi}, \dot{\theta}, \dot{x_R}, \dot{y_R}, \dot{d_R}, \dot{d_{left}}, \dot{d_{right}}) \tag{78}$$

Løsningen af ligningerne foregår ved brug af en valgt numerisk løsningsmetode. Vi har implementeret begge de nævnte løsningsmetoder i henholdsvis afsnit 3.1.1 og 3.1.2, dvs. Euler og 2. ordens Runge-Kutta. Løsningen af ligningerne sker ved, at robottens tilstandsvektor hentes med `getStateVector`.

Den fremskrives så ud fra den afledte af tilstandsvektoren, som hentes med `getDerivedStateVector`. Når den fremskrevne tilstandsvektor er beregnet, sættes robottens tilstand med `setStateVector`. Den anvendte numeriske løsningsmetode er uafhængig af den implementerede model, så løsningsmetoden kan vælges på køretidspunktet. Alle løsningsmetoder implementerer interfacet `dk.diku.robotsim.simulation.solver.Solver`, som dikterer at alle løsningsmetoder **skal** sørge for følgende:

1. At fremskrive simulationstiden med faste skridtstørrelser, som beskrevet ovenfor.
2. At robottens controller får lov til at køre, dvs. dens `update`-metode kaldes, nævnt i det ovenstående.
3. At sikre at ligningerne løses for robotten.
4. At logge simulationsdata.

Det er endvidere muligt at angive et forhold (`ratio`) mellem antallet af gange ligningerne løses på ny, inden controllerens `update`-metode kaldes (som standard 5:1). Simulationstiden opdateres i `solve`-metoden, specifikt alt efter hvilken løsningsmodel der er valgt.

Følgende pseudokode viser, hvorledes ovenstående punkter kan programmeres i en `Solver`-implementering, idet det antages at `time` er simulationstiden på et givent tidspunkt, `advanceTime` er den tid simulationstiden skal fremskrives (parameter), `delta` er den faste tidsskridtstørrelse, `ratio` angiver forholdet mellem opdatering og løsning på ny, og `solvable` er robotten med tilknyttede ligninger:

```
advanceTime += timeReminder;
while (advanceTime >= delta) {
  if (pending == 0) {
    // Update robot -> update gadgets and controller
    solvable.update();
    pending = ratio;
  }
  // Equations are solved numerically
  solve(solvable);
  // Log simulation data
  logger.log(time);
  advanceTime -= delta;
  pending--;
}
timeReminder = advanceTime;
```

Da den simulerede robot implementerer `Solvable` er det selv sagt dens `update`-metode der kaldes. Den sørger for at opdatere sine tilknyttede gadgets samt controlleren. Simulationstidsskridtet (`delta`) er sat til 10 ms som standard, og ved

standardforholdet 5:1, svarer det til, at controlleren kaldes 20 gange i sekundet. Dette svarer til frekvensen brugt på robotten Murphy. Logning af simulationsdata er beskrevet i afsnit 5.4. Figur 27 i afsnit 5.6.3 på side 66 viser en oversigt over det samlede *control-flow* ved afvikling af simulationer.

For at enhver ny `Solver` ikke behøver at tage højde for ovenstående, har vi implementeret en abstrakt løsningsmetode ved klassen `AbstractSolver`; begge vores løsningsmetoder nedarver fra den.

5.4 Simulationsdata

Komponenter og gadgets ("robotdele") *kan* reportere hele eller et udsnit af deres interne tilstand, mens en simulering kører. Det er op til den enkelte del at bestemme, hvad og hvor meget, den vil rapportere. Rapporteringen indeholder endvidere metadata om, hvad der reporteres. Metadata og tilstanden udgør tilsammen *simulationsdata* for én given del på ét bestemt tidspunkt i simuleringen. Et eksempel på simulationsdata kan være for implementeringen af et tachometer (sensor), som returnerer den tilbagelagte afstand i både ticks og meter: Metadata er da ('Afstand', 'Ticks') og ('Afstand', 'Meter'), og data er (*afstand i ticks*, *afstand i meter*). Simulationsdata bruges **ikke** af modellen, men kun til at rapportere tilstanden til eksternt brug, for eksempel aflæsning eller videre bearbejdning såsom grafgenerering.

Interfacet `dk.diku.robotsim.logger.Loggable` angiver, at alle implementerende klasser rapporterer deres tilstand på samme måde. Rapporteringen foretages ved brug af en *simulationslogger*, som **kun** bruges til at rapportere simulationsdata, og er repræsenteret ved `dk.diku.robotsim.logger.Logger` klassen. Til en given simulationslogger registreres de dele ("loggables"), der skal rapporteres for. Mens simulationen kører, sørger loggeren for at indsamle data fra alle registrerede dele, enten i hvert simulationstidsskridt, eller per et givet interval ("sampling"). Loggeren kaldes automatisk fra den brugte numeriske løsningsmodel. Figur 24 illustrerer bl.a. hvorledes simulationsloggere er opbygget, og figur 27 viser, hvor og hvornår en simulationslogger bruges.

Formatteringen og opbevaring af simulationsdata er adskilt. Hver simulationslogger har en bestemt formattering tilknyttet, som bestemmer, hvordan simulationsdata skal formatteres. Det kan for eksempel være en formattering, der formatterer data, så de kan læses af Matlab og danne baggrund for grafgenerering eller et mere let-læseligt format, som umiddelbart kan læses med menneskeøjne. Opbevaring af formatterede simulationsdata sker i en eller flere buffere tilknyttet loggeren. Det kan for eksempel være én som skriver til en fil, og/eller én som skriver til `stdout`, og/eller én som skriver til en *brugergrænsefladelogger* som beskrevet

i afsnit 5.6.1. Første gang der logges til en buffer, vil der automatisk blive genereret kommentarer i loggen angående simulationen, såsom hvilken løsningsmetode der bruges, samt endvidere hvilke objekter der logges data for, deres metadatainformation, mm. Dette sikrer, at man alene via loggen har nok information til fuldstændigt at genskabe simulationen.

Endeligt kan loggere tilknyttes hinanden i en hierarkisk struktur. Det kan være nødvendigt, hvis man ønsker at rapportere forskellige dele uafhængigt af hinanden, hvis der skal bruges forskellig *sample-rate*, eller hvis man ønsker at generere uddata i forskellige formater samtidigt.

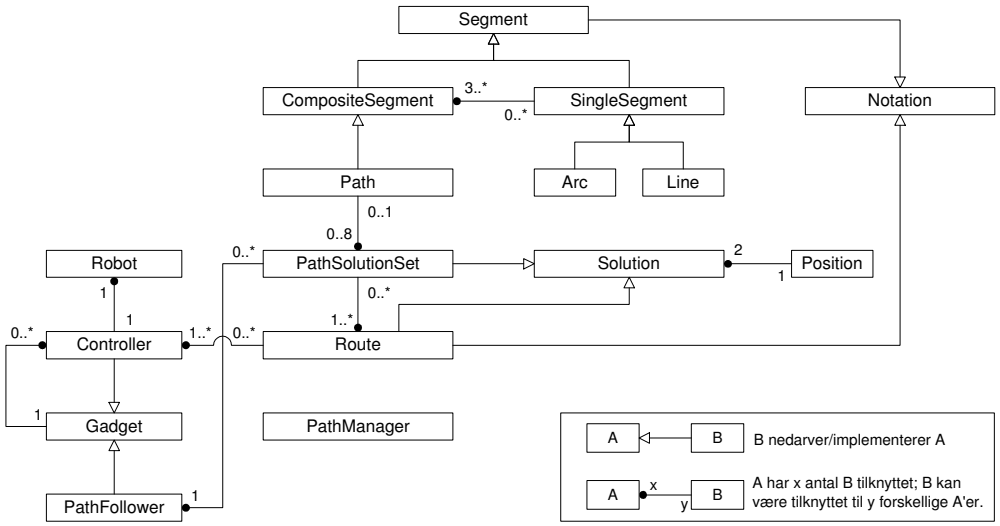
5.5 Korteste rute

Pakken `dk.diku.robotsim.route` indeholder de klasser, der bruges til at finde den korteste rute mellem to positioner, med begrænset mindste krumningsradius, ud fra metoden, som blev opsummeret i afsnit 4.3.7 på side 28. En oversigt over vores implementering kan ses på figur 23.

Det vil typisk være en controller eller en simulator, der bruger disse klasser; en controller til at følge ruten (via en stiføl ger), og simulatoren til eventuelt at initialisere controlleren.

For at finde sættet af mulige ruter mellem to eller flere på hinanden følgende positioner, og en given begrænset mindste krumningsradius, bruges klassen `dk.diku.robotsim.route.PathManager`. Herunder vises et eksempel på anvendelsen:

```
..
Position start  = new Position(1.0, 0.5, 0.9);
Position end    = new Position(2.0, 1.2, 1.5);
double   radius = 1.0;
..
PathSolutionSet paths = PathManager.findOptimalPath(start, end, radius);
Path optimal = paths.getOptimalPath();
if (optimal != null) {
  // Optimal...
}
// Number of solutions between start and end...
int size = paths.size();
..

List positions = ..
positions.add(start);
positions.add(end);
positions.add(new Position(3.0, 1.7, 0.76));
..

// From start -> end, then end -> (3.0, 1.7, 0.76), etc.
Route route = PathManager.findOptimalRoute(positions, radius);
```



Figur 23: Oversigt over implementeringen til at finde den korteste rute i pakken `dk.diku.robotsim.route`. Et segment, dvs. et liniestykke er en cirkelbue, er repræsenteret ved `Segment`-interfacet. Et segment kan være indeholdt i et `CompositeSegment`. En kurve indeholdende tre segmenter, svarende til ord som beskrevet i afsnit 4.3.1, er defineret ved klassen `Path`. Det kan for eksempel være ordet lsr svarende til en cirkelbue, en linie og en cirkelbue. Kurven er ikke nødvendigvis en optimal kurve mellem to positioner. Et `PathSolutionSet` (løsningssæt) indeholder alle løsninger givet en start- og slutposition og en mindste krumningsradius; endepositionen er implicit indeholdt i repræsentationen, da alle segment længer kendes. Der vil maksimalt være otte `Path` objekter indeholdt i et `PathSolutionSet`. Endeligt indeholder et `Route` ("samlet rute") objekt et vilkårligt antal løsningsæt, hvor startpositionen for det i'te løsningssæt er slutpositionen for det $i-1$'te løsningssæt. Både løsningssæt og rute objekter beregnes ved hjælp af singletonklassen `PathManager`. Det vil ofte være en controller, der via en tilknyttet stifølger, bruger en rute. Stifølgeren vil da bruge et løsningssæt valgt af controlleren fra den rute den vil følge.

```
if (route != null) {
  for (Iterator i = route.iterator(); i.hasNext(); ) {
    PathSolutionSet paths = (PathSolutionSet)i.next();
    ..
  }
}
..
```

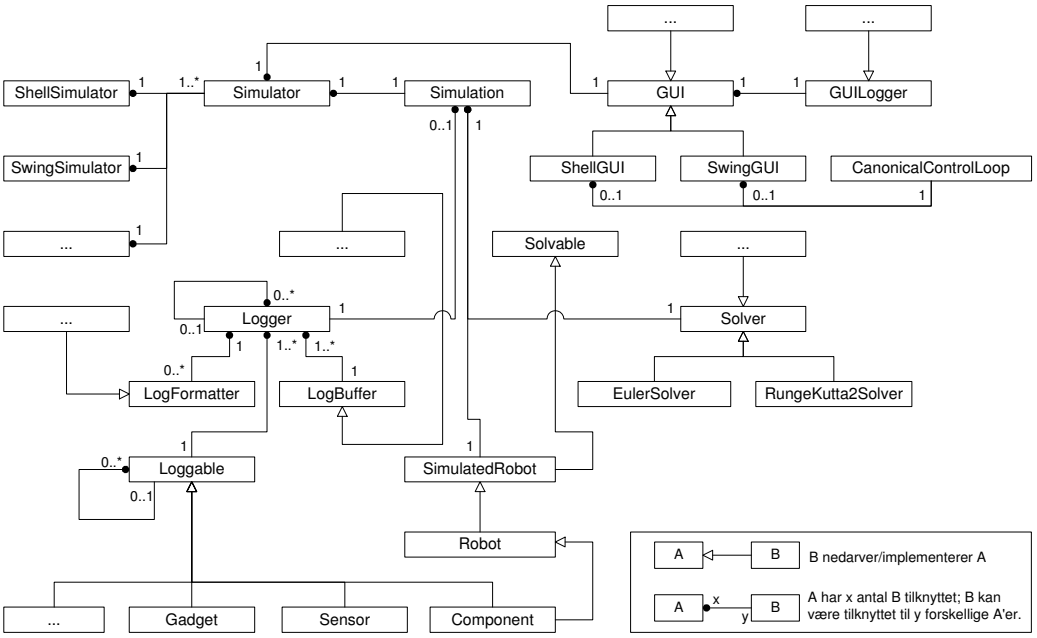
Klassen `dk.diku.robotsim.route.PathSolutionSet` repræsenterer de mulige løsninger mellem to positioner (maksimalt otte, jævnfør det ovenfor refererede afsnit), og `dk.diku.robotsim.route.Route` indeholder en ordnet rækkefølge af sådanne løsningssæt. På et løsningssæt kan alle specifikke løsninger hentes og er defineret ved klassen `dk.diku.robotsim.route.Path`. Det er muligt at udskrive løsningerne med både Dubins samt Reeds og Shepp notation, som beskrevet i afsnit 4.3.1 på side 14.

`PathManager` definerer (private) metoder til at udregne kurver af formen lrl, lsl og lsr (inklusiv hjælpe funktionerne R og M), og ved brug af spejling findes alle de mulige kandidater

Denne pakke er fuldstændigt uafhængig af alle andre pakker.

5.6 Simulatorer

Klassen `dk.diku.robotsim.simulator.Simulator` repræsenterer i designet en *simulator*. Denne klasse er fast og kan ikke nedarves (erklæret `final`), men skal bruges i forskellige *simulatorapplikationer*. I en kørende applikation vil der derfor altid kun være én instans af klassen allokeret, svarende til den simulator der afvikler simulationerne. Selve klassen er blot en container, der indeholder referencer til den nuværende aktive simulation, samt til en brugergrænsefladeimplementation (engelsk: *GUI – Graphical User Interface*). Afviklingen – hvordan og hvornår – af simuleringer er uddelegeret til brugergrænsefladen. En simulatorapplikation er altså brugen af `Simulator`-klassen med én specifik implementeret og tilknyttet brugergrænseflade. Der kan oprettes flere forskellige brugergrænseflader og således flere forskellige simulatorapplikationer. Alle brugergrænseflader skal implementere interfacet `dk.diku.robotsim.simulator.gui.GUI`, og en kørende simulation er defineret ved klassen `dk.diku.robotsim.simulation.Simulation`. Figur 24 viser, hvorledes vi har implementeret forskellige simulatorer, og hvilke andre objekter der bruges.



Figur 24: Opbygning af simulatorer og afvikling af simulationer. En simulator er repræsenteret ved `dk.diku.robotsim.simulator.Simulator` klassen. En given simulatorapplikationen er en `Simulator` tilknyttet en brugergrænseflade, der kan afvikle simulationer repræsenteret ved `dk.diku.robotsim.simulation.Simulation` klassen. Til at sikre korrekt afvikling kan `dk.diku.robotsim.simulator.CanonicalControlLoop` klassen benyttes, som beskrevet i afsnit 5.6.3 og som vist på figur 27. En `Simulation` har tilknyttet den numeriske løsningsmetode der skal bruges, repræsenteret ved en implementering af `dk.diku.robotsim.simulation.solver.Solver`-interfacet. En `Solver` løser ligningerne for en instans af `dk.diku.robotsim.simulation.solver.Solvable`, svarende til `SimulatedRobot` klassen for alle vores simulatorer. Robotten er knyttet til den aktive simulation, og løsningsmetoden er ikke afhængig af `Solvable` objektet. Såfremt der er en simulationslogger tilknyttet simulationsobjektet, da vil simulationsdata blive logget til for eksempel en fil (jævnfør figur 27). Se afsnit 5.3 for uddybning vedrørende vores implementering af numeriske løsningsmetoder.

5.6.1 Brugergrænseflader

En brugergrænseflade kan startes og stoppes. Når den startes, så starter selve applikationen; brugergræsefladen vil (kan) aldrig blive startet før `Simulator`-klassen er fuldt ud initialiseret. Det er op til brugergrænsefladen at afslutte sig selv. Når det sker, så afsluttes applikationen ligeså. Desuden kan en *brugergrænsefladelogger* tilknyttes til at udskrive beskeder til brugeren, for eksempel til en konsol eller til `stdout`, alt efter brugergrænsefladens implementering Klassen `dk.diku.robotsim.simulator.gui.logger.GUILogger` er en superklasse for alle brugergrænsefladeloggere. En brugergrænsefladelogger er **ikke** er det samme som en simulationslogger som beskrevet i afsnit 5.4. Men det betyder ikke, at simulationsdata ikke kan blive logget til en konsol i brugergrænsefladen, idet `dk.diku.robotsim.simulator.gui.logger.SimulationToGUILogger` klassen er en *wrapper*, der modtager simulationsdata fra en `Logger` og føder det til en `GUILogger`.

I skrivende stund har vi implementeret to forskellige brugergrænseflader, svarende til to forskellige simulatorapplikationer:

1. `dk.diku.robotsim.simulator.gui.ShellGUI`: Afvikler simulationer fra kommandolinien, og er således kun baseret på tekst. Applikationen startes med klassen `dk.diku.robotsim.simulator.ShellSimulator`.

2. `dk.diku.robotsim.simulator.gui.SwingGUI`: Afvikler simulationer via en grafisk brugergrænseflade implementeret i SWING[7] via applikationen `dk.diku.robotsim.simulator.SwingSimulator`. Denne simulator foretager animation af simulationerne, tillader ændring af den rute robotten skal følge, konfigurering, mm.

En brugergrænseflade kan ikke umiddelbart blot oprettes, men skal konstrueres via manageren `dk.diku.robotsim.simulator.gui.GUIManager`. Nedenstående "pseudokode" svarer stort set til klassen, der starter `ShellSimulator` applikationen:

```
public class ShellSimulator {
  public static void main(String[] args) {
    // Get singleton instance of the simulator...
    Simulator simulator = Simulator.getInstance();
    // Create and attach gui...
    GUI gui = GUIManager.createGUI(ShellGUI.class.getName());
    simulator.setGUI(gui);
    // Run application...
```

[7] Se `http://java.sun.com/docs/books/tutorial/uiswing/`.

```
    gui.startGUI();
  }
}
```

Her vil applikationen afsluttes når `startGUI`-metoden afsluttes, idet der ikke bruges separate tråde til afviklingen af applikationen. I `SwingGUI`, derimod, afsluttes startmetoden med det samme, men der bruges separate tråde til afvikling, og applikationen afsluttes som følge af en *event* igangsat efter brugerens ønske.

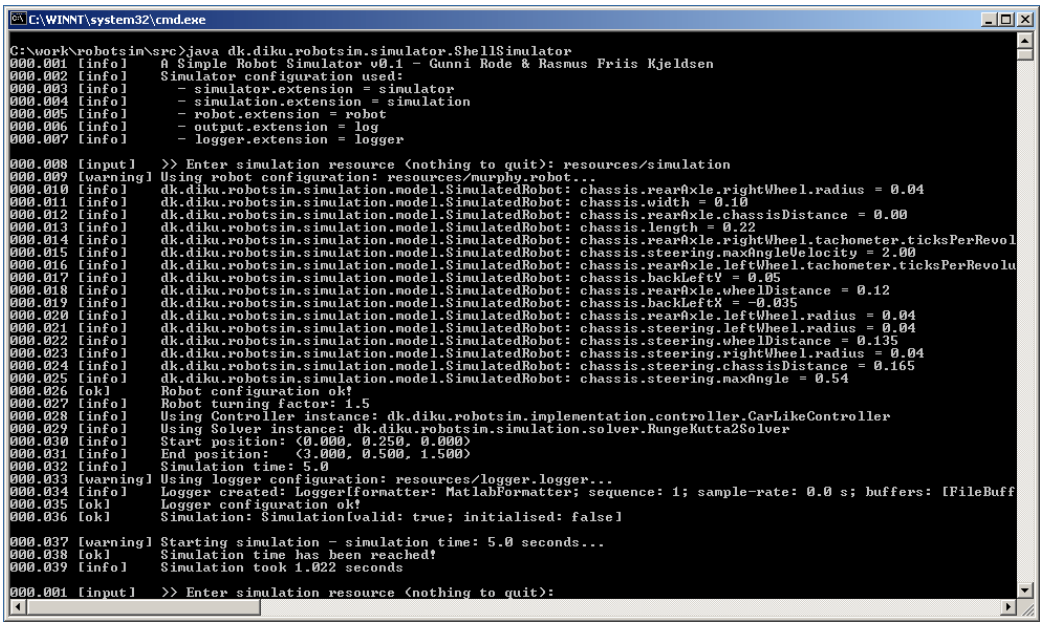
Brugergrænsefladen er altid tilgængelig via den aktive simulator. Det *kan* være nødvendigt at kontrollere for om den er `null`, idet for eksempel SWING kan aflede events *inden* brugergrænsenfladen er fuldt ud initialiseret. Først når brugergrænsefladen er fuldt ud initialiseret, vil den blive tilknyttet som vist ovenfor og derfor ikke være `null`:

```
SwingGUI gui = (SwingGUI)Simulator.getInstance().getGUI();
if (gui != null) {
  // Fully initialised...
}
```

ShellGUI Denne brugergrænseflade tillader afvikling af simulatoren direkte fra en prompt. Det eneste der behøves angives en en simulationskonfigurationsfil, hvis format er beskrevet i afsnit 5.7.2. Konfigurationsfilen kan enten angives ved en fuld sti til filen eller som et *resource*-navn; vi har vedlagt flere simulationsfiler i vores standard resource bibliotek `resources`, for eksempel konfigurationsfilen `resources/simulation.simulation`. Figur 25 viser et *screen-dump* af `ShellGUI`.

SwingGUI Denne brugergrænseflade er en komplet grafisk brugergrænseflade der bl.a. tillader brugeren at:

- Oprette, indlæse og gemme forskellige simulation konfigurationer (se afsnit 5.7.2);
- Oprette, indlæse og gemme forskellige robot konfigurationer (se afsnit 5.7.3) som kan tilknyttes en simulation;
- Oprette, indlæse og gemme forskellige simulationslogger konfigurationer (se afsnit 5.7.4) som kan tilknyttes en simulation;
- Vælge numerisk læsningsmodel;
- Vælge controller;

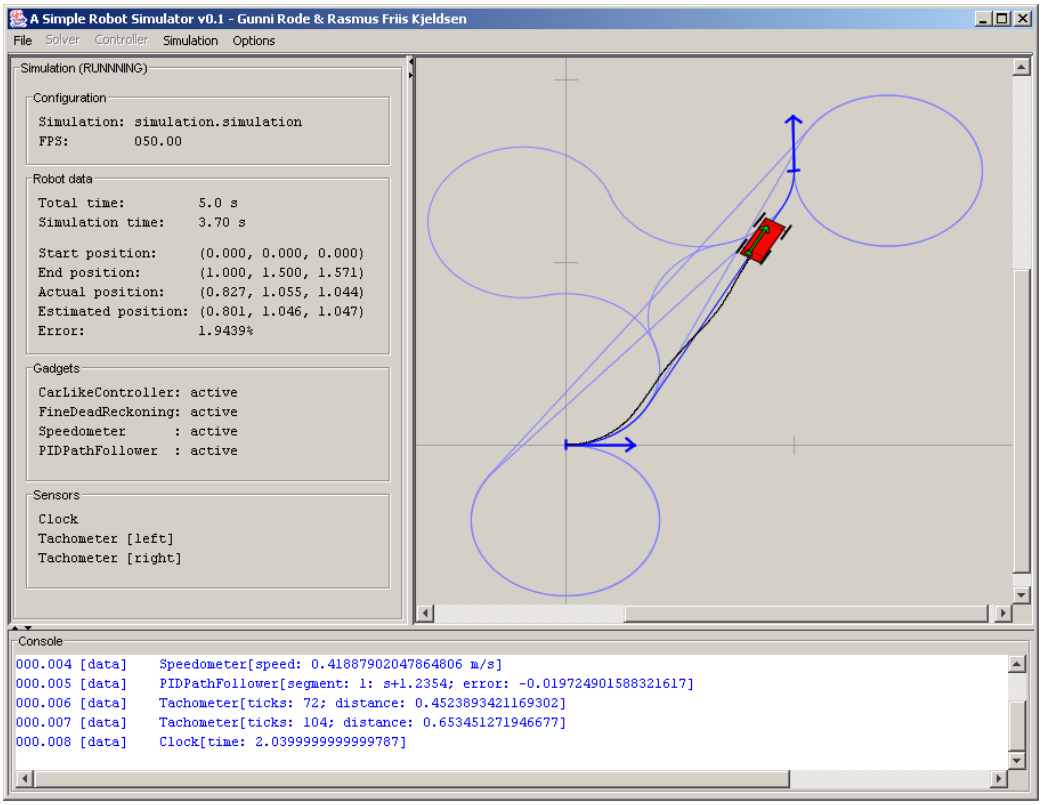


Figur 25: Screen-dump af `ShellSimulator`, klassen som bruger `ShellGUI`, under Windows 2000.

- Starte, pause og stoppe simulationer;
- Animere robotten mens simulationen kører, inklusiv spor af hvor robotten har kørt;
- Vise status for alle tilknyttede komponenter, sensorer og gadgets;
- Vælge og tegne alle løsninger robotten kan følge;
- Sætte realtids- og zoomfaktor;
- Og meget mere...

I vores standard resource bibliotek `resources` forefindes en mængde af forskellige simulations-, robot-, og loggerkonfigurationsfiler. For at starte en simulation, klikkes på *File*-menuen, og herefter på *Simulation*-menupunktet. Herved kan man angive den ønskede simulationskonfigurationsfil. Såfremt konfigurationen er fyldestgørende, klikker men derefter på menuen *Simulation*, og derefter på menupunktet *Run*, hvorfra man kan starte, pause og stoppe selve simuleringen.

Figur 26 viser et screen-dump af SWING brugergrænsefladen som den bliver brugt af `SwingSimulator` simulatorapplikationen.



Figur 26: Screen-dump af `SwingSimulator`, klassen som bruger `SwingGUI`, under Windows 2000. Mens simulationen afvikles animeres robottens færd, og robottens komponenter tegnes. Den mørkeblå kurve markerer den optimale kurve som robotten forsøger at følge fra $(0,\ 0,\ 0)$ til $(1.0,\ 1.5,\ \frac{\pi}{2})$, og den sorte kurve er kurven robotten rent faktisk har fulgt. Den grønne pil på robotten svarer til dens estimerede position. De lyseblå kurver er de resterende løsningsforslag for den valgte start- og slutposition. Hvad, og hvor meget, der tegnes kan vælges via brugergrænsefladen. Til venstre i skærmbilledet kan data vedrørende robotten ses, såsom dens estimerede og reelle position. Endvidere er alle tilknyttede gadgets og sensorer listet. I konsolen kan "snapshots" af data fra disse udskrives, såfremt der klikkes på det tilsvarende navn. Uanset robottens og controllerens design, så præsenteres tilknyttede gadgets og sensorer automatisk via brugergrænsefladen.

Idet alle komponenter et tilknyttet logisk sammen via `Component`-interfacet, er det i øvrigt nemt for en brugergrænseflade at udnytte dette. Vores implementering af modellen definerer for eksempel, at en komponent kan tegne sig selv, returnere sin omkreds, returnere den transformation der skal bruges for at skrifte fra forældrekomponentens koordinatsystem til dens eget, etc. Dette udnyttes i SWING-brugergrænsefladen til kun at opdatere de dele af animationen, der er ændret siden sidste frame, samt til at man kan tegne omridset for komponenterne enkeltvis, uanset antallet af komponenter, eller hvordan de er knyttet sammen. Dette betyder derfor, at *enhver* brugergrænseflade nemt kan udvides til at håndtere andre typer af robotter, for eksempel én med seks hjul, én med fire-hjulsstyring, osv.

Endeligt præsenteres også informationer om alle tilknyttede sensorer og gadgets i brugergrænsefladen, thi de også kan tilgås via henholdsvis komponenter og controlleren.

5.6.2 Simulationsobjekt

En simulator kan afvikle én simulation ad gangen. For at afvikle en simulation skal en mængde information kendes: Start- og slutposition, robotten der skal bruges (inklusiv controller), den totale simuleringstid, simulationslogger samt den valgte numeriske løsningsmetode. Disse informationer opbevares i en instans af klassen `dk.diku.robotsim.simulation.Simulation`, og den aktive simulation kan tilgås via simulatoren.

For at afvikle en "fuld simulation" kaldes følgende tre metoder på `Simulation`-klassen i rækkefølge:

- `initialise`: Initialiserer simulation og tilknyttede objekter, såsom robotten, controlleren og den numeriske løsningsmetode. Såfremt alle objekter er tilknyttede, og bliver initialiseret korrekt, markeres simulationen som initialiseret. Hvis simulationen ikke bliver initialiseret korrekt, da vil simuleringen ikke kunne startes.

- `update(double)`: Fremskriver simulationstiden svarende til parameteren angivet i sekunder. Dette kald videresendes til den numeriske løsningsmetode, som sørger for at simulationstiden fremskrives med faste tidsskridt (beskrevet i afsnit 5.3 på side 54), og løser den fysiske model svarende til den nye simulationstid.

- `complete`: Afslutter simulationen; sikrer bl.a. at simulationsloggeren flusher sine buffere.

En brugergrænseflade skal sørge for at overholdende ovenstående, og som standard sikres det ved at bruge et `CanonicalControlLoop` som beskrevet i næste afsnit.

5.6.3 Afvikling af simulationer

Uanset brugergrænsefladen, så kan en simulation afsluttes korrekt på en af følgende måder:

1. Robottens destinationspunkt er nået.
2. Den angivne simulationstid er udløbet.
3. Brugeren vælger at afbryde simulationen.

Punkt 1) og 2) vil altid gælde, men derimod er det ikke sikkert, at alle brugergrænseflader tillader, at en simulation kan afbrydes.

Såfremt brugergrænsefladen tillader en grafisk fremstilling af simulationens nuværende tilstand, da vil der blive brugt tid på at *rendere* robottens tilstand, dvs. tegne robotten i den nuværende situation. Denne tid må ikke påvirke den fysiske simulation, da det ville betyde, at simulationsdata for den samme simulation kunne blive forskellige for to maskiner med forskellig hastighed. Dette kan naturligvis generaliseres således, at det skal sikres, at simulationstiden og fremskrivningen af samme er uafhængig af maskinen.

For at adskille simulationstiden fra rendering, anden kontrol, osv., bruges klassen `dk.diku.robotsim.simulator.gui.CanonicalControlLoop`, som sørger for at bruge det aktive simulationsobjekt i simuleringen. Herved sikres det automatisk, at simulationsobjektets metoder kaldes i korrekt rækkefølge, og på korrekte tidspunkter. Den sørger samtidigt for at signalere, når den ønskede simulationstid er nået, eller robotten har løst sin opgave. Som standard afvikler klassen simulationer i realtid, men dette kan justeres, så simulationstiden kan gå hurtigere eller langsommere. For `ShellSimulator`, for eksempel, er det ønskværdigt, at simulationen skal færdiggøres så hurtigt som muligt, idet kommandoprompten er inaktiv, mens simulationen foregår, hvorimod det i `SwingSimulator` er fordelagtigt, at simulationen kører i realtid, så robotten kan animeres, mens den løser sin opgave[8]. Løkken kan endvidere afvikles som en separat tråd, som kan pauses når en simulering ikke er igangværende. Klassen er abstrakt, så implementerende

[8]Det er dog muligt via brugergrænsefladen i `SwingSimulator` at ændre realtidsfaktorens, så tiden kan gå hurtigere eller langsommere.

klasser skal for eksempel implementere, hvorledes renderingen skal foregå. Figur 27 viser *control-flow*'et når en simulation afvikles af en simulator som bruger `CanonicalControlLoop`-klassen.

Det er vigtig at påpege, at sikringen af at simulationstidsskridtet altid er det samme ligger i løsningen af den fysiske model, så det er totalt uafhængigt af brugergrænsefladen. Det betyder, at man kan give et vilkårligt stort tidsskridt til løsningsmodellen, som så selv sørger for at splitte tiden op, så der foretages et variabelt antal tidsskridt, men alle med samme størrelse.

5.7 Konfigurering

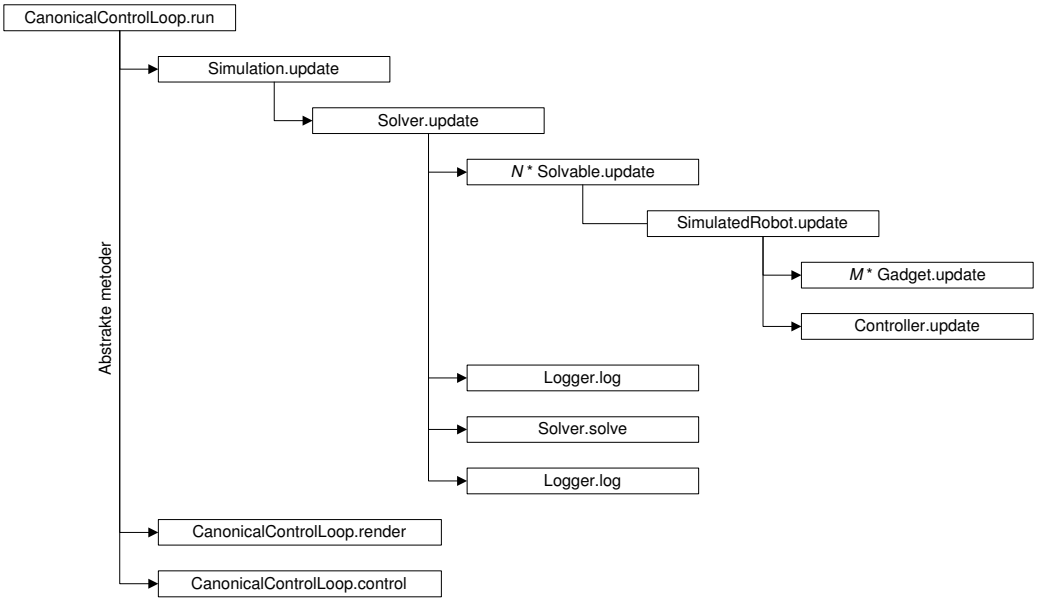
Selvom en bil-agtig robot har faste, tilknyttede komponenter, så er det ikke sikkert, at disse komponenter har samme bredde, højde, dybde, vægt, placering, osv. på alle robotter. Endvidere kan forskellige robotter have forskellige controllere, og dertil knyttede gadgets. I stedet for at oprette flere forskellige robotklasser, så kan den almindelige robotklasse nemt konfigureres, idet den implementerer interfacet `dk.diku.robotsim.configuration.Configurable`. Alle klasser, der implementerer dette interface, kan lade sig konfigurere af en specifik implementering (nedarvning) af klassen `dk.diku.robotsim.configuration.Configuration` og kaldes *konfigurerbare objekter*. For at fortsætte eksemplet med robotten, så bruges `dk.diku.robotsim.model.RobotConfiguration` til at konfigurere `dk.diku.robotsim.model.Robot`-instanser.

En konfiguration kan indlæse, *parse* og gemme en konfigurationsfil af et bestemt format fælles for alle `Configuration`-implementeringer. Konfigurationen forventer blot en sti til en fil eller et resursenavn, som kan tilgås via Java class path. De parsede værdier kan implementeringen så bruge til at initialisere det konfigurerbare objekt (for eksempel robotten). Formatet er altså ens for alle konfigurationer (*syntaksen*), men betydningen af de fundne konfigurationsværdier kendes kun af den specifikke implementering af `Configuration` (*semantikken*).

Følgende regler gælder for alle konfigurationsfiler:

- Værdier angives ved par af formen (`nøgle`, `værdi`).
- Rækkefølgen af angivne (`nøgle`, `værdi`)-par bevares, også hvis konfigurationen ændres og gemmes via en brugergrænseflade.
- Man kan angive kommentarer.

Det, at rækkefølgen bevares, er vigtigt, thi det kan være at én konfigurationsværdi afhænger af en anden.



Figur 27: Simulations-control-flow ved brug af `CanonicalControlLoop`-klassen. `run`-metoden vil være pauset ("idle loop") indtil en `Simulation` tilknyttes `Simulator`-instansen, og løkken signaleres (se afsnit 5.6.2). Herved kaldes metoden `Simulation.update`, som sørger for at den valgte numeriske løsningsmetode bruges, den er en implementering af `Solver`-interfacet (se afsnit 5.3). I `Solver.update` signaleres det `Solvable`-objekt som simuleres, dvs. implementeringen af robotten, via dens `Solvable.update`-metode. Denne metode kaldes et konfigurerbart antal gange N, inden den numeriske implementering løser ligningerne igen. Simulationsdata kan **enten** hentes efter hvert kald til `Solvable.update` **eller** først efter `Solver.solve` ved `Logger.log` (se afsnit 5.4). Den simulerede robot er implementeret ved `SimulatedRobot`, som implementerer `Solvable`; det betyder, at det altså er `SimulatedRobot.update` der i praksis kaldes fra `Solver.update`. I sin `update`-metode sørger robotten for at opdatere alle sine aktive, tilknyttede gadgets, M, hvorefter controlleren altid opdateres til sidst (se afsnit 5.2.4). Herefter kan kontrolløkken rendere den nuværende tilstand ved `render`-metoden, og der kan foretages yderlige opgaver, hvis det er nødvendigt, via `control`-metoden; begge er abstrakte, som skal implementeres i den anvendte kontrolløkke. Herefter starter den interne løkke forfra i `run`-metoden, og fortsætter (eventuelt pauset) indtil det aktivt afbrydes; for eksempel fra en tilknyttet handler som bliver aktiveret når robotten har nået sit mål, eller den angivne totale simulationstid er nået.

Såfremt parsingen af konfigurationsfilen er succesfuld, da kan værdierne tilgås via deres tilknyttede nøgle og derved bruges af konfigurationen til at initialisere det konfigurerbare objekt.

I applikationen er der fire konfigurerbare objekter, hvis konfigurering beskrives i de næste sektioner: `Simulator`, `Simulation`, `Robot` og (Simulations-)`Logger`. JavaDoc'en for de enkelte konfigurationsobjekter indeholder endvidere en detaljeret beskrivelse af de enkelte nøgler og deres funktion.

5.7.1 Simulator

Simulatoren kan konfigureres, idet en konfigurationsfil kan indeholde standardværdier for filendelser for andre konfigurationsfiler brugt af simulatoren. På sigt kan denne konfiguration udvides til at indeholde andre fælles egenskaber for simulatoren.

Klassen `dk.diku.robotsim.simulator.SimulatorConfiguration` bruges til at konfigurere simulatoren. Nøgler til konfigurationen er specificeret ved den indre klasse `ConfigurationKeys`.

5.7.2 Simulation

En konfigurationsfil til at konfigurere en `Simulation`-instans kan oprette og konfigurere alle tilknyttede objekter og attributter. Det betyder, at når en simulationskonfiguration er gennemført, vil simulationen eventuelt være klar til at blive kørt. Konfiguration kræver nemlig ikke at start- og slutpositionen skal indtastes, idet brugergrænsefladen skal sørge for det.

Klassen `dk.diku.robotsim.simulation.SimulatorConfiguration` bruges til at konfigurere simulatoren. Nøgler til konfigurationen er specificeret ved den indre klasse `ConfigurationKeys`. Nedenfor vises et eksempel på nøgler og tilknyttede værdier.

```
# Robot configuration
robot.configuration = resources/murphy

# Controller class
controller.class = dk.diku.robotsim.implementation.controller.CarLikeController

# Solver class and log mode
solver.class   = dk.diku.robotsim.simulation.solver.EulerSolver
solver.logmode = update

# Start and end position
position.start = 0.000; 0.250; 0.000
```

```
position.end   = 2.000; 1.500; 1.500

# Maximum "Simulation time" in seconds
simulation.time = 10

# Logger configuration
logger.configuration = resources/logger
```

Nøglerne `robot.configuration` og `logger.configuration` forventer et ressursenavn på konfigurationsfiler til henholdsvis en robot og en simulationslogger. Det vil automatisk betyde, at robotten tilknyttet simulationen vil blive konfigureret (og oprettet hvis nødvendigt), og det samme vil være gældende for simulationsloggeren.

Ydermere specificerer nøglerne `controller.class` og `solver.class` henholdvis implementeringerne af de to interfaces `dk.diku.robotsim.model.Controller` og `dk.diku.robotsim.simulation.solver.Solver`, der skal bruges i simulationen.

5.7.3 Robot

Som nævnt har robotten mange attributter at skrue på. Det er ikke praktisk muligt at kende alle (`nøgle`, `værdi`)-par svarende til disse på forhånd. Robotkonfigurationen tager højde for dette ved at bruge Java's indbyggede *Reflection* API i pakken `java.lang.reflect`. Denne tillader, at man i sin programkode forespørger på, og bruger, metadata omhandlende pakker, klasser, metoder, attributter, osv. Konfigurationen antager, at en nøgle svarer til en "sti" til en attribut, og værdien skal tildeles denne attribut. Stien (nøglen) skal være af formen:

```
object1.object2.object3.[..].objectN.attribute
```

Her har `object1` `object2` tilknyttet som en offentlig (`public`) attribut eller kan tilgås via en *getter*-metode navngivet `getObject2`; `object2` har `object3` tilknyttet på samme måde; osv. Til sidst skal `objectX` have tilknyttet attributten med navnet `attribute` som en offentlig attribut, eller den skal kunne tildeles en værdi via en *setter*-metode navngivet `setAttribute`. Attribut/parameter-type transformeres automatisk fra en strengtype (værdi) til den type, der forventes. Selvom det er muligt at opdatere en attribut via reflection uanset dens tilgængelighed, så sætter vi kun `public`-attributter direkte, da private attributter ofte er private med god grund. For eksempel kan det være, at når en privat attribut opdateres via en *setter*-metode, så efterbehandles værdien, inden den tildeles. Endeligt vil dette også virke for metoder nedarvet fra superklasser.

Et eksempel på et (`nøgle`, `værdi`)-par kan være:

```
chassis.steering.maxAngle = 0.50
```

Dette betyder, at robottens chassis hentes (`getChassis`), hvorfra styringsakslen (forakslen) hentes (`getSteering`), og hvorpå den maximale drejningsvinkel sættes til 0,5 meter (`setMaxAngle(0.5)`). I dette tilfælde er alle metodekald defineret som en del af modellen, undtaget opdateringen af vinklen. Dette vil jo oftest være tilfældet, da modellen ikke specificerer, hvordan man sætter attributter; kun hvordan de tilgås.

5.7.4 Logger

En konfigurationsfil til simulationsloggere kan konfigurere en eller flere loggere. Hver logger kan konfigureres unikt, og såfremt der er defineret flere loggere, så tilknyttes de 2-n loggere som børn af den første definerede logger. Faktisk kan alle egenskaber beskrevet i afsnit 5.4 på side 56 konfigureres. Klassen `dk.diku.robotsim.logger.LoggerConfiguration` bruges til konfigureringen, og de mulige nøgler er angivet ved den indre klasse `ConfigurationKeys`.

6 Evaluering

I denne rapport vil vi ikke foretage en formel afprøvning af den udviklede simulator, idet det vil være en endog meget tidskrævende proces. Vi vil blot evaluere nogle centrale dele af den – dvs. foretage nogle testkørsler, og derefter kommentere på resultaterne af disse.

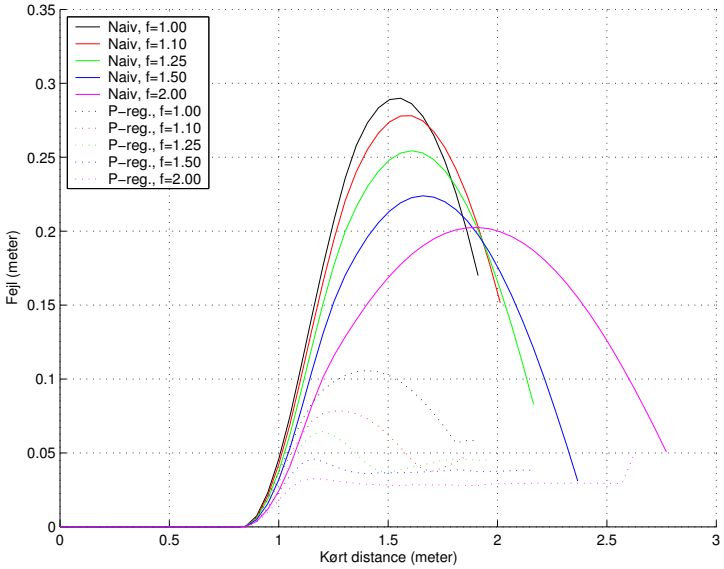
6.1 Evaluering af kurveføIgningsalgoritmerne

I dette afsnit vil kurveføIgningsalgoritmernes evne til at følge en kurve med en given mindste krumningsradius blive evalueret. Dette er et helt centralt element i den samlede løsning af position-til-position bevægelsesproblemet, og vi vil her komme med et godt bud på, hvor lille ρ_{min} kan vælges for en kurve så en given robot kan følge kurven med en given kurveføIgningsalgoritme. Jo mindre ρ_{min}, jo kortere kurve. Derudover evalueres de forskellige kurveføIgningsalgoritmer mod hinanden.

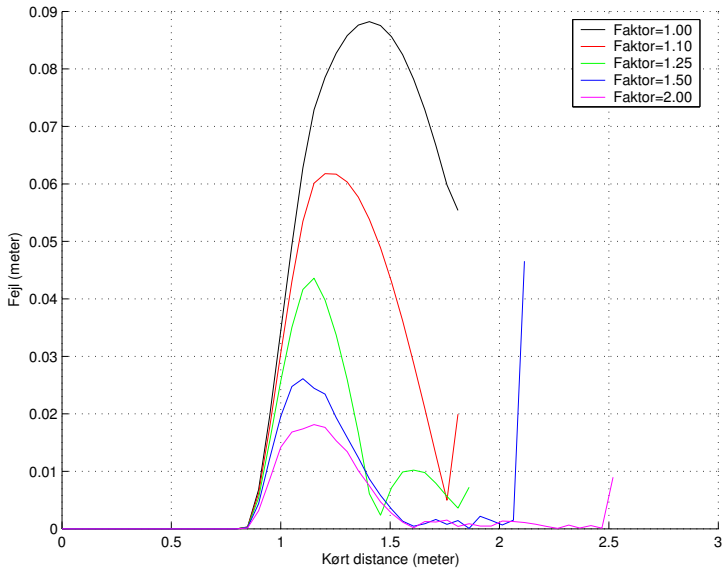
Hver af de fire kurveføIgningsalgoritmer (inklusive PID-regulering) beskrevet i afsnit 4.5 afprøves med fem forskellige bud på ρ_{min}: 1.00, 1.10, 1.25, 1.50 og 2.00 gange robottens mindste drejeradius udregnet efter formel 40. Robotten anvender sit GPS-positionsbestemmelsesmodul, så usikkerhed i bestemmelsen af positionen ikke påvirker resultaterne. Resultaterne kan ses på figurerne 28, 29 og 30.

Som det ses af figurerne, kan ingen af algoritmerne følge testkurven helt præcist, når cirkelbuen har en radius $r = \rho_{min}$. Dette er ikke så overraskende, da det fysisk ikke kan lade sig gøre pga. hjulvinklens begrænsede drejehastighed. Det kan desuden straks ses, at den naive algoritme er meget dårligere end de andre algoritmer, samt at PID-reguleringsalgoritmen i alle tilfælde er bedre end P-reguleringsalgoritmen. Det begrænser valget til enten PID-algoritmen eller den successive punktstyringsalgoritme. PID-algoritmen har en stor initiel fejl, hvor cirkelbuen starter, men herefter falder fejlen stødt. Den successive punkstyring har en væsentlig mindre initiel fejl, men til gengæld svinger fejlen derefter kraftigt pga. det successive valg af nyt styrepunkt. Valget er ikke entydigt, men må falde på enten PID-algoritmen med en mindste krumningsradius omkring 25% højere end robottens ρ_{min}, eller på den successive punktstyring med samme mindste krumningsradius som robottens. PID-algoritmen giver en mere jævn styring, men den successive punktstyring styrer robotten gennem den korteste rute.

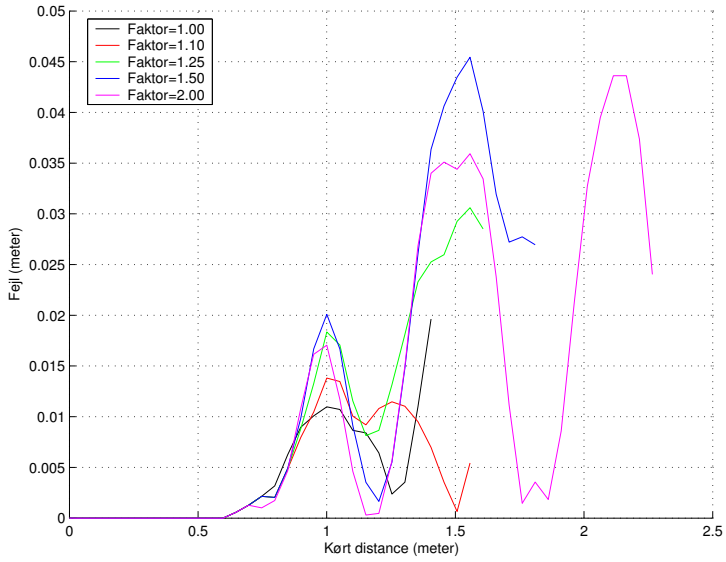
Valget af kurveføIgningsalgoritme afhænger endvidere af robottens hastighed, samt generelt af kurvens form.



Figur 28: Robotten kører langs en kurve som består af en 1,0 meter lang linie, efterfulgt af en cirkelbue på 180°. Cirklens radius varieres fra 1,0 til 2,0 gange ρ_{min} for robotten. Robotten accelererer til 1,0 m/s langs linien inden cirklenbuen mødes. På figuren styres robotten af henholdsvis den naive kurvefølgningsalgoritme, jævnfør afsnit 4.5.1, og p-reguleringsalgoritmen, jævnfør afsnit 4.5.2.



Figur 29: Robotten kører langs samme kurve og med samme hastighed som på figur 28, men denne gang med PID-reguleringsalgoritmen, jævnfør afsnit 4.5.2. Det afsluttende "hop" på kurverne skyldes, at robotten netop når at køre forbi slutpunktet, før simulationen afsluttes, og er derfor ikke udtryk for unøjagtigheder i algoritmen.



Figur 30: Robotten kører langs samme kurve og med samme hastighed som på figur 28, men denne gang med den successive punktstyringsalgoritme, jævnfør afsnit 4.5.3.

6.2 Evaluering af løsningen af position-til-position bevægelsesproblemet

I dette afsnit vil vi sammenligne vores teoretiske løsning af position-til-position bevægelsesproblemet med den simulerede løsning. Forskellige konkrete løsninger vil blive sammenlignet. Vi vælger at lade den simulerede robot køre den korteste kurve mellem to positioner med henholdsvis PID-reguleringen og den successive punktstyring som kurvefølgningsalgoritmer. Ved PID-reguleringen vælgers kurvens mindste krumningsradius 25% større end robottens minimale drejeradius, og ved den successive punktstyring vælges kurvens mindste krumningsradius som robottens minimale drejeradius, jævnfør forrige afsnit. Ved hver kørsel prøves både GPS- og Dead Reckoning-positioneringen, så disse samtidigt kan sammenholdes. Tre eksempler på start- og slutpositioner vælges på en måde, så de tre grundlæggende ruteklasser lsr, lsl og lrl afprøves.

6.2.1 Ruteklassen lsr

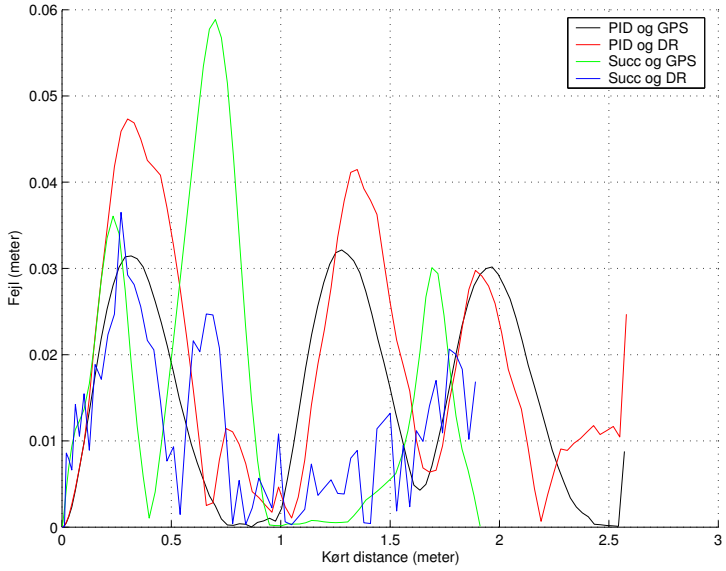
Start- og slutpositionen vælges som henholdsvis $(0.165,\ 0.000,\ 0.000)$ og $(-0.335,\ 1.500,\ 0.000)$. For $1.00 \cdot \rho_{min}$ giver dette den korteste rute $l^+_{0.7141} s^+_{0.9211} r^+_{0.7141}$ med en samlet længde på 2.3493; for $1.25 \cdot \rho_{min}$ bliver den korteste rute $l^+_{1.0078} s^+_{0.6600} r^+_{1.0078}$ med samlet længde på 2.6756. Se figur 31.

6.2.2 Ruteklassen lsl

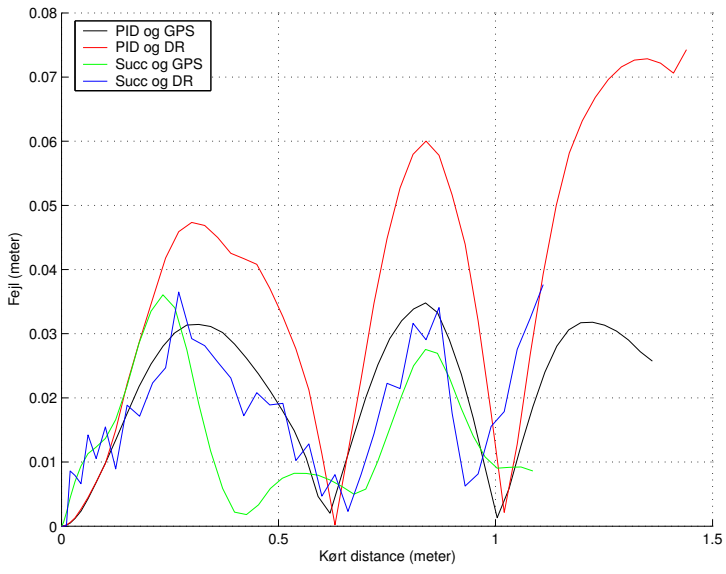
Start- og slutpositionen vælges som henholdsvis $(0.165,\ 0.000,\ 0.000)$ og $(0.165,\ 1.000,\ 3.1415)$. For $1.00 \cdot \rho_{min}$ giver det den korteste rute $l^+_{0.4324} s^+_{0.4495} l^+_{0.4323}$ med en samlet længde på 1.3142; for $1.25 \cdot \rho_{min}$ bliver den korteste rute $l^+_{0.5405} s^+_{0.3118} l^+_{0.5404}$ med samlet længde på 1.3928. Se figur 32.

6.2.3 Ruteklassen lrl

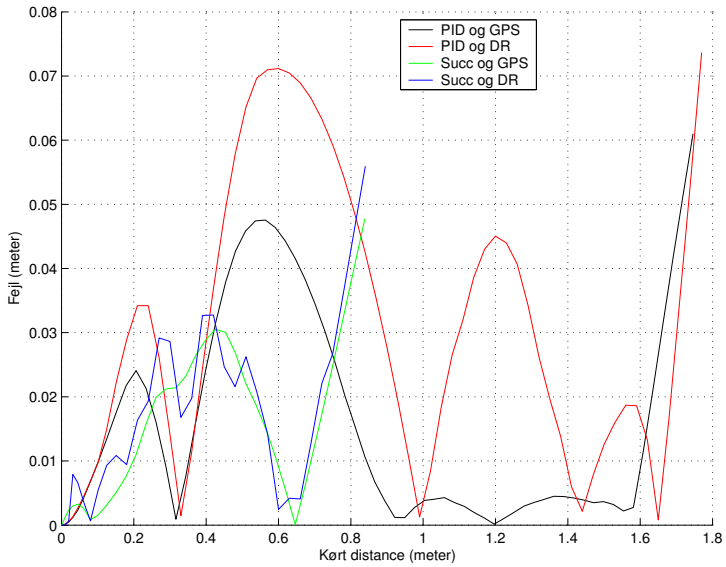
Start- og slutpositionen vælges som henholdsvis $(0.165,\ 0.000,\ 0.000)$ og $(0.165,\ -0.500,\ 3.1415)$. For $1.00 \cdot \rho_{min}$ giver det den korteste rute $l^+_{0.0837} r^+_{1.0322} l^+_{0.0837}$ med en samlet længde på 1.1996; for $1.25 \cdot \rho_{min}$ bliver den korteste rute $l^+_{0.1820} r^+_{1.4450} l^+_{0.1820}$ med samlet længde på 1.8090. Se figur 33.



Figur 31: Kørsel langs en kurve af formen lsr. For PID-regulering er mindste krumningsradius 25% større end robottens mindste drejeradius; for successiv punktstyring er de ens. De fulgte ruter er beskrevet i afsnitsteksten.



Figur 32: Kørsel langs en kurve af formen lsl. For PID-regulering er mindste krumningsradius 25% større end robottens mindste drejeradius; for successiv punktstyring er de ens. De fulgte ruter er beskrevet i afsnitsteksten.



Figur 33: Kørsel langs en kurve af formen lrl. For PID-regulering er mindste krumningsradius 25% større end robottens mindste drejeradius; for successiv punktstyring er de ens. De fulgte ruter er beskrevet i afsnitsteksten.

6.2.4 Sammenligning af resultaterne

Aflæses graferne på figur 31, 32 og 33 kan det umiddelbart se ud som om, at den successive punktstyring i nogle tilfælde kører en *kortere* rute end den teoretisk mulige. Dette grunder dog i, at den regner ruten som værende kørt til ende, når det sidste punkt på ruten er "passeret", jævnfør afsnit 4.5.3. Dette vil typisk ske allerede inden robotten er nået punktet. Dette gør det naturligvis lidt svært at evaluere dens kørte længde mod den teoretisk korteste, men det kan konstateres, at den helt sikkert kører en kortere rute end ved brug af PID-regulering. Sidstnævnte stopper til gengæld temmelig præcist!

Tabel 2 udregner, hvor meget længere robotten rent faktisk kører i forhold til det teoretisk korteste, når den styres af PID-reguleringen, som gør brug af Dead Reckoning som positioneringssystem.

Klasse	**Kørt længde**	**Teoretisk v.** ρ_{min}	**Teoretisk v.** $1.25 \cdot \rho_{min}$
lsr	2.58	2.3493 (110%)	2.6756 (96%)
lsl	1.44	1.3142 (110%)	1.3928 (103%)
lrl	1.77	1.1996 (148%)	1.8090 (98%)

Tabel 2: Sammenligning mellem tilbagelagt strækning i praksis og den teoretiske korteste rute ved brug af PID-regulering og Dead Reckoning.

Som det ses af tabel 2 kører PID-reguleringen omtrent den samme længde som længden af ruten den skal følge ved den valgte radius på $1.25 \cdot \rho_{min}$. Men desværre er dette mellem 110% og 148% længere end det teoretisk korteste ved radius ρ_{min}.

Hvis start- og slutpositionen ligger langt fra hinanden vil det meste af ruten være en lige linie. Da både PID-reguleringen og den successive punktstyring er rigtigt gode til at køre langs en lige linie, så vil den kørte distance i sådanne tilfælde komme tæt på det teoretisk mulige.

Yderelige kan det af tabel 3, samt figurne 31, 32 og 33, ses, at det ikke gør den store forskel om positioneringssystemet er GPS eller Dead Reckoning; GPS giver dog trods alt – og som ventet – en smule kortere vej, da det selvsagt er mere nøjagtigt.

Klasse	**Kørt længde**	**Teoretisk v.** ρ_{min}	**Teoretisk v.** $1.25 \cdot \rho_{min}$
lsr	2.57	2.3493 (109%)	2.6756 (96%)
lsl	1.36	1.3142 (103%)	1.3928 (97%)
lrl	1.75	1.1996 (146%)	1.8090 (97%)

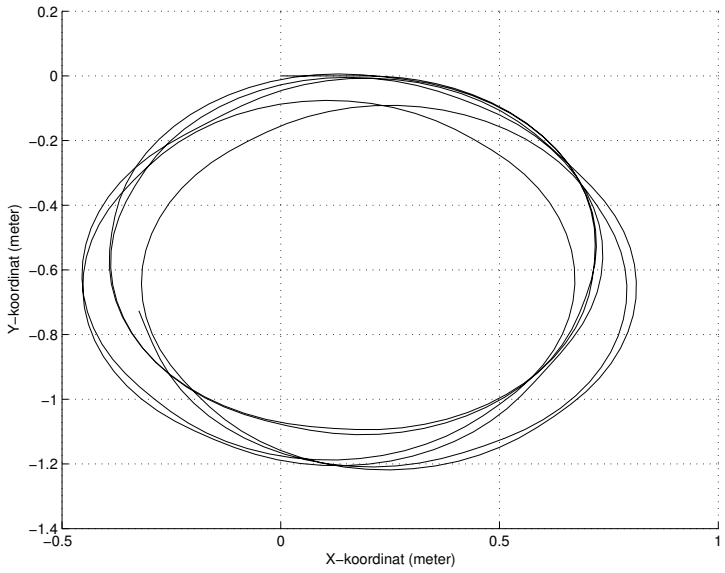
Tabel 3: Sammenligning mellem tilbagelagt strækning i praksis og den teoretiske korteste rute ved brug af PID-regulering og GPS.

6.3 Numeriske løsningsmetoder

Det er umiddelbart svært at komme med et konkret eksempel, som viser forskellen mellem de to implementerede numeriske løsningsmetoder, dvs. mellem Euler og 2. ordens Runge-Kutta. Hvis forskellen mellem dem tydeligt skal illustreres skal der konstrueres en differentialligning, som derefter både løses analytisk og ved brug af de to numeriske metoder. Herved kan resultaterne holdes op mod hinanden.

Vi har valgt at udelade sådanne sammenligninger. I stedet har vi blot foretaget en *testkørsel*, hvor en robot skiftevis kører ligeud og drejer 90°. Resultatet af simulationen løst med hver af de to metoder kan ses på henholdsvis figur 34 og 35. De to løsninger er tydeligvis forskellige, men figurene viser selvfølgelig ikke, om den ene

er bedre end den anden. Men den gængse litteratur konstaterer, at Runge-Kutta-metoden er den mest præcise, jævnfør for eksempel [Burden and Faires, 1997, kapitel 5].

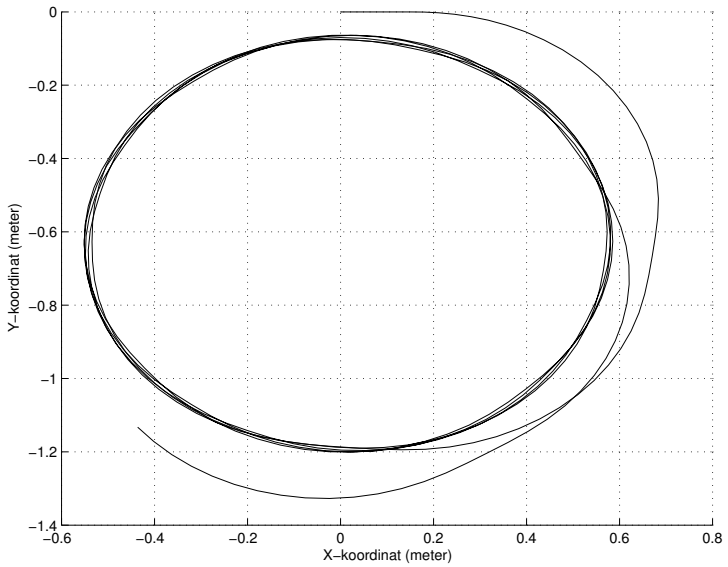


Figur 34: Afprøvning af Euler-løseren.

6.4 Simulatorer

Vores mål angående at udvikle en simulator, der nemt kan simulere en mængde af forskellige bil-agtige robotter som specificeret i indledningen samt i den matematiske model, er lykkedes til fulde. For begge simulatorer er det muligt at bruge konfigurationsfiler, så et minimum af data skal indtastes, og simuleringer kan nemt genskabes – med samme resultat, naturligvis! Endvidere er det i `SwingSimulator` bl.a. muligt at animere robotten, mens den løser sin opgave. Dette har faktisk været en stor hjælp under udvikling og fejlfinding af de implementerede stiføl gere. Uanset brugergrænseflade, så fremstår implementeringen af robotmodellen, såvel som resten af simulatordesignet, yderst fornuftigt og brugbart.

Men, når det er sagt, så bliver vi nødt til at nævne, at der er kendte problemer med SWING brugergrænsefladen. Ved projektets start havde ingen af forfatterne arbejdet med SWING før, og det blev valgt, fordi det er en integreret del af Java. Således kræves ingen ekstra biblioteker for at afvikle en grafisk simulator som bygger på SWING. Men SWING er **meget** tungt og omstændigt at arbejde med,



Figur 35: Afprøvning af Runge-Kutta-løseren

og faktisk er klart hovedparten af udviklingen af simulatorerne gået med SWING-brugergrænsefladen – hvilket bestemt ikke har været hensigten til at starte med. Det ses også på antallet af klasser brugt til denne brugergrænseflade sammenlignet med antallet af simulatorklasser i almindelighed. Såfremt vi skulle udvikle en ny grafisk brugergrænseflade på ny ville vi helt sikkert ikke vælge SWING, men i stedet *The Standard Widget Toolkit* (SWT)[9], for eksempel. Der kan derfor forekomme fejl i SWING-brugergrænsefladen, som vi grundet tidspres ikke har nået at løse.

Vi har ikke brugt tid på at modellere brugergrænsefladerne via *useability* eller HCI (*Human-Computer Interaction*) modeller. Derfor evaluerer vi heller ikke brugergrænsefladerne ud fra deres brugervenlighed, kun funktionalitet. Og som nævnt kan begge simulatorapplikationer afvikle simulationer.

[9] `http://www.eclipse.org/articles/Article-SWT-Design-1/SWT-Design-1.html`.

7 Visioner

Dette kapitel omhandler *forslag* om udvidelser til det udviklede simulatorframework. Kapitlet er på ingen måde fyldestgørende, da vi kunne finde på endnu flere – og måske mere spændende – ting. Vi beskriver ikke problemer med SWING brugergrænsefladen her, idet det er oplagt, at de bør løses så hurtigt som muligt.

7.1 Implementation på en fysisk robot

Dead Reckoning, som beskrevet i afsnit 4.4.2 er allerede blevet implementeret på Murphyrobotten, og med de samme gode resultater som opnået i simulationerne, nemlig med en afvigelse på omkring kun 2% i positionen pr. kørte meter. Yderligere optimering og justering ved simulationer med efterfølgende fysisk implementering af ruteberegnere og stifølgere på robotten ønskes, så RoboCup kan vindes år 2005!

7.2 Modellen

7.2.1 Nye robottyper

Som beskrevet, så opererer vi med en fast type af robotter specificeret ved modellen. Den generelle model er dog så generisk designet og implementeret, at den relativt nemt kan udvides til at omhandle andre typer robotter. Komponenter, såvel som gadgets, skal allerede nu registreres til en forældre-komponent, og på den måde gør det for eksempel ingen forskel med hensyn til brugergrænsefladen.

Det springende punkt er naturligvis såfremt nye kræfter/aktuatorer introduceres, thi de skal modelleres rent matematisk, og herefter implementeres. De vil dog sagtens kunne fungere indenfor rammerne af de eksisterende model- og simulationsinterfaces, idet tilstande hentes og sættes ved hjælp vektorer af data. Det vil dog naturligvis tage nogen tid at skulle definere og implementere nye kræfter som tilføjelse til den overordnede model.

7.2.2 Eksterne fysiske påvirkninger

I den nuværende løsning antages at underlaget er plant og glat, og at det ikke har nogen friktionsmodstand. Dette er ikke en realistisk betragtning, og det synes fornuftigt at indføre ændringer til modellen, så mere avancerede eksterne fysiske kræfter kan påvirke robotten på dens færd. Samtidigt vurderes det dog som en

krævende opgave (som i høj grad også vil føre til ændringer i brugergrænsefladen for især for grafisk baserede simulatorer), samt i bevægelsesligningerne i modellen af robotten.

7.2.3 Forhindringer

I forlængelse af ovenstående punkt kunne det være interessant at introducere fysiske forhindringer, som robotten skal tage højde for. Dette vil komplicere robottens opgave med at finde den optimale rute. Som implementationen er i dag, så kan robotten allerede finde den optimale rute bestående af flere på hinanden følgende del-optimale løsninger. Ved at detektere og klassificere fysiske forhindringer på vejen mellem to positioner, kunne enhver forhindring betragtes som et endepunkt på en delløsning. Dette kræver dog at alle forhindringer kendes på forhånd; det kunne være tilfældet, hvis robotten har kørt ruten før – hvilket er interessant, idet ruten til RoboCup er kendt på forhånd. Alternativt kan den optimale rute genberegnes når en fysisk forhindring mødes, hvor den den bruges som nyt startpunkt, og beholdende det originale slutpunkt. Dette vil dog også kræve, at yderligere sensorer er til rådighed for robotten, for eksempel et kamera eller en afstandsmåler. Af denne grund vurderes dette til at være en ret stor ændring, men samtidigt værende en interessant en af slagsen; især taget i betragtning af, at der under RoboCup-turneringen jo vil være sådanne fysiske forhindringer, og at ruten som sagt er kendt på forhånd. [Jacobs and Canny, 1989] beskriver en algoritme, som med udgangspunkt i den samme løsning af position-til-position bevægelsesproblemet som blev beskrevet i afsnit 4.1, løser problemet, hvor forhindringer medtages vha. en grafsøgealgoritme.

7.3 Simulator

7.3.1 Andre numeriske løsningsmetoder

Vi har implementeret både Euler og 2. ordens Runge-Kutta. Det kunne være interessant at implementere højereordens Runge-Kutta-metoder og sammenligne dem med den allerede implementerede metode. Især sammenholdt med at de beregningsmæssigt er dyrere end en 2. ordens.

7.3.2 Nye brugergrænseflader

I forbindelse med udviklingen af simulatorerne opdagede vi, at især SWING kan være særdeles besværligt og tungt at danse med, både i udviklingstid og i effek-

tivitet på køretidspunktet. En “lettere” udgave af den grafiske brugergrænseflade kunne ønskes, for eksempel som en *applet*, der kan afvikles i en browser. De samme muligheder vil dog ikke direkte kunne være til stede i en sådan version, for eksempel filhåndtering grundet de sikkerhedsmæssige aspekter der introduceres, når Java afvikles i en browser. Alt andet lige vil en applets brugergrænseflade også kunne være med til at udbrede kendskabet til den udviklede løsning.

7.3.3 Brug af metadata

Idet vi allerede specificerer at objekter kan reportere deres tilstand til brug for logning af simulationsdata, ville det være oplagt at bruge disse metadata direkte i brugergrænsefladerne. Som det er nu udskrives for eksempel sensores information via deres `toString`-metode, men det kunne generaliseres til at bruge metadata i stedet, så man ville opnå fuldstændig konsistens mellem genererede logfiler og brugergrænsefladen.

8 Konklusion

Det i indledningen angivne mål, inklusive delmål, er blevet indfriet i dette projekt.

Vi har opstillet og implementeret en matematisk model for en bil-agtig robots bevægelse i planen.

Vi har fundet frem til to artikler der beskriver en algoritme, som løser opgaven med at finde den korteste afstand mellem to positioner under hensyntagen til modellen. Ligningerne for algoritmen blev udledt, da de i artiklen postulerede ligninger indeholdt diverse småfejl. Algoritmen er blevet implementeret, og det er konstateret, at den finder de forventede ruter.

Til at følge de fundne ruter er adskillige algoritmer opstillet, implementeret og sammenlignet.

Vi har implementeret en simulator med to forskellige brugergrænseflader, som kan afvikle simulationer af bil-agtige robotter. Det udviklede framework føler vi endvidere er så godt gennemarbejdet, at det vil være nemt at videreudvikle, eller blot tilføje nye brugergrænseflader. Samtidigt tillader det udvikling og afprøvning af andre problemstillinger for bil-agtige robotter end position-til-position bevægelsesproblemet.

Litteratur

[rob, 2004] (2004). *DTU RoboCup: Robotternes tumleplads*. Danmarks Tekniske Universitet.

[Burden and Faires, 1997] Burden, R. L. and Faires, J. D. (1997). *Numerical Analysis*.

[Dubins, 1957] Dubins, L. E. (1957). *On curves of minimal length with a constraint on average curvature, and with prescribed initial and terminal position and tangents*.

[Egerstedt et al., 1997] Egerstedt, M., Hu, X., Rehbinder, H., and Stotsky, A. (1997). *Path Planning and Robust Tracking for a Car-Like Robot*.

[Fabricius-Bjerre, 1977] Fabricius-Bjerre, F. (1977). *Laerebog i geometri II - Differentialgeometri, kinematisk geometri*.

[Jacobs and Canny, 1989] Jacobs, P. and Canny, J. (1989). *Planning smooth paths for mobile robots*.

[Kongsbak et al., 2002] Kongsbak, U., Larsen, K. L., Leuteritz, H., Lundkvist, J., Philipsen, M., and Tejlmand, T. (2002). *Trajectory Tracking System for an Autonomous Agricultural Vehicle*.

[Ljung and Glad, 1991] Ljung, L. and Glad, T. (1991). *Modellbygge och simulering*.

[Luca et al., 1999] Luca, A. D., Oriolo, G., and Samson, C. (1999). *Robot Motion Planning and Control*.

[Lund, 2000] Lund, J. (2000). *Den store danske encyklopædi*.

[McManis, 2003] McManis, C. (2003). *Servo-Motor 101*. CDROM: web/F_Servo101.html.

[Meriam and Kraige, 1998] Meriam, J. and Kraige, L. G. (1998). *Engineering Mechanics: Dynamics*.

[Moret, 2003] Moret, E. N. (2003). *Dynamic Modeling and Control of a Car-Like Robot*.

[Ole Jannerup, 2000] Ole Jannerup, P. H. S. (2000). *Introduktion til reguleringsteknik*.

[Reeds and Shepp, 1990] Reeds, J. and Shepp, R. (1990). *Optimal paths for a car that goes both forward and backwards*.

[Sekhavat and Hermosillo, 2001] Sekhavat, S. and Hermosillo, J. (2001). *Cycab Bi-steerable Cars: a New Family of Differentially Flat Systems*.

[Shiller and Gwo, 1991] Shiller, Z. and Gwo, Y.-R. (1991). *Dynamic Motion Planning of Autonomous Vehicles*.

A Robotten Murphy

Dette appendiks beskriver karakteristika for robotten Murphy som deltog i RoboCup 2004 – og i øvrigt fik en 2. plads. Første afsnit omhandler dens udseende og hardware; andet afsnit består af en tabel over robottens fysiske dimensioner.

A.1 Hardware

Robotten er en firehjulet bil-agtig robot, dvs. den er forhjulsstyret. Chassiset er bygget i LEGO Technic, og hvert baghjul er drevet af hver sin LEGO-motor. På hvert baghjul sidder desuden et tachometer. Forhjulene bliver styret af en standard hobbyservomotor; for mere om servomotorer se [McManis, 2003]. Robotten bliver styret af en selvbygget controller med en 14 Mhz Atmel AtMega32 som CPU. CPU'en har 32 kb program flash-ram, samt 2 kb SRAM (alm. hukommelse) on-chip.

Robottens udstyr af mindre interesse for nærværende rapport er: tre IR-afstandssensorer, en syv-kanals stregsensor, en ultralydsafstandssensor, 80-tegns LCD-display, talesyntese, forlygter, bremselys og afviserblink.

A.2 Robottens dimensioner

Enhed	**Størrelse**
Længde	0,22 m
Bredde	0,10 m
Hjulradius	0,04 m
Længde mellem for- og bagakse (L)	0,16 m
Længde mellem de to baghjul (W)	0,12 m
Tachometeropløsning	40 ticks pr. omdrejning
Forhjulenes maksimale styrevinkel	ca. $\pm$ 0,54 radianer
Forhjulenes maksimale styrevinkelhastighed	ca. 2 radianer pr. sekund
Maksimal hastighed	Ukendt, men i omegnen af 1 m/s

Vi har defineret en robotkonfigurationsfil til brug ved simulation med de fleste af ovenstående karakteristika. Den kan findes på den til denne rapport vedlagte cd som filen `murphy.robot` under `simulator/resources`.