BHT Berliner Hochschule für Technik

Aufgabensammlung AMS

Autonome Mobile Systeme

Masterstudiengang
Technische Informatik – Embedded Systems

© Prof. Dr.-Ing. Volker Sommer

WS 2024/25

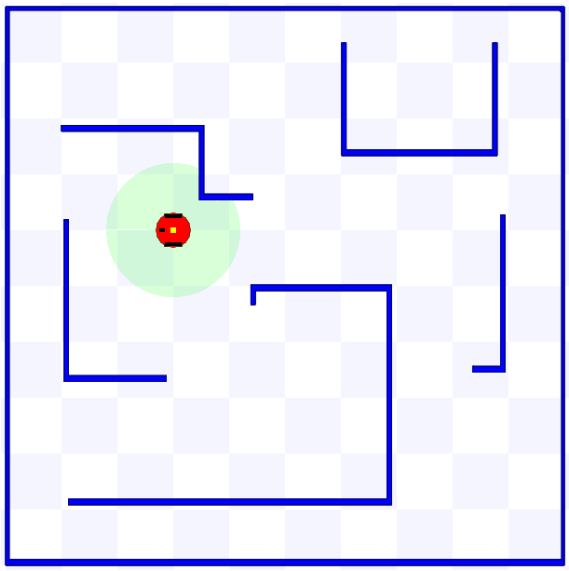
Die hier beschriebenen Aufgaben basieren auf den in der Vorlesung AMS vermittelten Inhalten. Sie sind unter Verwendung des Roboter-Frameworks Player/Stage und einer Entwicklungsumgebung für C++ (CodeBlocks) weitestgehend selbständig zu lösen, wobei vorgegebene Programmgerüste (Templates) zu ergänzen sind. Als Grundlage dient die Klasse AMS_Robot, die auf Basis der Player-Interfaces position2d, ranger und graphics2d eine einfache Realisierung von Robotersimulationen ermöglicht, indem sie die Erstellung von Player-Clients vollständig kapselt und außerdem zahlreiche Methoden zum Lesen und Setzen von Roboterparametern, zur Grafikausgabe innerhalb des Stage-Fensters und zur Steuerung bzw. Regelung der Roboterbewegung zur Verfügung stellt. Die Konfiguration von Player wird durch die Datei AMS.cfg vorgegeben, zur Festlegung der Simulationsparameter in Stage dienen die Datei AMS.world und außerdem das Binärbild AMS.png, welches Hinderniskonturen enthält. Obwohl in der Übung ausschließlich mit Simulationen gearbeitet wird, können die erstellten Programme grundsätzlich ebenfalls zur Steuerung realer mobiler Roboter eingesetzt werden, sofern deren Software die Player-Schnittstellen position2d und ranger unterstützt. Hierzu müssen einige Parameter in den Programmen geändert werden. Außerdem muss die Datei AMS.cfg modifiziert werden, so dass der Roboter unter seiner IP-Adresse gefunden wird. Für nähere Informationen wird auf die Dokumentation von Player verwiesen.

Inhalt

1. Aufgabe – Quasi-zufällige Bewegung eines mobilen Roboters

In dieser ersten Übungsaufgabe soll aus gegebenen C++ Quelltextdateien mit Hilfe einer integrierten Entwicklungsumgebung ein lauffähiges Programm erstellt werden, das einen mit einem Laserentfernungssensor ausgestatteten mobilen Roboter innerhalb einer *player/stage*-Simulationsumgebung quasi zufällig steuert. Das dabei generierte Projekt innerhalb der Entwicklungsumgebung bildet auch die Grundlage für spätere Übungen.

Damit das Steuerprogramm über das *player*-Framework Informationen mit dem Robotersimulator *stage* austauschen kann, müssen zuerst diese beiden Programme gestartet werden. Dies geschieht mittels der Befehlszeile *player /path/AMS.cfg*, wobei *path* für den Pfad auf Ihrem Linux-System zu der *player*-Konfigurationsdatei *AMS.cfg* steht. Innerhalb dieser Datei ist der Pfad zur *stage*-Konfigurationsdatei *AMS.world* enthalten, die den simulierten Roboter und die Umgebung definiert. Dazu ist in dieser Datei auch der Pfad zu der Binärdatei *AMS.png* enthalten, welche die Konturen des in der Übung verwendeten, simulierten Raumes enthält, siehe nachfolgendes Bild.



a) Kopieren Sie die drei genannten Konfigurationsdateien in Ihr Arbeitsverzeichnis und starten Sie von einem Terminal aus wie gezeigt die beiden Programme *player* und *stage*, so dass das oben dargestellte Simulationsfenster geöffnet wird.

b) Öffnen Sie danach das Programm *Codeblock*s und legen Sie z. B. mit Hilfe des *Wizards* ein neues C++ Projekt vom Typ *Console application* an, wobei Sie sich auf die *Release*-Konfiguration beschränken können.

c) Binden Sie in das Projekt die Quelltextdatei *AMS_DriveRandom_template.cpp* ein, die das Hauptprogramm der zufälligen Steuerung des Roboters enthält. Der Linker benötigt zusätzlich die Dateien *AMS_Robot.cpp* sowie *AMS_Robot.hpp*, in denen die Klasse *AMS_Robot* beschrieben ist. Diese Klasse implementiert die Schnittstellen zu *player* und stellt darüber hinaus viele nützliche Methoden zur Verfügung.

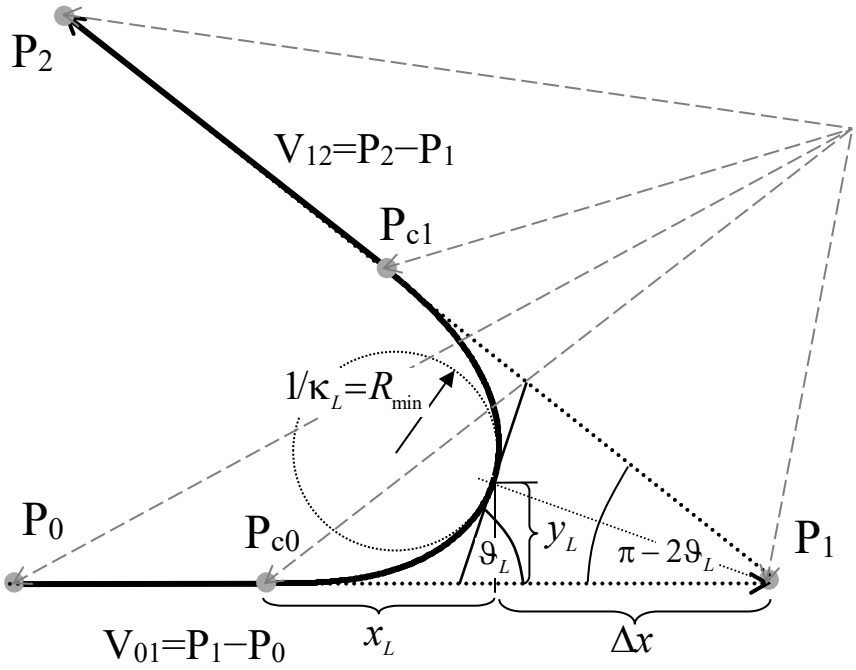
d) Fügen Sie jetzt die Pfade zu den von der Klasse *AMS_Robot* benötigten Bibliotheken und Include-Verzeichnissen in das Projekt ein. Werten Sie dazu die bei der Erstellung des Programms erzeugten Fehlermeldungen aus.

e) Schreiben Sie ergänzenden Quellcode, um den Sollwert für die Winkelgeschwindigkeit abhängig vom Parameter *TurnDist* und von den aktuellen Hindernisabständen zu setzen, so dass der Roboter sich kollisionsfrei in der simulierten Umgebung bewegt.

2. Aufgabe – Krümmungsstetiger Übergang

In dieser Übung soll ein Programm erstellt bzw. erweitert werden, mit dem ein krümmungsstetiger Übergang unter Verwendung von zwei identischen, gespiegelten Klothoiden berechnet und die resultierende Trajektorie in das *Stage*-Fenster gezeichnet wird.

Die zu erstellende Trajektorie werde dazu durch drei Punkte bzw. Vektoren P_0, P_1 und P_2 vorgegeben, siehe nachfolgende Skizze:



Ergänzen Sie die als Vorlage dienende Datei *AMS_CalcClothoid_template.cpp* innerhalb der markierten Abschnitte, und gehen Sie dabei wie folgt vor:

a) Ermitteln Sie den minimalen Krümmungsradius R_{min} der Klothoide aus der maximalen Bahn- und Winkelgeschwindigkeit des Roboters, die Sie mit den Methoden *get_vmax()* und *get_wmax()* aus dem Roboterobjekt auslesen können.

b) Berechnen Sie aus den Punktvektoren P_0, P_1 und P_2 die beiden Vektoren V_{01} und V_{12}, und ermitteln Sie deren Beträge, d. h. deren Längen L_{01} und L_{12}.

c) Bestimmen Sie aus V_{01}, V_{12}, L_{01} und L_{12} unter Verwendung des Skalar- und des Vektorproduktes den Betrag und das Vorzeichen des zur maximalen Krümmung der Klothoiden gehörenden Tangentenwinkels ϑ_L.

d) Berechnen Sie mit den aus der Vorlesung bekannten Formeln die Länge L der Klothoiden, die zum Scheitelpunkt gehörenden Koordinaten x_L und y_L sowie den Abstand *delta* = $x_L+\Delta x$ zwischen den Punkten P_{c0} bzw. P_{c1} und dem Eckpunkt P_1.

e) Ermitteln Sie aus P_0, P_1, V_{01}, V_{12}, L_{01}, L_{12} und *delta* die Vektoren zu P_{c0} und P_{c1}.

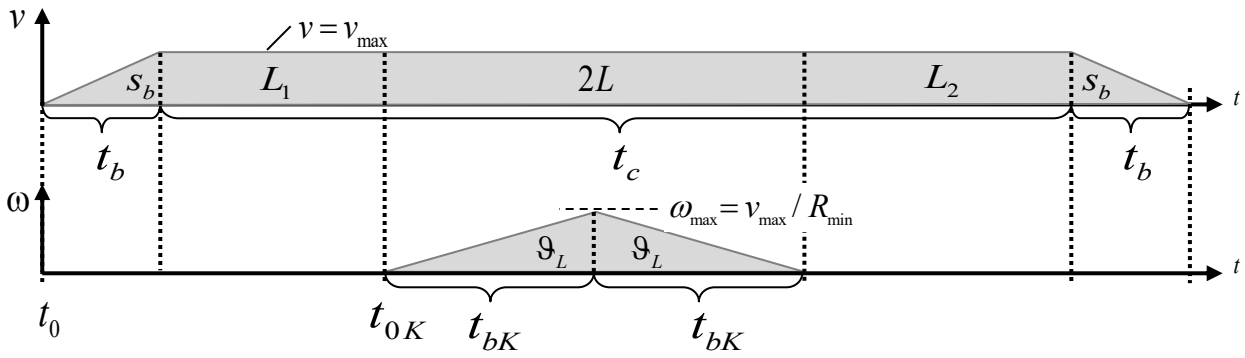
f) Berechnen Sie innerhalb der vorgegebenen *for*-Schleife zunächst die lokalen Koordinaten x_s und y_s und damit unter Berücksichtigung des Startpunktes P_{c0} und einer Rotation um den Winkel φ_1 die globalen Koordinaten der Klothoiden.

Erzeugen Sie ein ausführbares Programm, und lassen Sie den krümmungsstetigen Übergang für verschiedene Koordinaten der Eckpunkte und unterschiedliche Maximalgeschwindigkeiten darstellen.

3. Aufgabe – Bahnsteuerung auf Geraden und Klothoiden

In dieser Übung soll eine Bahnsteuerung für einen nicht-holonomen mobilen Roboter mit Differenzialantrieb implementiert werden, um einen zuvor berechneten krümmungsstetigen Übergang auf Basis von zwei Klothoiden zu befahren.

Der Roboter soll hierbei bis auf eine Beschleunigungs- und Abbremsphase mit konstanter Bahngeschwindigkeit v_{max} bewegt werden, wobei die Winkelgeschwindigkeit während des Befahrens der Klothoiden erst linear bis auf ω_{max} ansteigt, um dann mit derselben konstanten Winkelbeschleunigung wieder bis auf null abzufallen, siehe nachfolgende Skizze, aus der alle relevanten Parameter entnommen werden können:



Erweitern Sie zur Lösung der Aufgabe die als Vorlage dienende Datei *AMS_DriveClothoid_template.cpp* bzw. die darin enthaltene Funktion *DriveRobot()* innerhalb der markierten Abschnitte, und gehen Sie dabei wie folgt vor:

a) Berechnen Sie den Beschleunigungsweg s_b, die Beschleunigungszeit t_b, die Zeitdauer t_c, während der der Roboter mit konstanter Bahngeschwindigkeit v_{max} fährt, sowie die Beschleunigungszeit t_{bK} bis zum Erreichen der maximalen Winkelgeschwindigkeit ω_{max}.

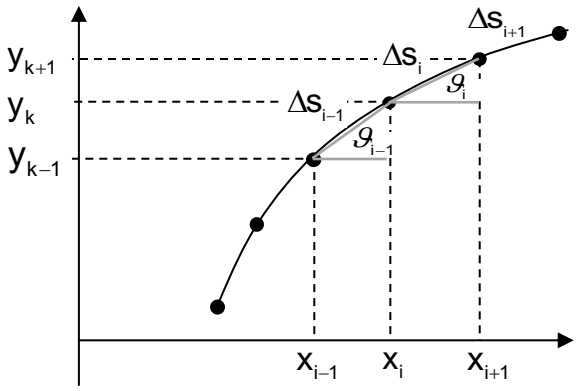
Die Gesamtbewegungszeit des Roboters ist in drei Zeitbereiche aufgeteilt, die jeweils im Programm durch eine *do-while* Schleife gebildet werden. In der ersten und dritten Bewegungsphase, die in dem vorgegebenen Template bereits implementiert sind, fährt der Roboter geradeaus und wird aus dem Stand auf v_{max} beschleunigt bzw. von v_{max} wieder bis zum Stillstand abgebremst. Während der mittleren Schleife fährt der Roboter mit maximaler Bahngeschwindigkeit, dreht sich anfangs jedoch noch nicht.

b) Schreiben Sie ergänzenden Programmcode für die mittlere Schleife, um den Roboter auf den Klothoiden zu führen. Lesen Sie dazu bei jedem Schleifendurchlauf die aktuelle Bewegungszeit ein, und sorgen Sie dafür, dass ab dem Zeitpunkt t_{0K} die Winkelgeschwindigkeit erst linear bis auf ω_{max} erhöht und ab dem Zeitpunkt $t_{0K} + t_{bK}$ wieder bis auf den Wert null erniedrigt wird.

c) Ergänzen Sie Ihr cpp-Projekt, deklarieren Sie *DriveRobot()* in der Datei *AMS_CalcClothoid.cpp* und rufen Sie die Funktion nach Berechnung der Trajektorie auf. Testen Sie dann die Funktionsfähigkeit der Bahnsteuerung und die erzielbare Genauigkeit, indem Sie für verschiedene Trajektorien den Roboter in der *Stage*-Simulation durch Aufruf der Methode *AMS_Robot::set_sim_pos()* an den Startpunkt setzen. Verwenden Sie dabei die Methode *AMS_Robot::set_slip_const()*, um den bei der Bewegung auftretenden Schlupf auf null zu setzen.

4. Aufgabe– Bahnsteuerung auf punktweise definierten Trajektorien

In dieser Übung soll aus dem in Aufgabe 2 berechneten krümmungsstetigen Übergang eine Folge von Wegpunkten berechnet und ein nicht-holonomer mobiler Roboter entlang dieser punktweise definierten Trajektorie gesteuert werden. Der Abstand zwischen benachbarten Punkten der Trajektorie entspricht dabei jeweils der Strecke, die während der konstanten Zeitdauer *T* befahren wird, so dass variable Punktabstände eine Beschleunigung bzw. Abbremsung des Roboters bewirken.



Erzeugen Sie zunächst eine Textdatei *AMS_trajekt.txt* mit den Wegpunkten auf Basis Ihrer Lösung von Aufgabe 2 und der Datei *AMS_CalcClothoid_template.cpp*, die Sie an den markierten Stellen entkommentieren und ergänzen.

a) Speichern Sie dazu einen neuen Wegpunkt immer dann, wenn in den Zeichenroutinen der Abstand des aktuellen Punktes zum jeweils vorherigen Wegpunkt Δs übersteigt, wobei Δs von der aktuellen Geschwindigkeit v_{akt} und der festen Abtastzeit T abhängt. v_{akt} muss innerhalb der Beschleunigungszeit t_b von null auf v_{max} erhöht und beim Erreichen der Bremsstrecke s_b während t_b wieder auf den Wert null reduziert werden.

Die Datei *AMS_DriveTraject_template.cpp* dient als Vorlage für die Bewegungssteuerung, und soll dazu innerhalb der markierten Abschnitte wie folgt erweitert werden.

b) Plotten Sie die in der Datei *AMS_Traject.txt* enthaltenen Wegpunkte mit der Methode *draw_traject()* in das *Stage*-Fenster. Berechnen Sie dann aus den beiden ersten Wegpunkten (x, y) sowie (x_1, y_1) deren Abstand ds_1 sowie den Winkel ϑ_1, und setzen sie den Roboter mit *set_sim_pos()* an den Startpunkt der Trajektorie.

Die in dem Quellcode enthaltene *while*-Schleife ermöglicht ein synchrones Auslesen der Wegpunkte, wofür jeweils zu Beginn die aktuelle Zeit ausgelesen und in t_{ref} gespeichert wird. Am Schleifenende erfolgt eine erneute Zeitmessung, und aus der Differenz zu t_{ref} wird die erforderliche Wartezeit bestimmt, damit ein Schleifendurchlauf exakt T dauert.

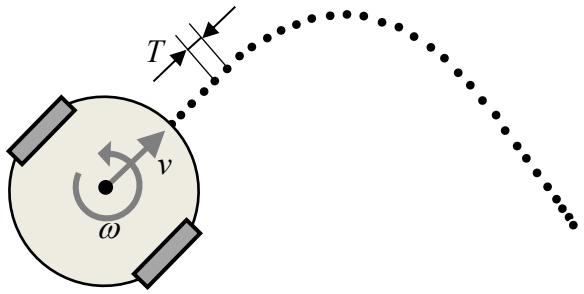
c) Berechnen Sie in der Schleife den Abstand ds und den Winkel ϑ zwischen den beiden zuletzt eingelesenen Punkten und daraus mit ds_1, ϑ_1 und T die Bahngeschwindigkeit v, die Krümmung κ und die Winkelgeschwindigkeit ω, und prägen Sie dann v und ω ein.

d) Jetzt müssen die aktuellen Werte x, y, ds und ϑ in die entsprechenden Variablen mit dem Index 1 umkopiert werden, damit im nächsten Schleifendurchlauf wieder ein neuer Punkt verarbeitet werden kann.

e) Erzeugen Sie ein ausführbares Programm, lassen Sie den Roboter schlupffrei entlang der Bahnkurve fahren, und beobachten Sie die Auswirkung einer Variation von T auf die Güte der Bahnsteuerung.

5. Aufgabe – Bahnregelung für nicht-holonome Roboter

In dieser Übung soll eine Bahnregelung für einen nicht-holonomen mobilen Roboter implementiert werden, um den Roboter möglichst exakt entlang einer als Folge von Punkten in der Datei *Trajekt.txt* vorliegenden Bahnkurve zu führen. Der Abstand zwischen benachbarten Punkten der Trajektorie entspricht dabei jeweils der Strecke, die während der konstanten Zeitdauer T befahren werden soll, so dass variable Punktabstände eine Beschleunigung bzw. Abbremsung des Roboters bewirken.



Die Datei *AMS_ControlTraject_template.cpp* soll als Vorlage zur Lösung diese Aufgabe verwendet, und dazu innerhalb der markierten Abschnitte ergänzt werden. Zu Beginn dieses Programms wird die Trajektorie durch Aufruf der Methode *draw_traject()* gezeichnet, es werden die ersten beiden Punkte als Sollkoordinaten eingelesen, und die aktuelle Roboterposition wird mit dem ersten Punkt initialisiert.

a) Berechnen Sie den Startwinkel ϑ des Roboters vom ersten zum zweiten Punkt und ermitteln Sie die Start-Sollgeschwindigkeiten v_{xs1} sowie v_{ys1}.

In dem Programm werden die jeweils aktuellen Führungsgrößen v_s sowie ω_s für den inneren Geschwindigkeitsregelkreis innerhalb einer zentralen *while*-Schleife ermittelt, in der bei jedem Durchlauf die Sollkoordinaten des jeweils nächsten Punktes eingelesen werden, solange bis der letzte Bahnpunkt erreicht ist.

Die Berechnung erfolgt synchron zur Bewegung des Roboters, weshalb die Zeitdauer für einen Schleifendurchlauf exakt T entsprechen muss. Dies wird dadurch erreicht, dass zu Beginn jedes Durchlaufs jeweils ein Zeitstempel mit Hilfe der Methode *localtime()* aus der Klasse *posix_time* erfasst wird, und am Ende aus der Differenz der aktuellen Zeit von dieser Referenzzeit die erforderliche Wartezeit ermittelt wird.

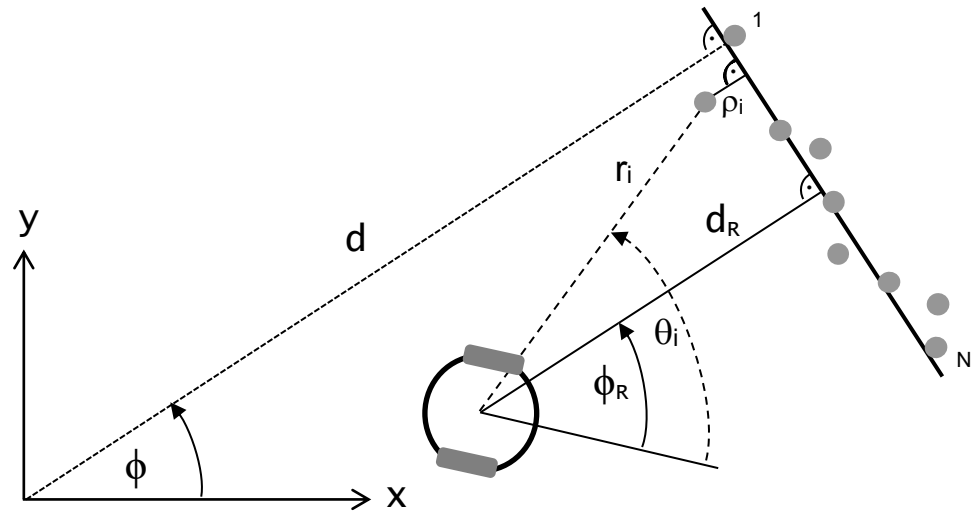
b) Lesen Sie aus dem Roboterobjekt nach der Aktualisierung der Daten die aktuelle Bahn- und Winkelgeschwindigkeit v und ω aus, berechnen Sie damit das Weg- und Winkelinkrement δ und φ, und daraus mit dem nicht-linearen Bewegungsmodell die aktuellen Roboterkoordinaten x, y, und ϑ. Außerdem müssen Sie aus v und ϑ auch die aktuellen Geschwindigkeitskomponenten v_x und v_y des Roboters bestimmen.

c) Ermitteln Sie aus den Koordinaten des aktuellen und des vorherigen Punktes die Sollgeschwindigkeiten v_{xs} und v_{ys}, sowie aus den Sollgeschwindigkeiten die Sollwerte für die Beschleunigungen a_{xs} und a_{ys}. Aus den Ist- und den Sollwerten können jetzt die aktuellen Fehler berechnet werden, und daraus die Stellgrößen v_{st} sowie ω_{st}.

d) Erzeugen Sie ein ausführbares Programm, und setzen Sie die beiden Regelparameter k_I und k_{II} auf geeignete Werte, damit der Roboter möglichst exakt der vorgegebenen Trajektorie folgt. Welchen wesentlichen Vorteil weist die Bahnregelung gegenüber einer Bahnsteuerung auf?

6. Aufgabe – Wanddetektion durch lineare Regression

Ziel dieser Übung ist die Detektion einer geraden Wand anhand eines vom Roboter aufgenommenen Entfernungsprofils. Dazu sollen mittels linearer Regression die lokalen und globalen Geradenparameter d_R, ϕ_R und d, ϕ sowie als Gütemaß der mittlere quadratische Fehler σ_ρ^2 aus den N Messpunkten berechnet werden, siehe nachfolgende Grafik:



Die Datei *AMS_DetectWall_template.cpp* soll als Lösungsvorlage für diese Aufgabe verwendet und dazu innerhalb der markierten Abschnitte ergänzt werden.

Zu Beginn dieses Programms werden mit der Methode *get_scan()* die aktuellen Messwerte r_i des Entfernungssensors in ein Array der Größe 360 eingelesen. Das Auslesen der Entfernungen erfolgt über den Zeiger **scan*, wobei der Feldindex dem jeweiligen Messwinkel θ_i mit einer Auflösung von 1° entspricht. Außerdem liefert *get_scan()* die maximale Messdistanz sowie den Messfehler des Sensors definiert durch die Standardabweichung σ_r .

a) Zur schnelleren Berechnung soll der Zugriff auf die verfügbaren Messwerte mittels einer *Look-Up-Table* erfolgen. Implementieren Sie dazu eine Schleife über sämtliche Winkel und speichern Sie die Indizes der N verfügbaren Messpunkte in dem Feld *LUT*. Bestimmen Sie dabei gleichzeitig die Anzahl N und verlassen Sie für $N=0$ das Programm mit einer entsprechenden Fehlerausgabe.

b) Berechnen Sie jetzt mittels der aus der Vorlesung bekannten Formeln den Normalenwinkel ϕ_R und den Normalenabstand d_R der Regressionsgeraden.

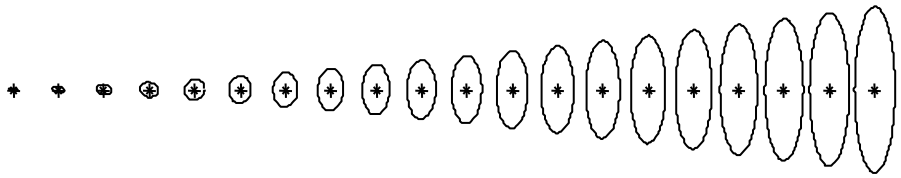
c) Bestimmen Sie den mittleren quadratischen Fehler σ_ρ^2, und legen Sie für σ_ρ^2 einen sinnvollen Schwellwert fest, um zu entscheiden, ob eine signifikante Gerade im Scan erkannt wurde.

d) Berechnen Sie nun unter Verwendung der Roboterkoordinaten x, y und ϑ, die Sie aus dem Simulator auslesen können, die globalen Geradenparameter d und ϕ. Korrigieren Sie die Werte, falls d bzw. d_R negativ sein sollten, und geben Sie die ermittelten Geradenparameter einschließlich des zugehörigen Fehlers auf dem Bildschirm aus.

e) Testen Sie das Verfahren, indem Sie den Roboter an verschiedenen Orten in der Karte positionieren. Wie wirkt sich eine Veränderung des Messfehlers σ_r mit der Methode *set_sigma_ranger()* aus, und welche Auswirkung hat eine Variation der maximalen Messdistanz in der *World*-Datei von *Stage*? Was geschieht, wenn das vom Roboter erfasste Entfernungsprofil mehrere Wände enthält? Erweitern Sie das Programm, so dass auch in diesem Fall eindeutige Geradenparameter ermittelt werden können.

7. Aufgabe – Prädiktionsmodell mit Darstellung von Fehlerellipsen

In dieser Übung soll die Systemkovarianzmatrix *P* des jeweils aktuellen Roboterzustands prädiziert und die Kovarianzmatrix der Koordinaten *x* und *y* als Fehlerellipse um den Erwartungswert der Roboterposition gezeichnet werden, wobei die Fläche der Ellipse die jeweils wahrscheinlichsten Aufenthaltsorte des Roboters umfasst.



Zur Lösung sollen zwei Methoden der Klasse *KalmanFilter* auf Basis der vorliegenden Datei *AMS_KalmanFilter_template.cpp,* die dazu innerhalb der markierten Abschnitte entsprechend der folgenden Teilaufgaben zu ergänzen ist, implementiert werden. Die Deklaration der Klasse ist in der Headerdatei *AMS_KalmanFilter.hpp* gegeben, für die benötigten Algorithmen der linearen Algebra verwenden Sie die Bibliothek *newmat*.

Im Konstruktor der Klasse *KalmanFilter* wird der übergebene Zeiger auf ein Roboterobjekt gespeichert und die Matrix *D* mit der Kinematik des Differenzialantriebs erzeugt. Außerdem erfolgt dort die Initialisierung der Schlupfkonstante k_s , die zur Berechnung der Eingangskovarianzmatrix *Q* dient.

Zunächst soll die Methode *PredictCov()* wie folgt erweitert werden:

a) Speichern Sie die übergebenen Werte für δ und φ im Vektor *u* und ermitteln Sie daraus mit der Inversen von *D* die Weginkremente der beiden Räder im Vektor u_{rl}. Besetzen Sie damit die Kovarianzmatrix Q_{rl} und bestimmen Sie daraus mit *D* die Kovarianzmatrix der Eingangsgrößen *Q*.

b) Berechnen Sie entsprechend des linearisierten Robotermodells die Systemmatrix *A* sowie die Eingangsmatrix *B* im jeweils aktuellen Schritt. Prädizieren Sie daraus und mit *Q* die Systemkovarianzmatrix *P* und geben Sie unter Verwendung des in *newmat* überladenen Ausgabeoperators << die Elemente von *P* auf dem Bildschirm aus.

Anschließend ist in *PlotEllipse()* die Fehlerellipse zu berechnen und zu zeichnen:

c) Kopieren Sie die Kovarianzmatrix von x und y als Teilmatrix von *P* in die Matrix P_1 , berechnen Sie mit *newmat* die Eigenwerte und –vektoren von P_1 , wobei diese jeweils in den Matrizen *L* und *T* gespeichert werden sollen.

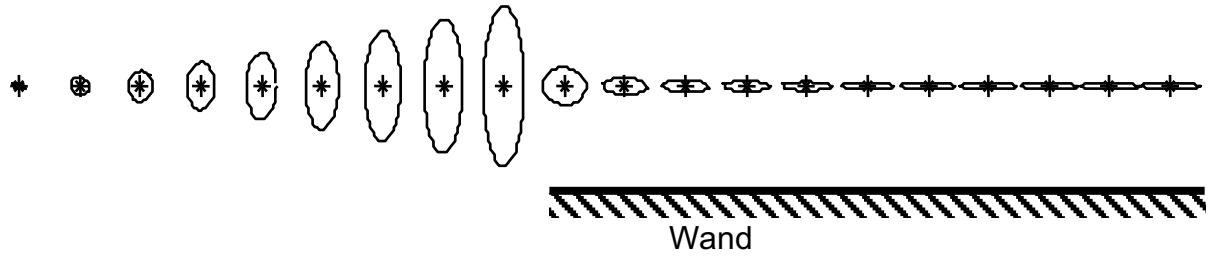
d) Berechnen Sie in einer Schleife mit den Eigenwerten die Ellipsenkoordinaten x_s und y_s in Hauptachsenform abhängig vom Parameter α mit $0 \le \alpha \le 2\pi$; transformieren Sie dann mit *T* und dem übergebenen Mittelpunkt (x_m , y_m) die Ellipse in die globalen Koordinaten *x* und *y*, und kopieren Sie zum Zeichnen die Werte in das Objekt *ellipse*.

Jetzt kann das Fehlermodell in die Bahnregelung des Roboters integriert werden:

e) Generieren Sie in der Datei *AMS_ControlTraject.cpp* ein Objekt der Klasse *KalmanFilter* und übergeben Sie den Zeiger auf das Roboterobjekt. Rufen Sie dann bei jedem Schleifendurchlauf die Methode *PredictCov()* und bei jedem fünften Durchlauf zusätzlich die Methode *PlotEllipse()* auf. Setzen Sie dann in *main()* mit der Methode *set_slip_const()* den simulierten Schlupf auf einen Wert von *0.005*, lassen Sie den Roboter mehrmals auf der vorgegebenen Trajektorie fahren und bewerten Sie die Güte der Positionsschätzung. Variieren Sie danach in der Klasse *KalmanFilter* die Parameter *b* und k_s und beobachten Sie die Auswirkung auf das Fehlermodell.

8. Aufgabe – Korrektur von Roboterpositionen mittels Kalman-Filter

Ziel dieser Übung ist die Implementierung des Korrekturschrittes eines Kalman-Filters, wobei vom Roboter gemessene Ausrichtungen und Abstände umgebender Wände mit den als Karteninformation vorliegenden exakten Wandparametern verglichen werden, um damit unter Berücksichtigung von Mess- und Prädiktionsfehlern eine exakte Positionierung des Roboters durchführen zu können.



Dazu soll eine zusätzliche Methode *Correction()* der Klasse *KalmanFilter* implementiert bzw. erweitert werden, wobei die Lösung der vorherigen Übungsaufgabe auf Basis der gegebenen Datei *AMS_KalmanFilter_template.cpp* vorausgesetzt wird. Für die benötigten Algorithmen der linearen Algebra verwenden Sie auch hier die Bibliothek *newmat*.
Zur Suche nach einer zur aktuellen Messung korrespondierenden Wand in der Karte dient darüber hinaus die in A*MS_WallMap.hpp* sowie *AMS_WallMap_template.cpp* deklarierte bzw. teilweise implementierte Klasse *WallMap*.

Zunächst wird in *Correction()* durch Aufruf der Methode *detect_wall()* aus der Klasse *AMS_Robot* versucht eine Wand im aktuellen Scan zu finden. Bei Erfolg werden die Parameter d_R, ϕ_R sowie deren Kovarianzmatrix R zurückgegeben.

a) Ermitteln Sie aus d_R und ϕ_R mit der aktuellen Roboterposition die globalen Parameter d und ϕ der gemessenen Wand. Implementieren Sie dann die *search()*-Methode der Klasse *WallMap* zur Suche einer korrespondierenden Geraden in der Karte, wobei Sie auch die Messmatrix H festlegen und die Kovarianzmatrix S zur Übergabe an *search()* bestimmen müssen. Der maximal zulässige quadratische Abstand *rsq_max* wird bei Aufruf übergeben, der minimal gefundene Wert soll zurückgegeben werden.

b) Bestimmen Sie mittels des linearen Messmodells die Messgrößen x_i^ϕ und ϑ_i im aktuellen Schritt i und speichern Sie diese im Messvektor y. Berücksichtigen Sie bei der Festlegung von y eine geeignete Fallunterscheidung abhängig von der prädizierten x-Koordinate in ϕ-Richtung.

c) Berechnen Sie jetzt aus H, R und der prädizierten Systemkovarianzmatrix P den Kalman-Gain K im aktuellen Schritt. Führen Sie anschließend das Mess-Update des Roboterzustands durch, wozu Sie die Roboterkoordinaten in den Vektor x und nach der Korrektur wieder zurückkopieren; korrigieren Sie danach ebenfalls P.

Anschließend kann die Funktion des vollständigen Kalman-Filters getestet werden:

d) Erweitern Sie die Datei *AMS_ControlTraject.cpp*, indem Sie nach jedem Prädiktionsschritt zusätzlich die Methode *Correction()* aufrufen. Setzen Sie dann in *main()* mit *set_slip_const()* den simulierten Schlupf auf 0.005 und mit *set_sigma_ranger()* die Standardabweichung $\sigma_r = 0.01$. Lassen Sie den Roboter mehrfach auf der Trajektorie fahren. Geben Sie dabei die prädizierten und korrigierten Zustandsparameter sowie die gefundenen Wände mit ihrem Abstandsmaß *rsq_max* aus, und beobachten Sie die Fehlerellipsen. Welche Auswirkung hat eine Variation des Schlupfes sowie von σ_r auf die Fehlerellipsen und die Güte der Lokalisierung?