Odometrie oder die Kunst einen Weg für den Roboter zu finden:

Definition Odometrie laut Wikipedia:

Odometrie oder auch **Hodometrie** (von griech. *hodós*, „Weg“ und *métron*, „Maß“ - also: „Wegmessung“) ist die Wissenschaft von der Positionsbestimmung eines mobilen Systems anhand der Daten seines Vortriebsystems. Durch Räder angetriebene Systeme benutzen hierzu die Anzahl der Radumdrehungen, während laufende Systeme (z. B. Roboter) die Anzahl ihrer Schritte verwenden. Ein Gerät, das die Odometrie zur Positionsbestimmung verwendet, ist ein Odometer. Die Odometrie ist im Zusammenspiel mit der Koppelnavigation ein grundlegendes Navigationsverfahren für bodengebundene Fahrzeuge aller Art (Kfz, Roboter), allerdings wird es auf Grund seiner Fehlereigenschaften selten als alleiniges Verfahren eingesetzt.

Schnell und einfach erklärt, aber was nun?
Klar, schaut man doch mal schnell unter Google nach, wer da mal was programmiert hat.

Aha ja, doch soviel ☺

Ok, ein Filter muss her:

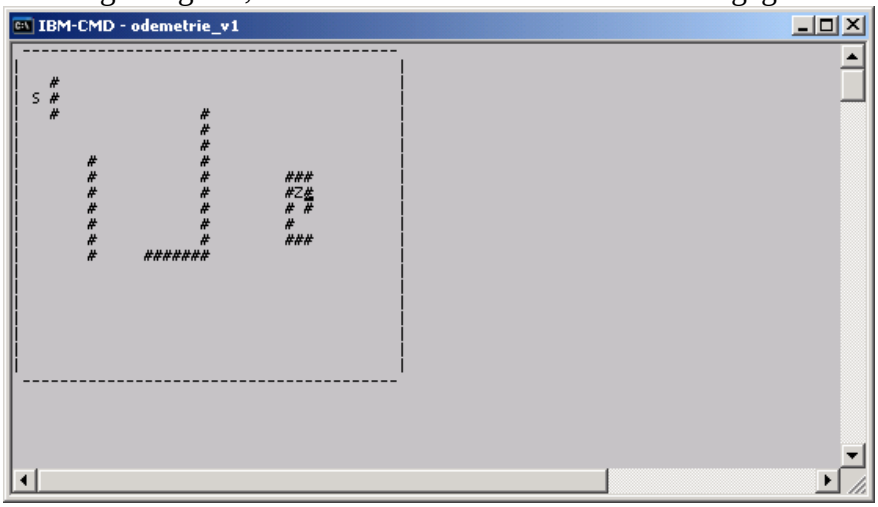
- Ich mag nun mal klassisches C++ → Immer noch nichts richtiges
- Hmm, Implementierung, Programmierung, „Pathfinding“

Die Liste könnte man nun recht lange ausführen.

Richtig gute Erklärungen findet man z.B. an Universitäten:
Ausarbeitungen, Diplomarbeiten und allg. Studienunterlagen. Am Ende dieses Berichtes werde ich einige gute Ausarbeitungen auflisten, die ich selbst als sehr hilfreich empfunden habe.

Nun, die Verzweiflung war groß, also selbst ist der Mann, kann doch nicht schwer sein ein Programm zu schreiben bei dem ein Weg um Hindernisse gefunden wird.

Mittlerweile war mir klar, ich probier das mal zuerst am PC mit einem DOS- Fenster als Testumgebung aus, bevor ich den Roboter einmal zu oft gegen die Wand fahre ☺



Gemein wie ich mir das so vorstellte, war das Ziel „Z“ schön hinter den Hindernissen versteckt.

Hoppla, doch ein paar viele **IF – IF ELSE - etc…** Abfragen, um ein „X“ von Startpunkt „S“, um ein paar Hindernisse herum zu manövrieren. Geschweige denn, dass der Algorithmus, oder besser die Abfragen noch funktionieren wenn man mal das Hindernis in der Lage verändert.

ALLGEMEINE ERNÜCHTERUNG FOLGTE:

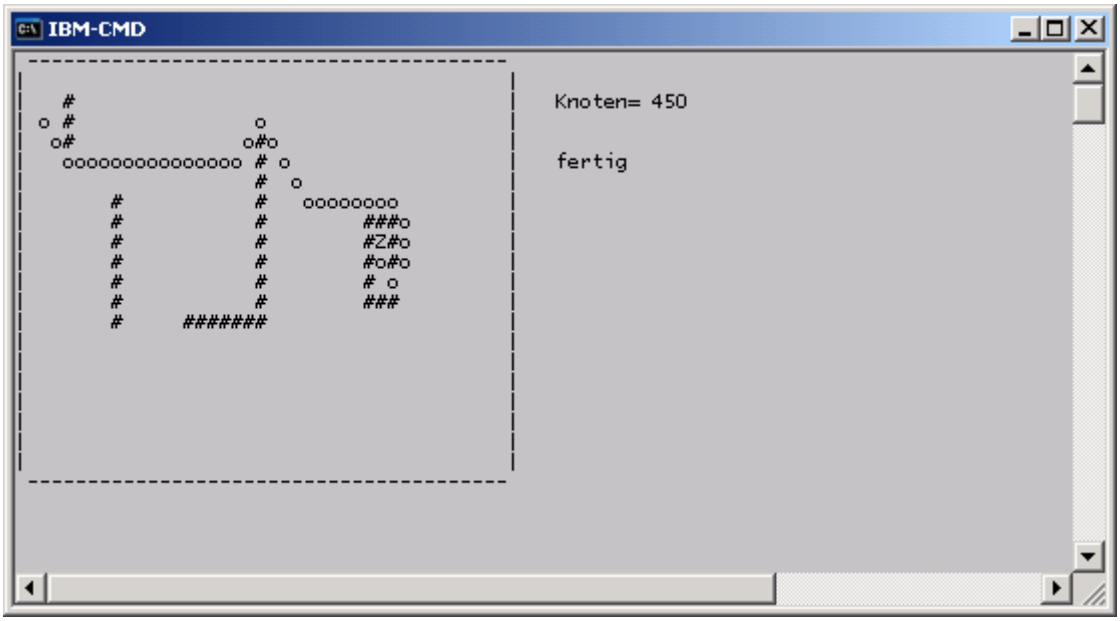
Also zurück ins Netz und Google. Ahhhhh, überall Hinweise die auf Pathfinding mit einem „A-Star“ Algorithmus hindeuten.

Hääääh. Hey, ich bin 45 Jahre alt, arbeite hauptberuflich in einer Bank als Controller und meine C++ Kenntnisse und Syntax sind aus dem Jahr 1994, als ich meine Diplomarbeit geschrieben habe und nun soll ich verstehen was die Studenten von heute so programmieren. Übersetzer bitte …
Also vergessen wir mal schnell die fertigen Programme (Source-Code) aus dem Internet.

Okay, man nehme sich die Algorithmusbeschreibung und programmiert das ganze mal in seinem eigenen Stil.
Das hat dann auch geklappt und den Sourcecode habe ich an den Artikel angehängt
U N D in meiner alten, schönen Syntax programmiert, so wie halt vor 20 Jahren hä, hä, hä ...

Ach ja, ob ich die Algorithmusbeschreibung richtig umgesetzt habe weiß ich nicht, aber mein PC Beispielprogramm hat einen Lösungsweg berechnet. Ist doch schon mal was und eine einfache C-Sourcecode Lösung lässt sich einfacher auf einen Mikrocontroller transferieren.

Gut: Wir haben jetzt eine Lösung, und sieht doch richtig hübsch aus, wie der PC den Weg zum Ziel findet. Berechnet wurden 450 Knoten.



Knoten, da war doch noch was:
Der Algorithmus funktioniert nur, wenn der Weg von Startpunkt zu Zielpunkt in Knoten, oder auch Felder genannt, eingeteilt ist. Diese Knoten können Rechtecke sein oder auch Dreiecke, ist eigentlich egal.

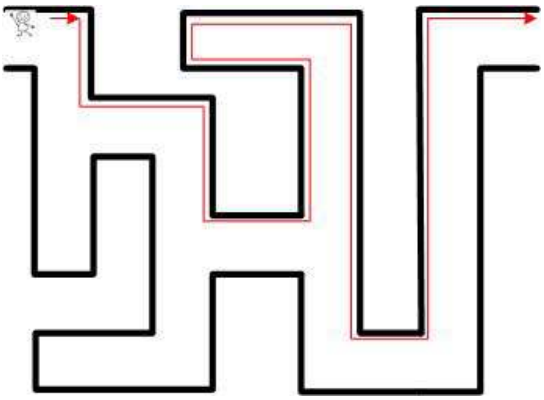
☹ O D E R doch nicht ☹

Jetzt kommt nämlich der Teil mit dem Roboter und dem PC. Also auf einem PC war das ganz einfach zu programmieren (Naja, einfach…), eine zweidimensionale Matrix mit Feldern gefüllt „#“ als Hindernis und seitliche Begrenzungen / Ränder die den Wirkungsbereich eingrenzen.

Jetzt muss ich einmal sagen, dass meine „Life“ Testumgebung, auch verächtlich von meiner Freundin Wohnzimmer genannt, nun mal nicht in Felder eingeteilt ist. Ein Versuch der von meiner Freundin, unter Androhung eines Platzverweises, sabotiert wurde.

Was nun, wie erkenne ich den die Hindernisse?
Wo ist mein Startpunkt?
Wo mein Ziel?

Der Roboter hat nun mal nur Bumper und IR-Sensoren. GPS in der Wohnung geht auch nicht. Also, entweder muss ein anderer Algorithmus her, oder die Möglichkeit, die Position des Roboters, des Ziels und der Hindernisse genau zu lokalisieren.
Es gibt solche Möglichkeiten z.B. IR Baken, oder eine Kamera an der Decke, die die Position der Hindernisse und des Roboters bestimmt. Klar, nur wie bekomme ich z.B. die Daten in den Roboter, der ja eigentlich autonom arbeiten soll.
Also wieder das Internet befragt, bzw. hatte ich in einigen der Ausarbeitungen etwas von Algorithmen gelesen, die ohne Knoten auskommen, dann aber nur die Hindernisse umfahren und keinen optimalen Weg suchen von Start zum Ziel.
Also fahren wir mal um ein Hindernis herum war das neue Ziel der Programmierung. Erst mal wieder auf dem PC verwirklichen, soweit war ich dann schon mal in den Überlegungen. Doch wie sollte es weitergehen.
Schließlich wurde ich fündig und der einfachste der Algorithmen mit dem hübschen Namen BUG2 (Bug engl. Kakerlake oder auch Fehler) hatte es mir angetan. Es gibt weitaus bessere Algorithmen, die aber auch wesentlich komplexer sind.
So als Abfallprodukt, beim lesen der Ausarbeitung gelernt: Wie kommt man eigentlich aus einem Labyrinth heraus? Ganz einfach z.B. die rechte Hand immer an der Wand des Labyrinth vorbeiführen, somit ständiger Kontakt mit der rechten Wand herrscht. Dauert halt lange, aber man kommt aus dem Labyrinth heraus.
Die Frage meiner Tochter (13 Jahre) gestellt, wie kommt man aus einem Labyrinth heraus. „Mann, Papa“ weiß doch jeder, rechte Hand an der Wand halten.



Woher wissen die Kids das heute so einfach, war vielleicht doch eine PISA-Frage ?!?

Der BUG2 Algorithmus,
oder die Suche nach dem nicht immer optimalen Weg zum Zielpunkt.

Wie komme ich eigentlich von Punkt A nach Punkt B ohne eine Karte. Treffer und versenkt ! Hier wir es nämlich richtig schwierig, da wir uns auf dem Feld der Navigation tummeln müssen.
Im Klartext, ich habe wieder das Problem, dass ohne ein GPS eigentlich nix richtig funktioniert.

ODER DOCH ?!

Es gab auch mal eine Zeit vor GPS und da hat man so ein komisches Ding mit Zahlen und einem immer nach Norden zeigender Nadel benutzt. Richtig, einen Kompass…
Einen elektronischen Kompass für den Robby gibt es schon und auch gleich mit I²C Schnittstelle zum Datenaustausch.

So frisch ans Werk, was haben wir denn nun… Einen Kompass und einen Kompass ???
Ich kann also den Roboter genau in der Testumgebung (Wohnzimmer) in Richtung 180° fahren lassen, was dann Süden wäre.
Toll, aber irgendwie auch ernüchternd, denn der Kompass weist nur die Richtung aber, verflixt noch mal wann soll der Roboter den aufhören zu fahren und zurück melden, Ziel erreicht.
Die einfache Lösung, solange fahren bis er gegen das Ziel fährt … schlecht, wenn das z.B: eine Mingvase wäre, oder man doch die Katze über den Haufen fährt … da ist man dann schnell wieder bei dem Punkt Platzverweis!
Was kann denn nun weiterhelfen. Hier kommen wieder die IR-Baken zum Einsatz. Wie bei der Navigation von Schiffen benötigen wir vermessene Fixpunkte (In der Seefahrt, markante Punkte an der Küstenlinie wie Leuchttürme oder Kirchtürme). Solche Punkte sind z.B. in Seekarten eingetragen.
Die Umsetzung dieses Themas kommt in dem Teil zwei, wo wir uns mit der Navigation näher beschäftigen wollen. Jetzt zurück (ach ich liebe diese Abschweifungen) zum Problem der Realisierung eines BUG2 Algorithmus auf dem PC.

Allgemeine Beschreibung des BUG 2 Algorithmus

nach Vladimir Lumelsky & Alexander Stepanov: Algorithmica 1987

Vorraussetzungen:

1. Die Richtung (hier oft auch m-Line genannt) zum Ziel muss bekannt sein, z.B. durch einen Kompass

2. Es sind entsprechende Sensoren am Roboter vorhanden, die ein Hindernis durch Berührung oder auf Entfernung durch z.B. Infrarotsensoren erkennen können

3. Die Erkennung mittels IR Sensoren muss linear sein z.B. 10 – 80 cm entspricht einer Spannung von z.B. 1 – 5 Volt am Ausgangsport, sonst wird's etwas holperig, aber nicht unlösbar

Grundsätzliches Vorgehen beim BUG 2 Algorithmus:

– Bewege dich gerade auf einer Linie Richtung Ziel

– Folge immer der Wand des Hindernisses (z.B. rechts herum)

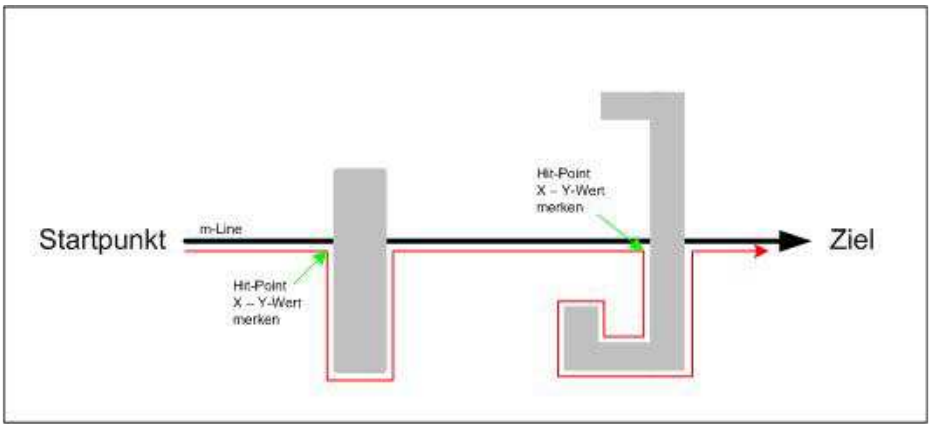
Algorithmus-Beschreibung:

1) Folge der m-Line in Richtung des Ziels. Bewege dich gerade auf einer Linie Richtung Ziel

2) Wenn ein Hindernis im Weg ist, umfahre das Hindernis (Folge der Wand) bis Du wieder auf die m-Line triffst, welcher näher zum Ziel ist, sonst Abbruch

3) Verlasse das Hindernis und folge der m-Line weiter in Richtung Ziel

Prinzipelle Vorgehensweise als Zeichnung:



Der Roboter fährt um das Ziel herum bis er wieder auf die m-Linie trifft. Von dort aus weiter zum Ziel.

Das wäre der BUG2 Algorithmus.

Der Vorgängeralgorithmus BUG1 wurde so entworfen, dass er erst mal das ganze Hindernis umrundet, sich den Punkt merkt, wo er wieder auf die m-Linie trifft und von dort dann bei der zweiten Umrundung von dort weiter in Richtung ziel fährt.
Die gefahrene Strecke ist somit größer als beim BUG2 Algorithmus.
In den Ausarbeitungen (siehe Literaturliste) wird dann noch auf die Vor –und Nachteile der einzelnen Algorithmen eingegangen.
Interessiert uns ja nicht, da wir den BUG2 Algorithmus gewählt haben ☺

Schauen wir uns nun mal die genannten Vorraussetzungen im Einzelnen an:

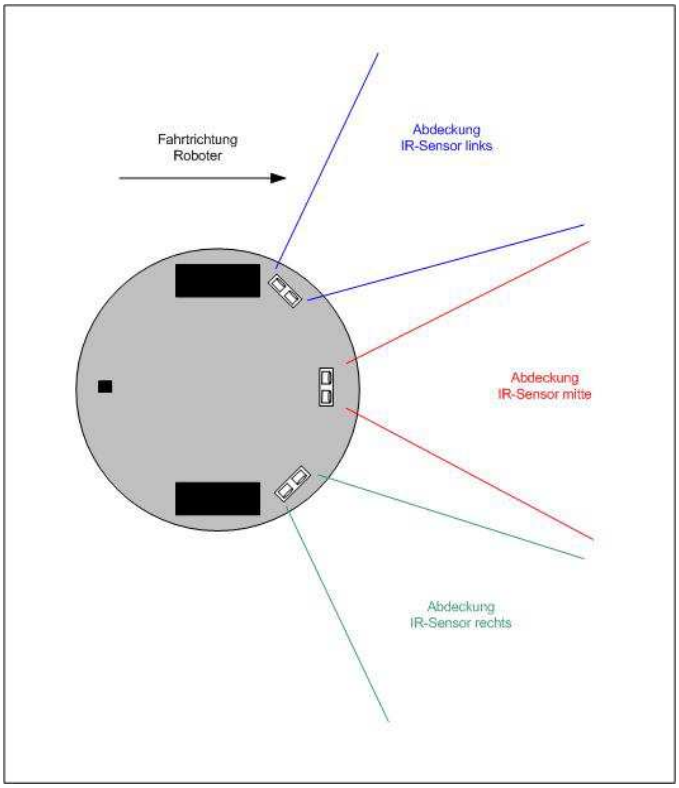
Richtungsvorgabe (m-Linie). Hier hatten wir einen Kompass gewählt. Glück gehabt, gibt es anschlussfertig zu kaufen.

Kompassmodul CMPS03

Entfernungsmesser z.B. Infrarotmesser. Wichtig ist für meine Implementierung, dass der Entfernungsmesser analog ist, also die Entfernung misst und in Volt ausgibt.

Entfernungsmesser GP2D120 (Analog)

Schauen wir uns die IR-Entfernungsmessung etwas genauer an. Im Bild wird die Funktionsweise deutlich. Ein Erkennen und anschließende Abtastung des Hindernis ist für den Roboter somit möglich.



In der Zeichnung des Roboters wird klar, dass man ständig die drei Sensoren abfragen muss, ob und wie weit das Hindernis entfernt ist. Ebenfalls müssen zwei weitere Parameter gespeichert, bzw. ausgewertet werden, bei der Veränderung der Richtung des Roboters. Die Richtung in die sich der Roboter auf Grund der Ergebnisse, die die IR-Sensoren liefern bei der Abtastung nach Hindernissen, bewegt. Ebenfalls benötigen wir die Information was die letzte gespeicherte Richtung war bevor ich eine neue Richtung einschlage.

WARUM: Bewege ich mich vom Ziel weg, ist es wichtig zu wissen
- wo her ich komme
- wohin gehe ich.

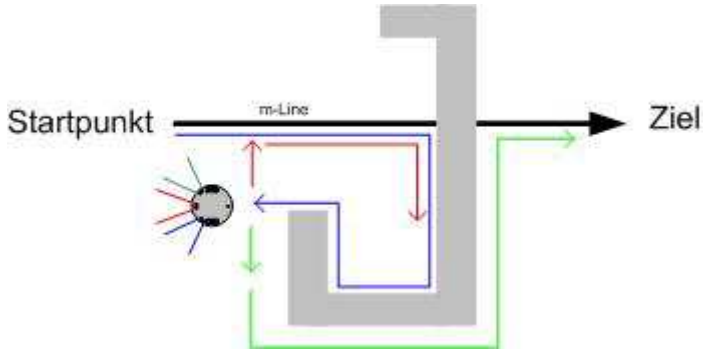
Ich denke das folgende Bild sollte das Problem erklären.

1) Der Roboter „sieht“ den Weg vor sich frei, kein Sensor signalisiert ein Hindernis.

2) Es ist klar, dass sich der Roboter drehen muss. Das kann man intelligent, wenn man weiß, dass die letzte Drehung in Richtung links war und nun nach links weiter muss, obwohl unsere Definition sagt, fahre per Default rechts herum.

3) Oder halt ohne die letzte Drehrichtung zu kennen auf den Defaultwert zurück zu fallen und nach rechts drehen. Damit landet man aber in einer Endlosschleife …

4) Endlosschleifen umgehen kann man auch, in dem man sich den Punkt merkt, an dem man die m-Line verlassen hat, um das Hindernis zu umfahren (manchmal auch HitPoint genannt). Also merken wir uns die Position beim Verlassen der m-Line. Im Computer-Testprogramm ganz einfach: HitPoint = X-Achsenwert (siehe Programm)



Nun muss man sich klar werden, welche Sensoren auf dem in blau eingezeichneten Weg an welchen Ecken angesprochen werden.
Dazu ist es wichtig zu wissen aus welcher Richtung ich eigentlich komme. Im Computer-Testprogramm ist das die Variable „move_dir“.
Eine weitere wichtige Variable ist „last_dir“. Diese Variable speichert den letzten Richtungswechsel.

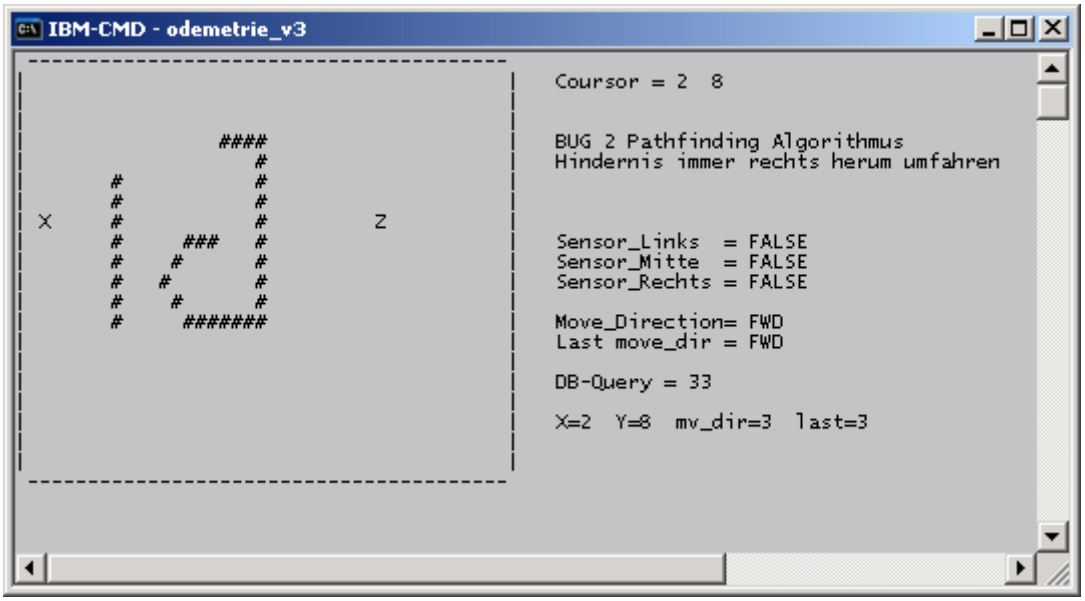
Somit müssen folgende Variablen ausgewertet werden, beim Umfahren eines Hindernisses:

Sensor_links, **Sensor_mitte**, **Sensor_rechts**, **move_dir** und **last_dir**

JETZT geht’s ans Eingemachte:

Vorstellungskraft ist gefragt und logisches Denken

Für das folgende Beispiel, siehe DOS-Box (auch hier gilt Drehung per Default nach rechts) würden die drei Sensoren in folgender Reihenfolge angesprochen:
Bewegung von „S“ dem Startpunkt nach dem Zielpunkt „Z“



Am einfachsten mit dem Finger den Weg des Roboters folgen auf der gedachten Line zwischen Startpunkt und Zeil (m-Linie) und sich die angesprochenen Sensoren und die Richtungsänderungen aufschreiben.

Hier ein Beispiel, wie so eine Tabelle aussehen kann

BUG2 Eingangsmatrix zur Richtungsbestimmung

	Sensoren			Auswerten der Move-Variablen	
Nr	links	mitte	rechts	move_dir	Last_Move
1	0	1	0	0	FWD
2	1	0	0	RIGHT	FWD
3	0	0	0	RIGHT	FWD
4	1	0	0	FWD	RIGHT
5	0	0	0	FWD	RIGHT
6	1	0	0	LEFT	FWD
7	1	1	0	RIGHT	FWD
8	1	0	0	BWD	RIGHT
9	1	1	0	BWD	RIGHT
10	1	0	0	LEFT	BWD
11	1	1	0	LEFT	BWD
12	1	0	0	FWD	LEFT
13	0	0	0	FWD	LEFT
14	1	0	0	LEFT	FWD
15	0	0	0	LEFT	FWD
16	1	0	0	BWD	LEFT
17	0	0	0	BWD	LEFT
18	1	0	0	RIGHT	BWD
19	0	0	0	RIGHT	BWD

Eine solche Matrix kann man z.B. durch ein zweidimensionales Feld (Array) erzeugen.
Im Source-Code ist das die Gewichte-Matrix und ist im File „bug2Defaults.h“ zu finden.

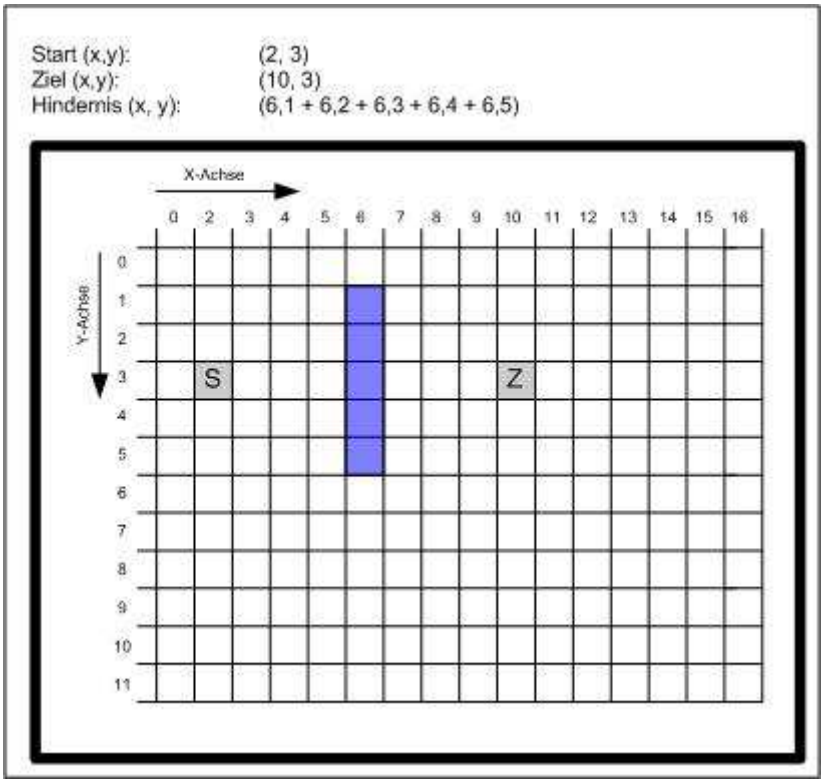
```
 // ********************** Gewichtsmartix ************************************
// in dieser Matrix wird die Reaktion auf Zustaende der Sensoren gespeichert
// einfach gesagt, was soll der Roboter tun wenn Zustand x an den Sensoren anliegt
// Vorraussetzung, dass Hindernis wird  IMMER  recht herum umfahren !!!
//
int          gewicht [25][10] =                 // 1=LEFT, 2=RIGHT, 3=FWD, 4=BWD
            {
              {0,1,0,3,3,5,6,2,3},
              {0,0,0,2,3,6,5,3,2},
              {0,0,0,3,2,5,4,1,3},
              {0,0,0,3,1,5,4,1,3},
              {0,0,0,1,3,4,5,4,1},
```

Abfrage der Sensoren im Computer Testprogramm:

Da sich beim Testprogramm der Roboter ja nicht wirklich dreht muss das durch eine modifizierte Abfragetechnik der Sensoren erfolgen. Es wird durch die Variable „move_dir“ simuliert in welche Richtung sich der Roboter drehen würde und welche Werte in der DOS-Box abgefragt werden müssen.
Gut zu Wissen, dass die Computer Y-Achse von oben nach unten läuft und nicht wie im Kartesischenkoordinatensystem von unten nach oben!

Definition:
Der Roboter kann sich NICHT Diagonal auf dem X, Y Feld bewegen

Schauen wir uns die gezeigte DOS-BOX mal näher an. Die folgende Zeichnung ist ergänzt um die X -und Y-Achsenwerte, zur besseren Nachvollziehbarkeit des Sourcecodes.



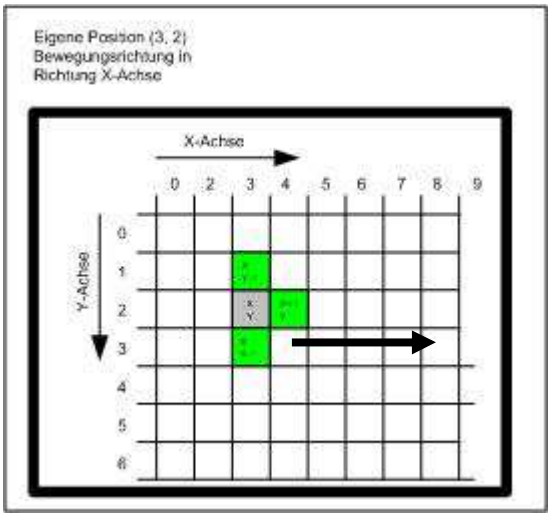
Was heißt das nun für die Abfrage der Sensoren ?

Zwei Beispiele zu verdeutlichen:

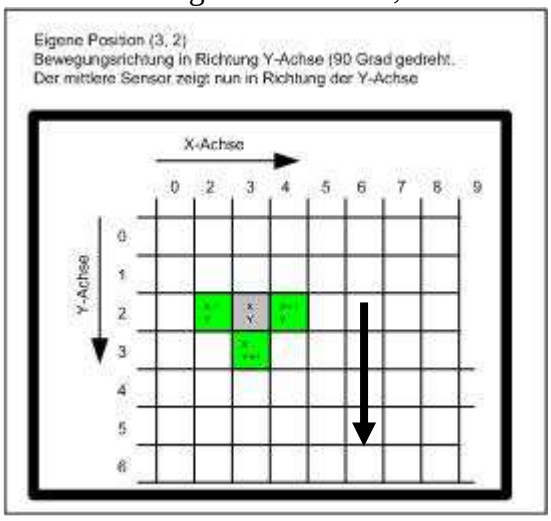
Roboter bewegt sich in Richtung der X-Achse

- Abfrage sensor_mitte Y, X + 1
- Abfrage sensor_links Y-1, X
- Abfrage rechter Sensor Y+1 , X

Klar ?
OK, mal in Worten. Schaut man vom Roboter in Richtung X-Achse, dann ist das Feld vor mir als derzeitige Position X plus ein Kästchen weiter von Interesse.
Das linke Kästchen wäre dann, Y-1 und das rechte Y+1



Nun die Abfrage der Sensoren, wenn sich der Roboter in Richtung Y-Achse bewegt.



BITTE BEACHTEN:
Diese Abfragen gelten nur für unser kleines Testprogramm auf dem PC. In der Realität dreht sich ja der Roboter mit den Sensoren ☺ ☺ ☺

OK, wo stehen wir auf dem Monitor, wo wollen wir hin ist jetzt klar, aber wie fragen wir denn das ARRAY mit den Vorgaben ab, wie sich der Roboter verhalten soll, wenn er auf ein Ziel trifft.
Hier hilft uns die Mathematik weiter, wenn man nicht unbedingt Variablen vom Type String vergleichen will.
Schauen wir uns noch mal das ARRAY mit den Vorgabe Werten im Detail an:

```
int     gewicht [25][10] =          // 1=LEFT, 2=RIGHT, 3=FWD, 4=BWD
        {
          {0,1,0,3,3,5,6,2,3},
          {0,0,0,2,3,6,5,3,2},
```

Das ARRAY ist vom Typ Integer, also RAM sparende Programmierung. Die Bewegungsrichtungen wurden durch Zahlen ersetzt, muss nicht unbedingt sein, da wir die Bewegungsrichtungen im „BugDefault.h“ File den gleichen numerischen Werten zugewiesen haben.

```
/*  ********** Globale Variablen ********** */
#define     NULL      0
#define     FWD       3
#define     BWD       4
#define     LEFT      1
#define     RIGHT     2
```

Nun der Abfragetrick

Sensor-Abfrage:

```
Links:          0
Mitte:          1
Rechts:         0
Move_dir:       3
Last_move:      3    entspricht Integerwert   1033


  // Datenbank Query vorbereiten Einzelabfragen in eine fünfstellige Zahl umwandeln
  db_query = sensor_links * 10000 + sensor_mitte * 1000 + sensor_rechts * 100 + move_dir * 10 + last_dir;
```

Gleiches vorgehen mit den Werten im ARRAY:

Der Vergleich erfolgt dann in dem eigentlichen DBUG Algorithmus, wo das ARRAY komplett nach dem „db_query“ durchsucht wird. Zu diesem Zweck werden die ersten fünf Stellen aus dem ARRAY ausgelesen und mit selber Logik in einen Interger-Wert umgewandet.

```
    temp_db = gewicht[i][0] * 10000 + gewicht[i][1] * 1000 + gewicht[i][2] * 100  + gewicht[i][3] * 10 + gewicht[i][4];
```

Somit wird aus (siehe erste Zeile im ARRAY) 0,1,0,3,3 der Integerwert 1033
Diesen kann man klasse mit den Sensorwert „db_query“ vergleichen.

Im Programm sieht das dann wie folgt aus:

```
for (i = 0; i < 25; i++)   // gewichtsmatrix auswerten
 {
   temp_db = gewicht[i][0] * 10000 + gewicht[i][1] * 1000 + gewicht[i][2] * 100  + gewicht[i][3] * 10 + gewicht[i][4];

       if ( db_query == temp_db )
       {
          X  = X + (gewicht[i][5] - 5);                                    // Die 5 wieder abziehen so das X / Y

          • • •
```

Dem aufmerksamen Leser ist schon vorher aufgefallen ☺, dass die X -und Y-Werte nicht negativ sind. Das ist Absicht, da bei der späteren Implementierung auf dem Microcontroller auf Integer verzichtet werden soll und die Speicher sparende Variable **uint8_t** genutzt werden soll. Die Variable **uint8_t** hat den Wertebereich 0 – 255, eine Intergervariable (**int**) hat im Gegenzug einen Wertebereich von -32768 bis 32767 und benötigt somit mehr Speicher.
Bei einem PC ist das relativ egal, aber nutzt man einen ATtiny oder ATmega8 kommt man schon mal schnell an das Ende der Kapazität.
Unschöne Sache, dabei kann man sich toll den Bootloader auf dem Microcontroller überschreiben ☹
Also, soll ein Wert der eigentlich zwischen „-1“ und „1“ liegt für einen Schritt vorwärts oder rückwärts nicht negativ werden, dann addiert man halt +5 dazu und zieht sie später wieder ab.

Ein paar Besonderheiten:

- Ich habe als Compiler mal den Borland C++ Compiler genutzt, da mir z.B. die kbhit () Funktion gut gefällt. Compiler Download siehe http://edn.embarcadero.com/article/20633
- Normaler Weise nehme ich den gcc auf meiner Linuxkiste
- Das Programm bitte in einer DOS-Box ausführen, von ausreichender Größe !!! Breite 120, Höhe 50
- Das Programm aufrufen und die ENTER – Taste solange drücken bis das Hindernis erreicht ist, ab dann über nimmt der BUG2 Algorithmus die Steuerung und Endet, wenn das Hindernis umgangen wurde. Von da an wieder mit „ENTER – Taste“ weiter, bis Ziel erreicht
- Zum Source-Code gehören die folgenden Dateien und können unter www.ps-robotics.de im Download Bereich heruntergeladen werden. Dort finden sich auch Source-Codes zum A-Star Algorithmus:

    - bug2Defaults.h (Default Werte und Gewichts-Matrix)
    - bug2Libaray.h (File mit dem BUG2 Algorithmus)
    - odometrie.cpp (Testprogramm Source-Code)
    - help_routines.h (Funktionen für die Darstellung auf dem Monitor)
    - Odometrie.exe (bereits compilierte Version)

Die nächsten Schritte:

- Implementierung des Algorithmus auf meinen Roboter (on going)
- Installation von IR-Baken für die Navigation in geschlossenen Räumen
- Schreiben von Teil II Navigation
- Bisher wurden die Antworten auf die Sensorwerte in der Gewichte-Matrix von Hand eingetragen. Wie wäre das, wenn das Programm diese Werte selbst ermittelt bzw. im Lerning Mode die Werte gezeigt bekommt. Wir sind dann schell bei „Neuronalen Netzen“ angekommen. Das ist aber noch Zukunftsmusik

Die von mir geschriebenen Source-Codes deklariere ich hiermit als Freeware und kann von jedem genutzt und nach belieben verändert werden. Über Feedback und Verbesserungs-vorschläge würde ich mich freuen.
Es ist bestimmt nicht die eleganteste Version von Software, wie man diesen Algorithmus programmieren kann, aber bitte bedenken, ich mach das nur als Hobby und wie gesagt, der jüngste bin ich auch nicht mehr ☺ ☺ ☺

Bei Fragen oder Anregungen: peter.schneider@ps-robotics.de

Mit freundlichen Grüßen
Peter Schneider
www.ps-robotics.de

Nützliche Links:
http://ls12-www.cs.tu-dortmund.de/raptor/06_Navigation_Bahnplanung.pdf

http://www12.cs.fau.de/edu/robertino/ausarbeitung.pdf

http://www.ke.tu-darmstadt.de/lehre/archiv/ss06/se-spiele/slides/PathFinding-Awerbuch.pdf

Disclaimer