Università degli studi di Brescia
Facoltà di Ingegneria Elettronica

ROBOTICA

Prof. Riccardo Cassinis

Relazione Elaborato Robot non Olonomo

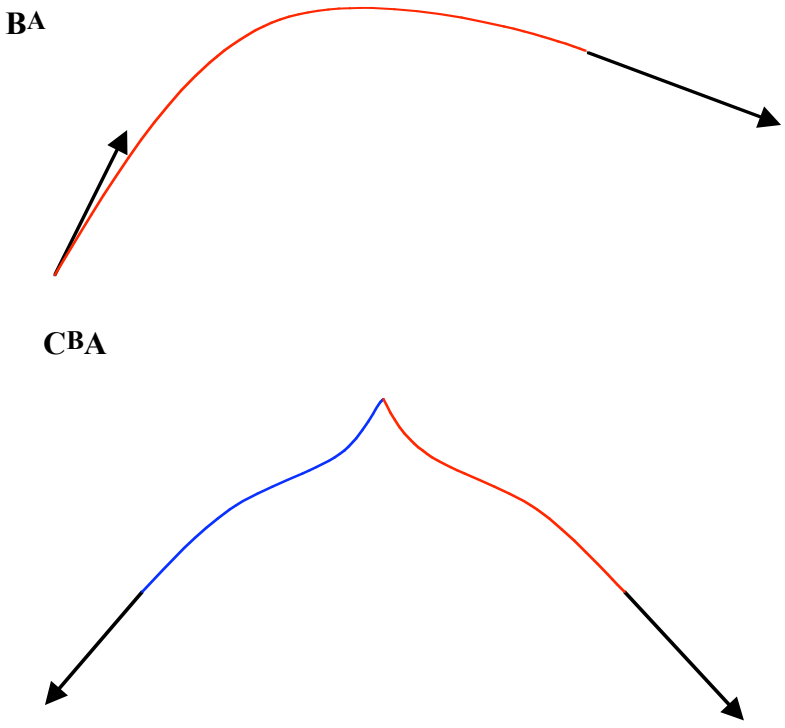
Studenti:

Migliorati Fabio (030126)

Morelli Nicola (030139)

Obiettivo:

Dato un algoritmo che estrapola i punti di una traiettoria di Bezier, realizzare un programma in colbert che la faccia seguire ad un robot non olonomo. Il problema rappresenta un'esigenza reale: pilotare un robot con limitato raggio di curvatura che per muoversi necessita di un software che tenga ben presente questo limite. Nella nostra realizzazione si è spesso pensato come esempio chiarificatore ad una macchina, per andare da un punto A a un punto B ci sono due possibilità: la prima è una traiettoria che parte tangente al vettore iniziale (individuato da A più il suo orientamento nel punto) e arriva nel punto B tangente al vettore finale (individuato da B più il suo orientamento nel punto). La seconda si presenta se la curva è tale da superare il raggio minimo di curvatura del robot, la soluzione è dividere il problema in due semicurve: una a marcia indietro da A a un punto C intermedio opportunamente calcolato e poi da C a B in avanti.



Il Makefile

```
# June 1999
#
# Makefile for Saphira applications
#

SHELL = /bin/sh

#################################################################

SRCD = ./
OBJD = ./
INCD = $(SAPHIRA)/handler/include/
LIBD = $(SAPHIRA)/handler/obj/
BIND = $(SAPHIRA)/bin/
COLBERT = $(SAPHIRA)/colbert/

# find out which OS we have
include $(SAPHIRA)/handler/include/os.h

CFLAGS =  -g -D$(CONFIG) $(PICFLAG)
CC = gcc
INCLUDE = -I$(INCD) -I$(X11D)include


#################################################################
all: $(BIND)mysaph $(BIND)async $(BIND)packet $(BIND)direct $(BIND)nowin
$(COLBERT)robotica.so


clean:
        rm *.o *~

#
# basic Saphira client
#
$(OBJD)mysaph.o: $(SRCD)mysaph.c
        $(CC) $(CFLAGS) -c $(SRCD)mysaph.c $(INCLUDE) -o $(OBJD)mysaph.o

$(BIND)mysaph: $(OBJD)mysaph.o
        $(CC) $(OBJD)mysaph.o -o $(BIND)mysaph -L$(LIBD) -lsf -L$(MOTIFD)lib
$(LLIBS) -lc -lm

#
# Packet communication client
#
$(OBJD)packet.o: $(SRCD)packet.c $(INCD)saphira.h
        $(CC) $(CFLAGS) -c $(SRCD)packet.c $(INCLUDE) -o $(OBJD)packet.o

$(BIND)packet: $(OBJD)packet.o
```

```
	$(CC) $(OBJD)packet.o -o $(BIND)packet \
-L$(LIBD) -lsf -L$(MOTIFD)lib $(LLIBS) -lc -lm

#
# Async function client
#
$(OBJD)async.o: $(SRCD)async.c $(INCD)saphira.h
	$(CC) $(CFLAGS) -c $(SRCD)async.c $(INCLUDE) -o $(OBJD)async.o

$(BIND)async: $(OBJD)async.o
	$(CC) $(OBJD)async.o -o $(BIND)async -L$(MOTIFD)lib -L$(LIBD) -lsf $(LLIBS) -lc
-lm

#
# Non-window client
#
$(OBJD)nowin.o: $(SRCD)nowin.c $(INCD)saphira.h
	$(CC) $(CFLAGS) -c $(SRCD)nowin.c $(INCLUDE) -o $(OBJD)nowin.o

$(BIND)nowin: $(OBJD)nowin.o
	$(CC) $(OBJD)nowin.o -o $(BIND)nowin -L$(OBJD) -L$(LIBD) -lsfx $(LLIBSX) -lc -
lm -lpthread




#
# Direct action client
#
$(OBJD)direct.o: $(SRCD)direct.c $(INCD)saphira.h
	$(CC) $(CFLAGS) -c $(SRCD)direct.c $(INCLUDE) -o $(OBJD)direct.o

$(BIND)direct: $(OBJD)direct.o
	$(CC) $(OBJD)direct.o -o $(BIND)direct \
-L$(LIBD) -lsf -L$(MOTIFD)lib $(LLIBS) -lc -lm

#
# Sample loadable object file
#
$(OBJD)robotica.o: $(SRCD)robotica.c $(INCD)saphira.h
	$(CC) $(CFLAGS) -c $(SRCD)robotica.c $(INCLUDE) -o $(OBJD)robotica.o

$(COLBERT)robotica.so: $(OBJD)robotica.o
	$(LD) $(SHARED) $(OBJD)robotica.o -o $(COLBERT)robotica.so
```

Questo file serve per compilare e creare la libreria robotica.so e tutte le sue dipendenze. Una volta creata la propria libreria si sostituisce il nome nel blocco chiamato “sample loadable object file” e si lancia usando il comando make.

Libreria ROBOTICA.SO

```
#include "stdio.h"
#include "math.h"
#include "tgmath.h"
#include "saphira.h"

typedef struct{
  float x;
  float y;
}TFpoint;

struct point2{
  int x;
  int y;
}m,m1;

typedef TFpoint T4punti[4];
typedef TFpoint Tpunti[204];

int ind_TFpoint;
Tpunti Bezier;
float  raggio;
float tinc;
/* calcola il coseno a partire dal seno dell'angolo voluto */
float coseno(float);
/* calcola il coseno di un angolo a partire dall'angolo in radianti */
float cose(float);
float sen(float);
/* calcola l'angolo tra i vettori individuati da x0-x1 e y0-y1 */
float arctan(float,float,float,float);
/* vengono passati i punti che individuano la traiettoria e il raggio minimo di curvatura, restituisce il raggio minimo trovato e modifica i  vettori (4 punti=2 vettori)  per riuscire ad ottenere la curva di Bezier */
float CercaCurvaBezier(T4punti,float);
/* calcola la curva di Bezier e mette 1/Tinc punti nella variabile Bezier */
void CalcolaCurvaBezier(T4punti,Tpunti);
/* restituisce il punto di intersezione tra i due vettori P0-P3 e P1-P2 */
TFpoint Intersezione(T4punti);
/* Spezza in due curve di Bezier la traiettoria trovando un punto intermedio sulla bisettrice dei vettori: modifica al sua interno l'array Bezier inserendo le coordinate dei punti */
void ElaboraParcheggio(T4punti);
/*calcola la distanza tra due punti*/
float dist(TFpoint,TFpoint);
/* controlla il punto di intersezione dei vettori: se cade all'interno del parallelogramma individuato dai 4 punti => traiettoria di parcheggio (restituisce "1"); altrimenti traiettoria unica (restituisce "0") */
int convergenti(T4punti);
/* Dati quattro punti e il raggio minimo di curvatura sceglie il tipo di traiettoria migliore e la calcola */
int calcola(int,int,int,int,int,int,int);
```

```
/* Setta le variabili punto e punto1 rispettivamente all' i_esino e all' i-1_esimo punto della
traiettoria: usata nel programma in colbert pid.act */
void givepoint (int);
EXPORT void sfLoadInit (void);

/* calcola il coseno a partire dal seno dell'angolo voluto */
float coseno(float a)
{
  return cos(asin(a));
}

/* calcola il coseno a partire dall'angolo in radianti */
float cose(float a)
{
  return cos(a*M_PI/180);
}

/* calcola il seno a partire dall'angolo in radianti */
float sen(float a)
{
  return sin(a*M_PI/180);
}

float arctan(float x0,float y0,float x1,float y1)
{
  float beta;
  if(x0==x1)
    if(y0>y1)
      beta=M_PI/2;
    else
      beta=-M_PI/2;
  else
    if(x0>x1)
      beta=atan((y0-y1)/(x0-x1));
    else
      beta=atan((y0-y1)/(x0-x1))+M_PI;
  return beta;
}

float CercaCurvaBezier(T4punti QuattroPuntiRobot,float rdcdato)
{
/* In input i quattro punti caratteristici della bezier in ordine di percorso */
  float k;
  float rdc,minrdc;
  float alpha,beta;
  float d03,x11,y11,x22,y22,x33,y33;
  float t;
  float num,n2,den;
  TFpoint Pinter;
  float rapporto;
  float ca,sa,cb,sb;
```

```
  float d1,d2;
  float maxk=0;

  minrdc=0;
/* Passiamo le coordinate dei punti, il punto di partenza è tacitamente (0,0) perché usiamo come riferimento la posizione del robot */
  x11=QuattroPuntiRobot[1].x;
  x22=QuattroPuntiRobot[2].x;
  x33=QuattroPuntiRobot[3].x;
  y11=QuattroPuntiRobot[1].y;
  y22=QuattroPuntiRobot[2].y;
  y33=QuattroPuntiRobot[3].y;

/* Distanza punto arrivo e punto partenza */
  d03=(float) sqrt((x33*x33)+(y33*y33));
/* Inizializzazione parametro "k" per modificare la lunghezza dei vettori partenza e arrivo: si potrebbe inizializzare a zero tuttavia per ridurre il tempo di calcolo e da prove sperimentali è risultato che raggio/2 è un buon valore di partenza */
  k=raggio/2.;

  alpha=arctan(x11,y11,0,0);
  beta=arctan(x22,y22,x33,y33);
  ca=(float) cos(alpha);
  sa=(float) sin(alpha);
  cb=(float) cos(beta);
  sb=(float) sin(beta);



  if(maxk==0) maxk=20.*raggio;
/* Modifichiamo P1 e P2 fino ad ottenere il raggio di curvatura desiderato (rdcdato) */
  while(minrdc<rdcdato)
    {
/* Incrementiamo k in modo diverso a seconda della differenza tra rdcdato e minrdc */
      rapporto=(float) fabs(minrdc-rdcdato)/rdcdato;
      if(rapporto>0.5)
          k=k*1.1;
      else
          {
            if(rapporto>0.05)
              k=k*1.01;
            else
              k=k+1.;
          }
/*cambiamo la posizione del punto P1 per variare la lunghezza del vettore partenza*/
      x11=k*ca;
      y11=k*sa;
/* Cambiamo la posizione del punto P2 per variare la lunghezza del vettore arrivo */
      x22=x33+k*cb;
      y22=y33+k*sb;
```

```
      t=0;
      minrdc=1000;
/* per ogni punto della traiettoria calcola il raggio di curvatura istantaneo, alla fine il minore
sarà la variabile minrdc*/
      while( ((t<=1) && (minrdc>rdcdato)) && (k<=maxk) )
          {
            n2=fabs(pow(t,4)*( 9*(x11*x11) + 6.*x11*(x33-3.*x22) + 9.*(x22*x22) - 6.*x22*x33
+ (x33*x33) + pow(3.*y11-3.*y22+y33,2.) ) – 4.*pow(t,3.)*( 6.*(x11*x11) + x11*(2.*x33-
9.*x22) + 3.*(x22*x22) - x22*x33 + (2.*y11-y22)*(3.*y11-3.*y22+y33) ) + 2.*(t*t)*(
11.*(x11*x11) + x11*(x33-11.*x22)+2.*(x22*x22)+11.*(y11*y11)+y11*(y33-11.*y22) +
2.*(y22*y22) ) - 4.*t*( 2.*(x11*x11) - x11*x22 + y11*(2.*y11 - y22)) + (x11*x11) + (y11*y11)
);

            num=(float) (3.*pow(n2,3./2.));
            den=(float) (2.*((t*t)*(x11*(3.*y22-2*y33)+x22*(y33-3*y11)+x33*(2.*y11-
y22))+t*(y11*(3.*x22-x33)-x11*(3*y22-y33))+x11*y22-x22*y11));

            rdc=(float) fabs(num/den);

            if(rdc<minrdc)
              minrdc=rdc;

            t=t+tinc;
          }
    }
/* se incrementando k supero il valore massimo impostato a 20*raggio esco senza trovare
la curva restituendo -1 altrimenti aggiorno di volta in volta le coordinate dei punti */
  if(k<=maxk)
    {
      QuattroPuntiRobot[1].x=x11;
      QuattroPuntiRobot[2].x=x22;
      QuattroPuntiRobot[3].x=x33;
      QuattroPuntiRobot[1].y=y11;
      QuattroPuntiRobot[2].y=y22;
      QuattroPuntiRobot[3].y=y33;

      return(minrdc);
    }
  else
    {
      return(-1);
    }
}

void CalcolaCurvaBezier(T4punti QuattroPuntiRobot,Tpunti Punti)
{
  float f;
  TFpoint punti[200];

  f=0;
```

```
/* f è il parametro con cui è parametrizzata la curva cubica di Bezier; lo 0.001 è presente
perché in certi casi 1=1 non viene rilevato a causa della precisione finita del calcolatore */
  while(f<=(1.+0.001))
    {
      Punti[(int) rint(f/tinc)].x=(float) (pow((1.-f),3)*QuattroPuntiRobot[0].x+3*f*pow(1.-
f,2)*QuattroPuntiRobot[1].x+3*(f*f)*(1.-
f)*QuattroPuntiRobot[2].x+pow(f,3)*QuattroPuntiRobot[3].x);
      Punti[(int) rint(f/tinc)].y=(float) (pow((1.-f),3)*QuattroPuntiRobot[0].y+3*f*pow(1.-
f,2)*QuattroPuntiRobot[1].y+3*(f*f)*(1.-
f)*QuattroPuntiRobot[2].y+pow(f,3)*QuattroPuntiRobot[3].y);
      f=f+tinc;
    }
}

TFpoint Intersezione(T4punti QuattroPuntiRobot)
{
  TFpoint unpunto;
  float den;

  den = (QuattroPuntiRobot[1].x*(QuattroPuntiRobot[2].y-QuattroPuntiRobot[3].y)-
QuattroPuntiRobot[1].y*(QuattroPuntiRobot[2].x-QuattroPuntiRobot[3].x));

  if (den==0)
    {
      unpunto.x=(QuattroPuntiRobot[1].x + QuattroPuntiRobot[3].x)/2.;
      unpunto.y=(QuattroPuntiRobot[1].y + QuattroPuntiRobot[2].y)/2.;
    }
  else
    {
      unpunto.x=QuattroPuntiRobot[1].x*(QuattroPuntiRobot[3].x*QuattroPuntiRobot[2].y-
QuattroPuntiRobot[2].x*QuattroPuntiRobot[3].y)/den;
      unpunto.y=QuattroPuntiRobot[1].y*(QuattroPuntiRobot[3].x*QuattroPuntiRobot[2].y-
QuattroPuntiRobot[2].x*QuattroPuntiRobot[3].y)/den;
    }
  return(unpunto);
}

/* Questa è la convenzione con cui passiamo i punti al programma:
   P0 = punto partenza
   P1 = punto orientamento di partenza
   P2 = punto di arrivo
   P3 =  punto orientamento di arrivo
   l'algoritmo in realta vuole P3 e P2 invertiti, operazione svolta */
void ElaboraParcheggio(T4punti PuntiR)
{
  T4punti NuoviPunti,NuoviPuntiB;
  TFpoint UnFPunto,Pmedio;
  float alpha,beta,gamma;
  float h;
  float rdc1,rdc2;
  Tpunti Bezier1,Bezier2;
```

```
  float f;
  float cg,sg;

/* invertiamo i vettori dati dai punti perché l'algoritmo considera il primo tratto del
parcheggio con moto in avanti mentre invece noi vogliamo per prima la retromarcia */
  PuntiR[1].x=2*PuntiR[0].x-PuntiR[1].x;
  PuntiR[1].y=2*PuntiR[0].y-PuntiR[1].y;

/* invertiamo il P3 con P2 in modo che in P2 ci sia il punto che definisce l'orientamento
dell'arrivo, mentre in P3 ci sono le coordinate dell'arrivo: questa operazione è fatta perché
intuitivamente preferiamo avere come ultimo punto P3 proprio la punta del vettore di arrivo
*/
  UnFPunto=PuntiR[3];
  PuntiR[3]=PuntiR[2];
  PuntiR[2]=UnFPunto;
  PuntiR[2].x=2*PuntiR[3].x-PuntiR[2].x;
  PuntiR[2].y=2*PuntiR[3].y-PuntiR[2].y;

/* Trova il punto medio tra il punto di partenza e il punto di arrivo */
  Pmedio.x=(PuntiR[0].x + PuntiR[3].x)/2;
  Pmedio.y=(PuntiR[0].y + PuntiR[3].y)/2;

/* NuoviPunti contengono le coordinate dei 4 punti che definiscono il primo tratto della
traiettoria di parcheggio */
  NuoviPunti[0]=PuntiR[0];
  NuoviPunti[1]=PuntiR[1];
  NuoviPunti[2]=Pmedio;

/* Inizializziamo il parametro "h" utilizzato per variare la posizione del punto intermedio  tra
le due semicurve lungo la bisettrice */
  h=raggio/2.;

  rdc1=-1.;
  rdc2=-1.;

  alpha=arctan(PuntiR[1].x,PuntiR[1].y,0.,0.);
  beta=arctan(PuntiR[2].x,PuntiR[2].y,PuntiR[3].x,PuntiR[3].y);

  gamma=(alpha + beta)/2.;

  if (fabs(beta - alpha)>M_PI)  gamma=gamma+M_PI;

  cg=cos(gamma);
  sg=sin(gamma);

/* Modifichiamo lungo la bisettrice la posizione del punto intermedio */
  NuoviPunti[3].x=Pmedio.x+(h*cg);
  NuoviPunti[3].y=Pmedio.y+(h*sg);
/* NuoviPuntiB contengono le coordinate dei punti che individuano l'ultima semicurva con
riferimento il punto intermedio che è il punto di partenza della semicurva  */
  NuoviPuntiB[0]=NuoviPunti[0];
```

```
/* Modifichiamo il punto intermedio sulla bisettrice finché CercaCurvaBezier non trova
entrambe le semicurve desiderate */
  while ( ((rdc1<0) || (rdc2<0)) && (h<(10.*raggio)) )
    {
/* modifico lungo la bisettrice la posizione del punto intermedio */
      NuoviPunti[3].x=(float) Pmedio.x+(h*cg);
      NuoviPunti[3].y=(float) Pmedio.y+(h*sg);


      rdc1 = CercaCurvaBezier (NuoviPunti,raggio);

      if (rdc1>0)
          {

            NuoviPuntiB[1].x=NuoviPunti[2].x-NuoviPunti[3].x;
            NuoviPuntiB[1].y=NuoviPunti[2].y-NuoviPunti[3].y;
            NuoviPuntiB[2].x=PuntiR[2].x-NuoviPunti[3].x;
            NuoviPuntiB[2].y=PuntiR[2].y-NuoviPunti[3].y;
            NuoviPuntiB[3].x=PuntiR[3].x-NuoviPunti[3].x;
            NuoviPuntiB[3].y=PuntiR[3].y-NuoviPunti[3].y;
            rdc2=CercaCurvaBezier(NuoviPuntiB,raggio);
            if (rdc2>0)
              {
/* Calcoliamo i punti delle due semicurve e li mettiamo nell'array Bezier */
                CalcolaCurvaBezier(NuoviPuntiB,Bezier2);
                CalcolaCurvaBezier(NuoviPunti,Bezier1);
                f=0;
                while (f<=(1.+0.001))
                    {
                      Bezier[(int) rint(f/tinc)]=Bezier1[(int) rint(f/tinc)];
                      Bezier[(int) (rint(1./tinc)+rint(f/tinc))].x = Bezier1[(int) rint(1./tinc)].x +
Bezier2[(int) rint(f/tinc)].x;
                      Bezier[(int) (rint(1./tinc)+rint(f/tinc))].y = Bezier1[(int) rint(1./tinc)].y +
Bezier2[(int) rint(f/tinc)].y;
                      f = f+tinc;
                    }
              }
          }
      h = h * 1.1;
    }
}

float dist(TFpoint a,TFpoint b)
{
  return sqrt(pow(a.x-b.x,2)+pow(a.y-b.y,2));
}

int convergenti(T4punti punti)
{
  TFpoint pinter;
  pinter=Intersezione(punti);
```

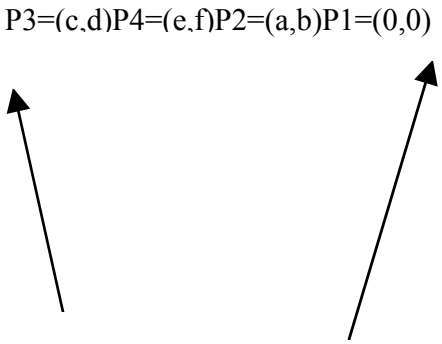
```
  if( ((dist(punti[0],pinter)<dist(punti[1],pinter)) && (dist(punti[2],pinter)<dist(punti[3],pinter)))
|| ((dist(punti[0],pinter)>dist(punti[1],pinter)) && (dist(punti[2],pinter)>dist(punti[3],pinter))) )
    return 1;
  else
    return 0;
}

int calcola(int a, int b,int c, int d,int e, int f,int rdc)
{
  T4punti punti;
  FILE *out,*out1;
  int i;
  int nc; /*nc=0 1 curva nc=1 2 curve*/
  float t;
  char ch;

  raggio=rdc;

  out1=fopen("./temp.dat","w");
  out=fopen("./robot.dat","w");

  punti[0].x=0;
  punti[0].y=0;
  punti[1].x=a;
  punti[1].y=b;
  punti[2].x=c;
  punti[2].y=d;
  punti[3].x=e;
  punti[3].y=f;


  nc=convergenti(punti);
  if(nc)
    ElaboraParcheggio(punti);
  else
    {
/* Cambiamo  i punti passati per adattarli all'algoritmo di Bezier */
      punti[0].x=0;
      punti[0].y=0;
      punti[1].x=a;
      punti[1].y=b;
      punti[2].x=2*c-e;
      punti[2].y=2*d-f;
      punti[3].x=c;
      punti[3].y=d;
      CercaCurvaBezier(punti,raggio);
      CalcolaCurvaBezier(punti,Bezier);
    }
/*scriviamo in robot.dat i punti della traiettoria (ci sono le virgole al posto dei punti nei
numeri reali: utile per excel) */
  fprintf(out1,"tinc=%f raggio=%f\n",tinc,raggio);
```

```
  if(nc)
  for(i=0;i<=200;i++)
    {
      fprintf(out1,"%d\t%f\t%f\n",i+1,Bezier[i].x, Bezier[i].y);
    }
  else
  for(i=0;i<=100;i++)
    {
      fprintf(out1,"%d\t%f\t%f\n",i+1,Bezier[i].x, Bezier[i].y);
    }
  fclose(out1);

  out1=fopen("./temp.dat","r");
  while((ch=fgetc(out1))!=EOF)
    {
      if(ch=='.')
          fprintf(out,"%c",',');
      else
          fprintf(out,"%c",ch);
    }
  fclose(out);
  fclose(out1);
  /* restituisce "1" se curva unica, "2" se curva di parcheggio */
  if(nc) return 2;
  if(!nc) return 1;
}

void givepoint (int i)
{
  m.x=10*rint(Bezier[i].x);
  m.y=10*rint(Bezier[i].y);
  m1.x=10*rint(Bezier[(i-1)].x);
  m1.y=10*rint(Bezier[(i-1)].y);
}
/* Funzione che rende visibili in Colbert le nostre funzioni e strutture dati */
EXPORT void sfLoadInit (void)
{
  sfSMessage("HELLO WORLD!!");
  sfAddEvalFn("calcola",calcola,sfINT,7,sfINT,sfINT,sfINT,sfINT,sfINT,sfINT,sfINT);
  sfAddEvalFn("coseno",coseno,sfFLOAT,1,sfFLOAT);
  sfAddEvalFn("cose",cose,sfFLOAT,1,sfFLOAT);
  sfAddEvalFn("sen",sen,sfFLOAT,1,sfFLOAT);
  ind_TFpoint=sfAddEvalStruct("TFpoint",sizeof(struct point2),(char *)&m,2,
                              "x",&m.x,sfINT,
                              "y",&m.y,sfINT);
  sfAddEvalVar("punto",ind_TFpoint,(fvalue *)&m);
  sfAddEvalVar("punto1",ind_TFpoint,(fvalue *)&m1);
  sfAddEvalVar("tinc",sfFLOAT,(fvalue *)&tinc);
  sfAddEvalFn("givepoint",givepoint,sfVOID,1,sfINT);
}
```

Questa libreria contiene le funzioni fondamentali per il calcolo dei punti della traiettoria:

- **Int calcola (int a, int b, int c, int d, int e, int f, int rdc)**

  I parametri passati sono le coordinate dei punti che definiscono la traiettoria scelta, sono nel sistema di riferimento locale del robot e quindi il punto di partenza essendo sempre (0,0) è stato omesso. Il parametro rdc è il raggio minimo di curvatura scelto per il calcolo.



  Questa funzione restituisce un intero che se uguale a 1 significa che il robot esegue una curva unica, se restituisce 2 invece esegue una traiettoria di parcheggio divisa in due parti: la prima eseguita in retromarcia e la seconda in avanti.Questa funzione ne chiama altre al suo interno che esplicano l'individuazione della tipologia di curva migliore (singola o con parcheggio) e calcolano sfruttando le curve cubiche di Bezier e il raggio minimo di curvatura i punti per cui il nostro robot dovrà passare: i punti sono contenuti nell'array Bezier[200];

- **void givepoint (int i)**

  Dato l'indice i immagazzina nella variabile "punto" le coordinate del punto Bezier[i] e nella variabile "punto1" le coordinate del punto Bezier[i-1]. Questa funzione è chiamata ciclicamente da PID.act per ogni tratto di retta perché restituisce le coordinate del punto di partenza con la variabile "punto1" e il punto d'arrivo con "punto". Questo metodo di passare con due variabili esportate alla volta i punti si è reso necessario in quanto in colbert non sono presente gli array.

Oltre a queste funzioni è stata definita una nuova struttura dati per rappresentare dei punti bidimensionali con tipi float, usata per compatibilità verso le funzioni di Bezier.
Le variabili "punto1" e "punto" sono visibili da colbert e vengono usate per passare alla struttura point presente le coordinate arrotondate all'intero.
Questa libreria esiste perché colbert essendo un linguaggio C-like interpretato non contiene tutte le funzioni e strutture supportate dal C, inoltre la computazione risulta più lenta di un programma C compilato: per ovviare a questi limiti è possibile realizzare librerie .SO (Shared Object) ad hoc che contengono a seconda delle esigenze parti di programma che direttamente colbert non può gestire.
Per permetterne la visione a colbert è necessario usare la funzione sfLoadInit che tramite delle specifiche funzioni di saphira sfAddEvalxxx permette di esportare variabili, strutture e funzioni.

PID.act

```
load robotica.so;

act bezier(int p2x,int p2y,int p3x,int p3y,int p4x,int p4y,int rdc,float t,int VEL)
{
  int i;
  int k;
  float a;
  float b;
  float c;
  float d;
  float z;
  point *origine;
  point *temp1;
  point *temp2;
  float in;    /* integrale errore */
  float dold;  /* errore ciclo precedente */
  float yold;  /*posizione angolare ciclo precedente */
  float y;     /* uscita equivalente alla sterzata del robot */
  int dir;     /*dir=1 avanti dir=-1 indietro*/
  int vrot;    /*velocita angolare di controllo*/
  int nc;      /*nc=1 1 curva; nc=2 traiettoria di parcheggio*/
  float Ti;    /*costante integrativa PID*/
  float Td;    /*costante derivativa PID*/
  float Kp;    /*costante proporzionale PID*/
  float cos;
  int ox;      /*origine x*/
  int oy;      /*origine y*/
  int vel;
  float velmax;

  vel=100;
  tinc=t;
  Kp=1;
  Ti=2;
  Td=0.05;
  nc=calcola(p2x,p2y,p3x,p3y,p4x,p4y,rdc);

  ox=sfRobot.ax;
  oy=sfRobot.ay;
  if(nc==1) sfSMessage("1 curva");
  if(nc==2) sfSMessage("2 curve");

  givepoint(1);
  temp1=sfCreateGlobalPoint(p2x+ox,p2y+oy,0); /*orientiamo il robot nella direzione p1-
p2*/
  turn(sfPointPhi(temp1));

  i=1;
  yold=0; /*inizializziamo le variabili utilizzate nel controllo PID*/
```

```
dold=0;
if(nc==1)
  dir=1;
else
  dir=-1;
while(i<=(nc*1/tinc))
  {
    if (k==5)
        {
          in=in/2;  /* integro su k tratti di rette l'errore e poi do un peso al passato*/
          k=0;
        }

    if(i==((1/tinc)+1))
        {
          dir=1;
          sfSetRVelocity(0);
        }

    if(i<=3)
        {
          velmax=(VEL/3)*i;
          sfSetMaxVelocity(velmax);
        }
    if( (i>(1/tinc-3)) && (i<=(1/tinc)) )
        {
          velmax=(VEL-100)*(1/tinc-i)/3+100;
          sfSetMaxVelocity(velmax);
        }
    if( (i>(1/tinc)) && (i<=(1/tinc+3)) )
        {
          velmax=(VEL/3)*(i-1/tinc);
          sfSetMaxVelocity(velmax);
        }
    if( (i>(2/tinc-3)) && (i<=(2/tinc)) )
        {
          velmax=(VEL-100)*(2/tinc-i)/3+100;
          sfSetMaxVelocity(velmax);
        }


    givepoint(i);
    temp1=sfCreateGlobalPoint(punto1.x+ox,punto1.y+oy,0); /* punto partenza */
    temp2=sfCreateGlobalPoint(punto.x+ox,punto.y+oy,0);   /* punto obiettivo */
    sfAddPoint(temp1);
    sfAddPoint(temp2);
    a=sfPointDistPoint(temp1,temp2);
    sfSetVelocity(dir*vel);
    c=a;
    b=0;
    cos=1;
```

```
      while( b<=(a/cos) || (c>a) )
          {

            c=sfPointDist(temp2);
            b=sfPointDist(temp1);

          z=-(punto.y-punto1.y)*(sfRobot.ax-punto1.x-ox)+(punto.x-punto1.x)*(sfRobot.ay-
punto1.y-oy);
          d=-z/a;
            if ( (d>60) || (d<-60))
              vel=vel-20;
            else
            if ( (d>30) || (d<-30))
              vel=vel-10;
            else
            if (vel<VEL)
              vel=vel+10;
            if (vel<100)
              vel=100;
          in=in+d*0.1;    /* aggiorno l'integrale dell'errore */
          y=Kp*(d+(1/Ti)*in+Td*(d-dold)/0.1); /* azione PID */
          dold=d;   /* aggiorno errore del ciclo precedente */
          vrot=(y-yold)/0.1;
            if(vrot>60) vrot=60;  /*pongo un limite superiore e inferiore a vrot*/
            if(vrot<-60) vrot=-60;
            sfSMessage("a=%f c=%f dold=%f vel=%d ",a,c,dold,vel);
            sfSMessage("z=%f in=%f y=%f i=%d",z,in,y,i);
            sfSetVelocity(dir*vel);
            sfSetRVelocity(vrot);  /* vrot>0 robot gira sx altromenti vrot<0 robot gira dx */
            yold=y;
            if(b==0)
              cos=1;
            else
              cos=coseno(z/(a*b));
            if(cos<0) cos=-cos;
          }
/*        sfRemPoint(temp1); */
/*        sfRemPoint(temp2); */
      i=i+1;
      k=k+1;
    }
  sfSetRVelocity(0);  /* arresta il robot */
  sfSetVelocity(0);
  sfRobotCom2Bytes(sfCOMVEL2,0,0);
}

act bezier1(int p3x,int p3y,int p4x,int p4y,int rdc, float t,int VEL)
{
  int p2x;
  int p2y;
  p2y=sen(sfRobot.ath)*100;
```

```
  p2x=cose(sfRobot.ath)*100;
  sfSMessage("p2x=%d p2y=%d",p2x,p2y);
  start bezier(p2x,p2y,p3x,p3y,p4x,p4y,rdc,t,VEL);
}
```

PID.act contiene l'activity Bezier a cui passiamo come parametri i punti che definiscono la traiettoria, il raggio di curvatura minimo, il passo di campionamento "t" e la velocità massima "VEL". Il parametro "t" serve per aumentare il numero di punti che definiscono la traiettoria, il valore massimo è di 10msec cioè t=0.01 che corrisponde a 100 punti per semicurva. Come unità di misura utilizziamo la seguente convenzione:

- p2,p3,p4 : [cm] ;
- rdc : [cm] ;
- t : [sec] ;
- VEL :[mm/sec] ;

Quest'activity controlla il robot sfruttando un controllore PID. L'idea è di segmentare la traiettoria, in questo modo abbiamo il sottoproblema che il robot percorra una retta. Una seconda possibilità è usare l'activity Bezier1 la quale non richiede l'orientamento iniziale perché usa quello attuale del robot.
Per calcolare la distanza tra il robot e la retta ideale sfruttiamo delle relazioni trigonometriche sul triangolo formato dal punto di partenza ("punto1"), il punto di arrivo ("punto") e la posizione attuale del robot (sfRobot).

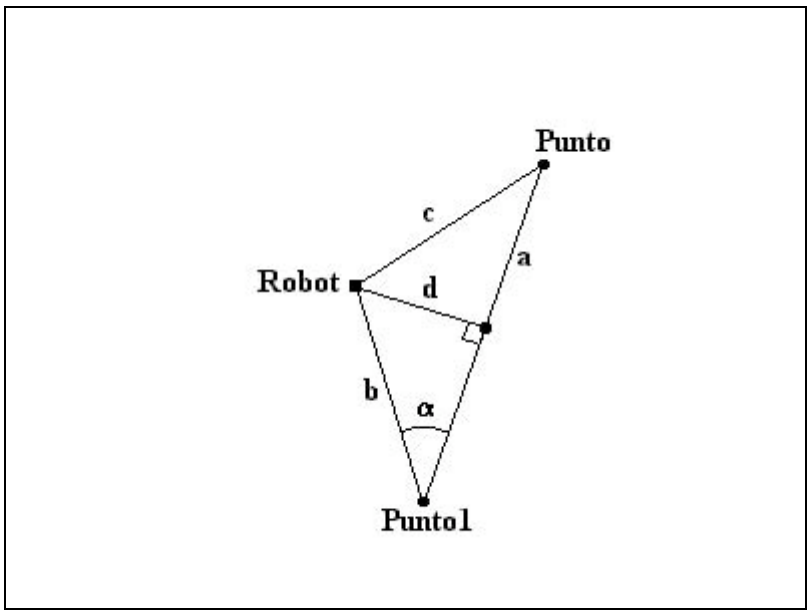


Figura 1

Per il calcolo di d sfruttiamo la seguente relazione:

$$d = b \cdot sen(\alpha)$$

Il seno di _ lo calcoliamo attraverso il prodotto vettoriale tra a e b:

$$\left|\bar{a}\times\bar{b}\right| = a\cdot b\cdot sen(\alpha) \Rightarrow sen(\alpha) = \frac{\left|\bar{a}\times\bar{b}\right|}{a\cdot b}$$

Per il calcolo del prodotto vettoriale sfruttiamo la relazione

$$z_{(i)} = \underline{a_{(i)}}\cdot b_{(i)}$$

prodotto tra due vettori "a" e "b" nello stesso riferimento i ma con "a" in forma di matrice emisimmetrica, mentre "b" nella forma a colonna classica. Per chiarezza di seguito presentiamo la forma matriciale:

$$\begin{bmatrix} z_x \\ z_y \\ z_z \end{bmatrix} = \begin{bmatrix} 0 & -a_z & a_y \\ a_z & 0 & -a_x \\ -a_y & a_x & 0 \end{bmatrix} \cdot \begin{bmatrix} b_x \\ b_y \\ b_z \end{bmatrix}$$

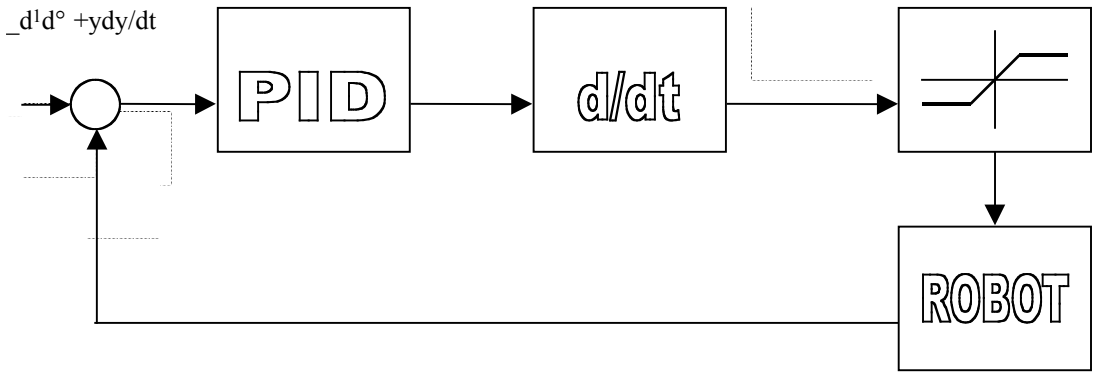
Essendo i due vettori nel piano x-y il prodotto vettoriale avrà componente solo sull'asse z, per questo nel programma abbiamo chiamato il prodotto vettoriale proprio z. Se il robot è a destra della retta formata tra punto partenza e punto di arrivo allora z sarà negativo, viceversa se sarà a sinistra z diverrà positivo. Questo segno di z è fondamentale per il controllo in quanto tramite la semplice relazione:

$$d^1 = \frac{z}{a}$$

ricaviamo di quanto è fuori asse il robot, e dal confronto tra il d^1 attuale del robot e il valore desiderato d° che chiaramente è zero otteniamo l'errore vero e proprio di traiettoria d:

$$d = d^0 - d^1$$

Quindi all'ingresso del mio controllore avrò la variabile errore, infatti per d>0, cioè robot fuori traiettoria a destra, l'uscita del controllo di posizione angolare y sarà anch'essa positiva e la funzione sfSetRVelocity(dy/dt) tenderà a far ruotare il robot a sinistra per correggere.
Lo schema completo del controllo adottato è il seguente:



Il PID controlla la posizione angolare del robot restituendo y in gradi, dopodiché calcoliamo la velocità angolare derivando y, essendo il nostro un sistema di controllo discretizzato a passi di 100ms diventa:

$$vrot = \frac{y - yold}{Tc}$$

dove Tc=0.1 [sec]. Il programma resta nel tratto attuale finchè:

$$b \le \frac{a}{\cos(\alpha)} \quad \text{o} \quad c > a$$

questa regola è stata introdotta perché il robot deve passare al tratto successivo solo quando ha superato la perpendicolare alla traiettoria ideale nel punto di arrivo nel tratto attuale: ciò assicura che il robot pur mancando l'intorno del punto destinazione cerchi di raggiungere il punto successivo.
Il controllo della velocità del robot è tale che per i primi tre tratti accelera progressivamente da fermo alla velocità massima, mentre negli ultimi tre tratti decelera fino a fermarsi. Per la traiettoria di parcheggio questo comportamento è seguito per ciascuna semicurva.
Per evitare fenomeni di saturazione della velocità e impulsi spuri nel passaggio al tratto successivo abbiamo aggiunto un blocco saturatore che limita la velocità angolare del robot. Il controllo effettuato dal PID garantisce ottime prestazioni statiche: l'errore d si mantiene praticamente sempre inferire al centimetro e solo durante curve molto accentuate sale fino a valori nell'intorno dei 3 centimetri. I test condotti sul simulatore ci hanno permesso velocità rettilinee di 300 mm/s.
Inoltre non si presentano praticamente oscillazioni nell'andamento del robot. Questi risultati sono stati ottenuti unendo le tabelle di taratura dei parametri PID di Ziegler e Nichols con prove al simulatore e facendo rallentare il robot ogni volta che l'errore d assume un valore eccessivo. La velocità cala fino a un minimo di 100 mm/sec, e viene aumentata quando il robot ha stabilizzato l'errore sotto la soglia fissata.

Ricordiamo brevemente che l'azione integrativa effettuata tramite Ti garantisce la precisione statica con $d \approx d^0$, mentre la costante derivativa Td agisce da "ammortizzatore" durante la sterzata evitando oscillazioni. L'equazione che rappresenta il controllore PID è la seguente:

$$y = Kp \cdot \left(d + \frac{1}{Ti} \cdot in + Td \cdot \frac{d - dold}{Tc} \right)$$

La scelta di utilizzare un controllore PID è dettata dalla non conoscenza della f.d.t. del robot. In queste situazioni solitamente si analizza la risposta all'impulso del sistema in anello aperto e si ricavano tramite le tabelle precedentemente menzionate i parametri da inserire nel controllore. Se il sistema in esame è approssimabile con un sistema del primo ordine con ritardo i PID sono la scelta più economica e semplice grazie alla vasta documentazione presente.
Nel continuo il blocco PID avrà una f.d.t.:

$$G_c(s) = K_p \cdot \left(1 + \frac{1}{T_i \cdot s} + T_d \cdot s \right)$$

Se si antitrasforma nel dominio del tempo l'azione di controllo

$$Y(s) = G_c(s) \cdot E(s)$$

esercitata dal controllore:

$$y(t) = K_p \cdot \left(e(t) + \frac{1}{T_i} \cdot \int_0^t e(\tau) \cdot d\tau + T_d \cdot \frac{de(t)}{dt} \right)$$

dove e(t) è l'errore e y(t) il segnale di controllo al processo.
Queste relazioni nel caso di controllo digitale vengono modificate nel seguente modo:

$$\begin{cases} i[n] = i[n-1] + e[n] \cdot T_c \\ y[n] = K_p \cdot \left\{ e[n] + \frac{1}{T_i} \cdot i[n] + T_d \cdot \frac{e[n] - e[n-1]}{T_c} \right\} \end{cases}$$

Dove i[n] è l'integrale al ciclo n-esimo dell'errore ed e[n-1] è l'errore al ciclo precedente, queste variabili nel nostro caso si chiamano "in" e "dold" dato che avevamo chiamato l'errore "d".
Il nostro PID integra dall'istante iniziale in avanti e quindi per evitare che il termine integrativo in caso di prolungata curvatura in un'unica direzione causi un eccessivo peso quando venga richiesta una repentina curvatura dalla parte opposta abbiamo imposto che dopo ogni quattro tratti di rette il peso del termine integrale "in" sia

ridotto della metà, la stessa riduzione avviene se il termine integrativo cresce pericolosamente oltre una certa soglia.

Conclusioni

Durante le prove le prestazioni migliori si sono avute con velocità intorno ai 150 mm/sec, anche se con velocità ben superiori si riscontrano solo delle piccole oscillazioni: queste ultime sono dovute principalmente al tempo di campionamento Tc di 100 msec con cui vengono aggiornate le informazioni del robot.