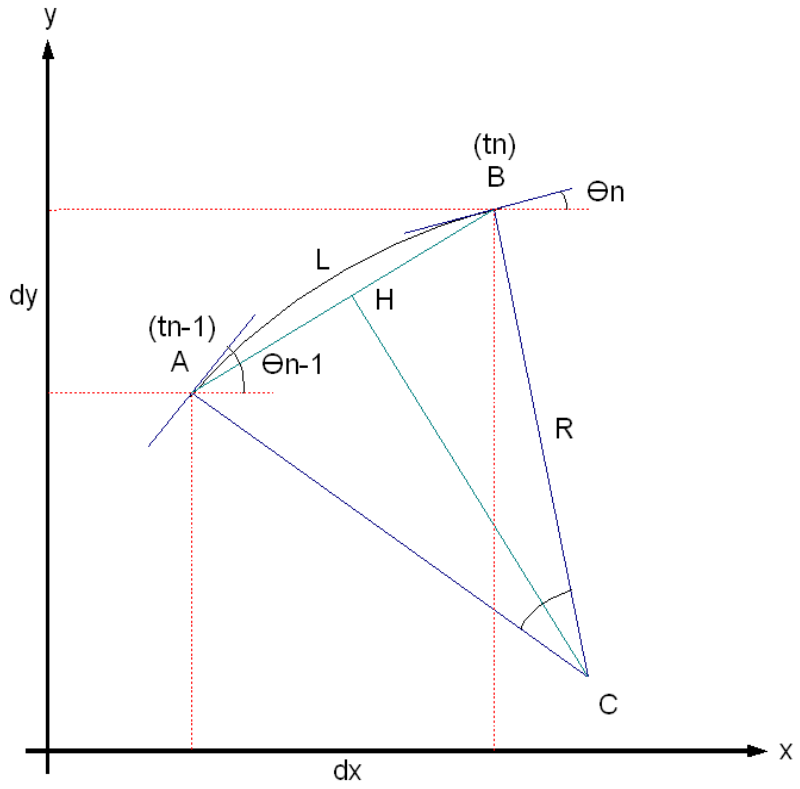
ODOMETRIE par l'équipe RCVA



Préalable :

Le robot est sensé utiliser 2 roues codeuses lui permettant de mesurer les déplacements gauche et droite du robot.
GAUCHE : variable comptant en nombre de pas le déplacement à gauche
DROITE : variable comptant en nombre de pas le déplacement à droite

A partir de ces 2 variables sont calculés le déplacement et l'orientation du robot.
D_ROBOT représente en nombre de pas le déplacement du centre de l'essieu.
θ_ROBOT représente en nombre de pas l'orientation de l'axe longitudinal du robot par rapport à l'axe des x.

D_ROBOT = ½(GAUCHE + DROITE)
θ_ROBOT= θ_ROBOT_INIT + (DROITE - GAUCHE)

On dit que les 2 variables **D_ROBOT, θ_ROBOT** exprimées en nombre de pas, appartiennent au système d'*unité Robot*.

Dans la partie théorique de ce document, on utilise les variables déplacement D et orientation θ, exprimées dans le *système SU* (mètre ou cm et radian), avec la correspondance :

D_ROBOT = COEF_D *** D** (D en mètre)
θ_ROBOT = COEF_ROT *** θ** (θ en radian)

COEF_D et COEF_ROT sont des constantes pré calculées puis ajustées expérimentalement.

On suppose le calcul échantillonné avec une période **Tech**.
A chaque instant **tn**, le robot reçoit les informations fournies par les codeurs incrémentaux et en déduit les valeurs de **Dn** et **θn**.

Dn : distance curviligne parcourue par le centre de l'essieu**.**
θn : orientation (axe longitudinal du robot par rapport à l'axe des x).

Sur la figure les points A et B représentent les positions du robot aux 2 instants consécutifs **tn-1** et **tn**, avec :

θn-1 : orientation à l'instant **tn-1**
θn : orientation à l'instant **tn**
L= Dn-Dn-1 : distance curviligne parcourue pendant une période **Tech**.
dx , **dy** : projections de l'arc **L** sur les 2 axes x et y.

Le robot ne recevant aucune information en dehors des instants d'échantillonnage, l'odomètrie se présente ainsi :

A chaque instant **tn**, le robot calcule **dx** et **dy** en fonction de **θn**, **θn-1, L**
Puis il procède à l'intégration numérique :

$$x_n = x_{n-1} + dx$$
$$y_n = y_{n-1} + dy$$

En théorie la mission est impossible, car le calcul des projections **dx** et **dy** de l'arc de trajectoire **L** dépend évidemment de sa forme. Hors les évolutions de la trajectoire entre A et B sont ignorées par le robot.

Afin de sortir de l'impasse, l'idée alors est de choisir une fréquence d'échantillonnage suffisamment rapide afin de permettre une approximation acceptable bien que simpliste à priori de la trajectoire entre les points A et B

<u>Approximation linéaire :</u>
Elle consiste à approximer l'arc de trajectoire entre A et B par la sécante AB (Soit l'approximation en distance **AB=L**).
On suppose également que l'orientation de la sécante est égale à la moyenne des orientations en A et B.
L'odomètrie se résume alors aux équations :
Orientation de la sécante AB :

$$\theta_{moy} = \tfrac{1}{2}(\theta_n + \theta_{n-1})$$

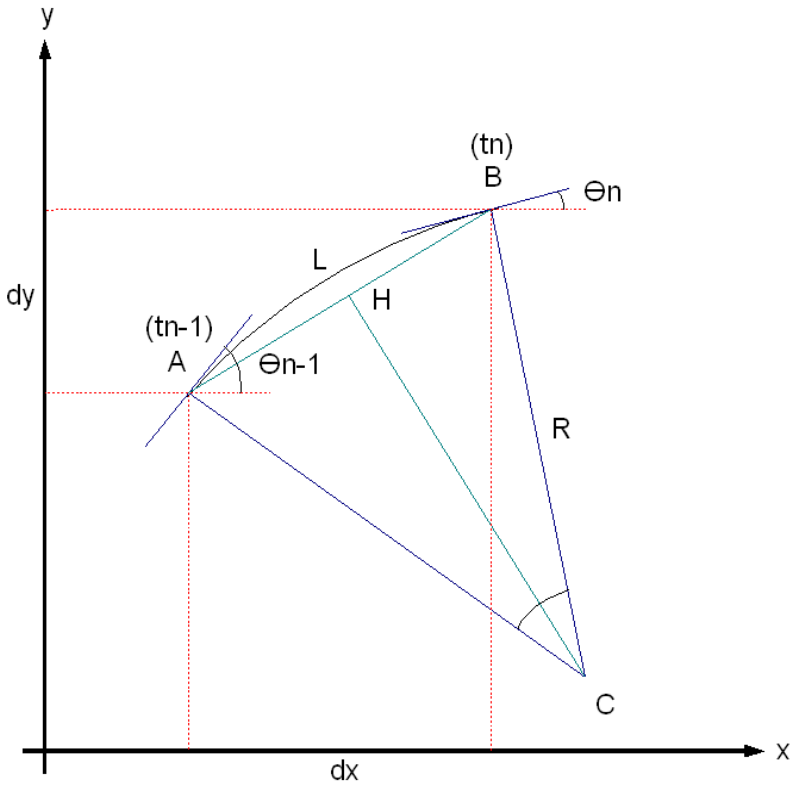
Projection de la sécante AB sur les 2 axes :

$$dx = L.\cos(\theta_{moy})$$
$$dy = L.\sin(\theta_{moy})$$

Loi de récurrence de l'intégration numérique :

$$x_n = x_{n-1} + dx$$
$$y_n = y_{n-1} + dy$$

Approximation circulaire. Version1 :



On approxime la trajectoire entre A et B par un arc de cercle.
L'angle au centre inscrit correspondant vaut en valeur absolue :

$\Delta\theta_n$=val_absolue($\theta_n - \theta_{n-1}$)

On en déduit le rayon du cercle (rayon de courbure de la trajectoire)

$R = L/\Delta\theta_n$

On considère la relation algébrique **: AB = CB – CA**

L'orientation de CB est égale à $\theta_n + \pi/2$.

L'orientation de CA est égale à $\theta_{n-1} + \pi/2$.

On projette sur les axes x et y la relation algébrique **AB = CB – CA**

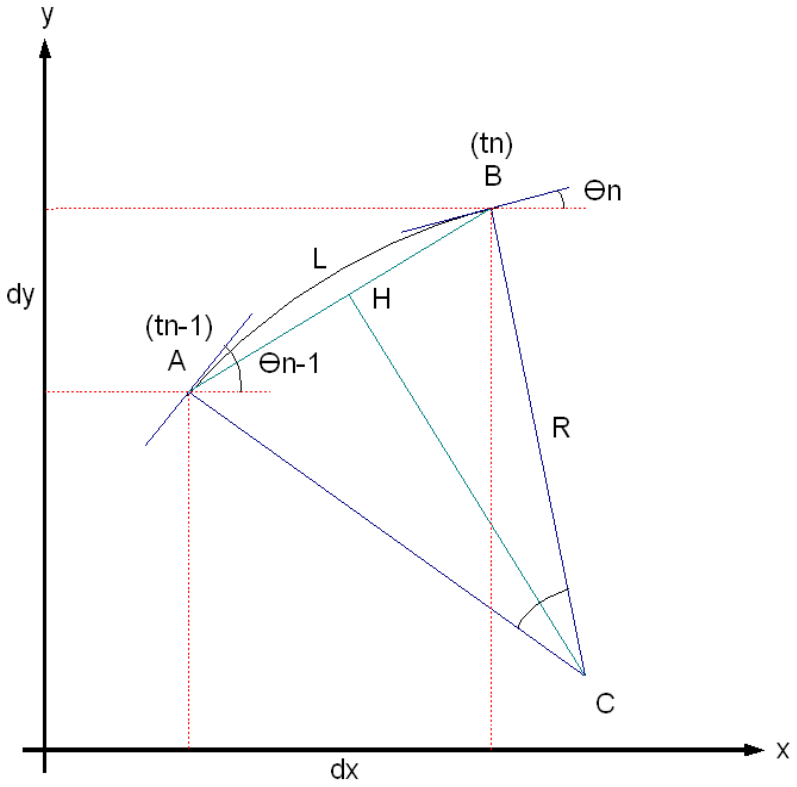
$$dx = -R.(\sin(\theta_n) - \sin(\theta_{n-1}))$$
$$dy = R.(\cos(\theta_n) - \cos(\theta_{n-1}))$$

Avec toujours la loi de récurrence de l'intégration numérique :

$$x_n = x_{n-1} + dx$$
$$y_n = y_{n-1} + dy$$

.

Approximation circulaire. Version 2:



On approxime la trajectoire entre A et B par un arc de cercle.
L'angle au centre inscrit correspondant vaut en valeur absolue :

$\Delta\theta n$=val_absolue(**$\theta n - \theta n_{-1}$**)

On en déduit le rayon du cercle (rayon de courbure de la trajectoire)

$R = L/\Delta\theta n$

Dans le triangle rectangle AHC :

$AH = R.\sin(\Delta\theta n/2)$

Soit **K** le rapport entre les longueurs de la sécante AB et de l'arc de cercle

$K = AB/L = \sin(\Delta\theta n/2) / (\Delta\theta n/2)$

On voit apparaître la célèbre fonction en sinx/x (K<1 et K→1 si $\Delta\theta n \rightarrow 0$ **)**
Orientation de la sécante AB :

$\theta moy = \frac{1}{2}(\theta n + \theta n_{-1})$

Projection de la sécante AB sur les 2 axes : (Sachant que AB=K.L)

$dx = K.L.\cos(\theta moy)$
$dy = K.L.\sin(\theta moy)$

Loi de récurrence de l'intégration numérique :

$xn = xn_{-1} + dx$
$yn = yn_{-1} + dy$

Comparaison entre les 2 algorithmes circulaires :

Rappel algorithme circulaire version1 :

dx = -R.(sin(θn) - sin(θn-1))
dy= R.(cos(θn) - cos(θn-1))

Rappel algorithme circulaire version2 :

dx = K.L.cos(θmoy)
dy = K.L.sin(θmoy)
avec K= sin (Δθn/2) / (Δθn/2)
θmoy= ½(θn +θn-1)

En fait il s'agit des mêmes relations mais l'une est présentée sous la forme d'une différence de sinus cosinus et l'autre sous la forme d'un produit de sinus cosinus. Pour les incrédules, faire appel aux propriétés classiques des fonctions trigo pour démontrer qu'il y a bien équivalence entre les 2 versions ou se reporter à l'ANNEXE 1 en fin de document.

Que choisir ?
Si on tatillonne, on dira : Avantage à la version 2 pour le temps de calcul car 3 fonctions trigo à calculer au lieu de 4 pour la version1. Mais cet avantage n'est recevable que pour un IUT utilisant des micros poussifs avec de vieux profs qui répugnent à changer leurs vieilles habitudes. (Clin d'œil à une remarque à intention péjorative, trouvée sur le site d'une équipe concurrente qu'on ne nommera pas).
Plus sérieusement, lorsque le robot se déplace en ligne droite (c'est-à-dire presque en ligne droite), **Δθn** qui doit rester nul peut prendre des valeurs minuscules et **R** des valeurs monstrueuses. La version 1 multiplie alors un infiniment grand par un infiniment petit, ce qui pose peut-être des problèmes d'erreurs de quantification.
La version 2 est plus clean de ce point de vue car elle évite le calcul du rayon et ne fait intervenir que des valeurs raisonnables.

Ainsi si on reprend l'exemple du robot RCVA lancé à 0.8 m/s avec une période d'échantillonnage de 5 ms, ça donne :
L=4mm
1**LSB** de variation sur le codeur d'une des roues donne alors :
Δθn= 4.36*10puissance-5 radians ! (Angle minuscule)
R=91.7 mètres! (Monstrueux si on compare à L=4mm)

Comparaison entre approximation linéaire et circulaire :

Rappel algorithme linéaire :

dx = L.cos(θmoy)
dy = L.sin(θmoy)
avec θmoy= ½(θn +θn-1)

Comparons aux expressions de l'approximation circulaire version 2.

On est surpris de constater que les expressions sont équivalentes au facteur **K** près**,** la fameuse fonction en sinx/x.

Prenons un exemple extrême pour chiffrer la valeur de **K.**
L'exemple est extrême dans la mesure où il peut représenter le pire des cas c'est-à-dire un robot à grande vitesse dans une courbe à rayon très court.
Vitesse=1m/s avec R=25 cm.
Tech=5 ms
Ca correspond à une force centrifuge de 60N pour un robot de 15 kg
(Notre robot RCVA est largement en perdition dans ces conditions).
L=5mm
$\Delta\theta n = L/R = 0.02$ rd
$K = \sin(0.01)/0.01 = 0.999983\ldots$
K vaut 1 à 2*10puissance-5 près !
Et il s'agit on le rappelle d'un cas extrême.
A chacun d'en tirer ses conclusions mais il semblerait que l'approximation linéaire soit suffisante en odométrie dans la plupart des cas d'un robot EUROBOT.
Il est vrai que le seul calcul de **K** permet de passer facilement d'une version linéaire à une version circulaire. Ca coûte 1 ou 2 lignes de code supplémentaires et ça mérite des essais comparatifs sur le terrain .

(Des vidéos devraient suivre pour illustrer ces résultats)

ANNEXE 1 :

Algorithme circulaire : Equivalence entre expressions des versions 1 et 2

Si on fait appel à sa mémoire reptilienne…ou plus sûrement à wikipedia

$$\sin(p)-\sin(q) = 2\sin[(p-q)/2]*\cos[(p+q)/2]$$
$$\cos(p)-\cos(q) = -2\sin[(p-q)/2]*\sin[(p+q)/2]$$

On applique aux expressions dx et dy de la version1

$$dx = -R.(\sin(\theta n) - \sin(\theta n-1))$$
$$dy = R.(\cos(\theta n) - \cos(\theta n-1))$$

$$dx = -2R.\sin(\Delta\theta n/2).\cos(\theta moy)$$
$$dy = 2R.\sin(\Delta\theta n/2)*\sin(\theta moy)$$
$$\text{avec } R = L/\Delta\theta n$$

Et on aboutit aux expressions de la version2

$$dx = L.\ [\sin(\Delta\theta n/2)]/\ (\Delta\theta n/2\ )*\cos(\theta moy)$$
$$dx = L.\ [\sin(\Delta\theta n/2)]/\ (\Delta\theta n/2\ )*\sin(\theta moy)$$

ANNEXE 2:

Exemple de programme en langage C

Dans ce programme les variables distance et orientation sont exprimées dans le système d'unité ROBOT c'est-à-dire en nombre de pas incrémentaux.
Bien sûr, les orientations seront provisoirement converties en radians pour les calculs des sinus et cosinus.
Toutes les variables pourraient être de type flottant mais pour des raisons d'éthique et autre, les types des variables seront adaptés à leur nature.

```
/**********  Définition des constantes   ***************************

#define COEF_D        3892L          // 1 cm → 3892 pas incrémentaux
#define COEF_ROT      2036L          // 1 degré → 2036 pas
#define PI            3.14159265
#define CONV_RD       PI/(180*COEF_ROT)
                                     //conversion teta en pas → teta en radian
#define LINEAIRE          1
#define CIRCULAIRE        2


/*********   Définition des variables globales  **********************
En fait ces variables seront rangées avantageusement dans des structures
*****************************************************************/
long  gauche, droite, distance,distance_precedente ; // variables entières sur32 bits
long  orient,  orient_precedente,  orient_init ;
float   x,  y;
int     x_cm,  y_cm ;        // variables sur 16 bits

char   algo ;   // Variable définie uniquement pour des raisons de présentation
                // En pratique le choix sera définitif entre algo linéaire ou circulaire


/************  Conditions initiales   ************************************
Fonction appelée 1 fois au début du programme qui initiale les variables orientation,
x, y en fonction de la position du robot au  départ
*********************************************************************/

void fonction_initialisation (long x0, long y0,  long teta0)
          { orient_init = teta0 ;
            orient_precedente=teta0
            x= x0 ;
            y= y0 ;
            distance=0 ;
            distance_precedente=0 ;
           }
```

```
/ ********** Interruption **********************************
Fonction déclenchée par Timer réglé à 5 ms.
Cette fonction est appelée toutes les 5 ms pendant toute la durée du match
*************************************************************/
void prog_interruption (…)
    {saisie_capteurs_incrementaux( ) ; //Actualise les variables gauche et droite
     calcul_xy ( ) ;                   // La fonction qui nous intéresse
     calcul_des_consignes( ) ;         // Les fameuses consignes trapézoïdales
     asservissement_polaire ( ) ;      // Asservit le robot sur ses consignes
     …
     }
/********** Calcul de x et y ***********************************************
Cette fonction récupère les 2 variables globales gauche et droite et actualise les
valeurs de x et y. Le calcul de x_cm et y_cm peut être fait en dehors de la fonction
*************************************************************************/
void calcul_xy (void)
  { int   delta_d, delta_orient ;
    long  orient_moy ;
    float orient_moy_radian, delta_orient_radian ;
    float K, dx,dy;
                          // Correspondance avec notation du document théorique
    distance= ½( droite + gauche) ;        // distance en pas parcourue à tn
    orient= orient_init + (droite – gauche) ;   //correspond à θn mais en pas
    delta_d =distance– distance_precedente;     // correspond à L mais en pas
    delta_orient= orient– orient_precedente ;   // correspond à Δθn mais en pas
    orient_moy = ½* (orient + orient_precedente) ; // correspond à θmoy en pas
    delta_orient_radian = CONV_RD * delta_teta;  // correspond à Δθn en rd
    orient_moy_radian = CONV_RD * orient_moy ; // correspond à θmoy en rd
  if (algo==LINEAIRE)
          K=1;
  else  // Choix de l'algo circulaire
          { if (delta_orient_radian ==0) // Pour éviter la division par zéro
                K=1;
            else
                K=( sin(delta_orient_radian /2)) / (delta_orient_radian /2);
          }
    dx=K* delta_d *cos(orient_moy_radian);
    dy=K* delta_d *sin(orient_moy_radian);

     x= x + dx;  // valeurs exprimées dans le système d'unité robot
     y= y + dy;

     x_cm= (int) x/COEF_D;  // Ce calcul peut être fait en dehors de la fonction
     y_cm= (int) y/COEF_D;  //  quand on en a besoin
     orient_precedente  = orient  ;        // actualisation de θn-1
     distance_precedente = distance ;      //actualisation de Dn-1
   }
```

ANNEXE 3

Correction de force centrifuge :

Un véhicule se déplaçant sur une trajectoire courbe est soumis à la force centrifuge. (Centrifuge ou centripète, c'est comme on veut).
Cette force qui tire le robot à l'extérieur de sa trajectoire, induit des déplacements non pris en compte par une odomètrie classique.
La méthode présentée ici permet de corriger dans une certaine mesure ces erreurs odomètriques induites.
La force centrifuge **F** étant perpendiculaire à la tangente à la trajectoire, l'idée est de supposer que le déplacement latéral résultant est proportionnel à cette force. Le coefficient de proportionnalité sera déterminé expérimentalement.

Cette force centrifuge peut prendre les 3 formes équivalentes.
$F= M.v^2/R = M.(d\theta/dt)^2.R = M.(d\theta/dt).v$

F: force centrifuge
M: masse du véhicule
v: vitesse linéaire
$d\theta/dt$: vitesse de rotation
R: Rayon de courbure de la trajectoire

Pour ce calcul, la 3eme forme **$F= M.(d\theta/dt).v$** est la plus intéressante car elle ne fait pas intervenir **R** .

Si on se reporte à l'exemple du programme en C
$d\theta/dt$ est représenté par **delta_orient**
v est représenté par **delta_d**

Cette force centrifuge étant perpendiculaire à la trajectoire, à l'intant tn, son orientation est perpendiculaire à l'orientation **orient_moy**

Si on admet que la dérive du robot est proportionnelle à cette force et dans la même direction, en projetant sur les 2 axes ça donne :

```
derive_x= -CORFUGE* delta_orient* delta_d *sin(orient_moy_radian);
derive_y=  CORFUGE* delta_orient* delta_d *cos(orient_moy_radian);
```

CORFUGE: constante de correction centrifuge déterminée expérimentalement.

Rappelons les expressions de **dx** et **dy** sans correction centrifuge

```
dx=K* delta_d *cos(orient_moy_radian);
dy=K* delta_d *sin(orient_moy_radian);
```

Si on suppose que k est très voisin de 1, on peut alors écrire en combinant :

```
derive_x = - CORFUGE* delta_orient*dy;
derive_y =   CORFUGE* delta_orient*dx;
```

Exemple CORFURE=1.2 (valeur déterminée pour le robot RCVA mais à adapter)
A noter la relation croisée entre x et y.
A noter aucune nouvelle fonction trigo à calculer !

Dans ces conditions la nouvelle fonction avec correction centrifuge deviendrait :

```
void calcul_xy_avec_correction_centrifuge (void)
  {
    int     delta_d, delta_orient ;
    long    orient_moy ;
    float   orient_moy_radian, delta_orient_radian ;
    float   K, dx,dy, derive_x, derive_y ;

                              // Correspondance avec notation du document théorique
    distance= ½( droite + gauche) ;          // distance en pas  parcourue à tn
    orient= orient_init + (droite – gauche) ;      //correspond à θn  mais en pas
    delta_d =distance– distance_precedente;      // correspond à L mais en pas
    delta_orient= orient– orient_precedente ;    // correspond à Δθn mais en pas
    orient_moy = ½* (orient + orient_precedente) ; // correspond à θmoy en pas
    delta_orient_radian =  delta_teta/ COEF_ROT;  // correspond à Δθn  en rd
    orient_moy_radian = orient_moy/ COEF_ROT ; // correspond à θmoy en rd
  if (algo==LINEAIRE)
          K=1;
  else  // Choix de l'algo circulaire V2
          { if (delta_orient_radian ==0) // Pour éviter la division par zéro
                K=1;
            else
                K=( sin(delta_orient_radian /2)) / (delta_orient_radian /2);
          }
    dx=K* delta_d *cos(orient_moy_radian);
    dy=K* delta_d *sin(orient_moy_radian);

    derive_x = - CORFUGE* delta_orient*dy;    // Correction de dérive
    derive_y =   CORFUGE* delta_orient*dx;

    x= x + dx + derive_x;  // valeurs exprimées dans le système d'unité robot
    y= y + dy +  derive_y;

    orient_precedente  = orient  ;
    distance_precedente = distance ;
  }
```