ENSTA ParisTech
2017/2018

Model-based design and automatic code generation – TCS301

Travaux Dirigés & Travaux Pratiques

Feuille n° 1

Objectif(s)

- Programmation à flots de données en C, algorithme de filtrage
- Programmation par automates en C

Exercice 1 – Flots de données

Un logiciel embarqué capte des données venant du monde extérieur au travers de capteurs. Ces derniers renvoient généralement des données bruitées qu'il convient de nettoyer, filtrer pour avoir des données utilisables par l'algorithme de contrôle.

Dans cet exercice nous allons considérer un filtre linéaire très simple, nommé moyenne mobile, à des ordres variables. Nous allons mettre en œuvre cet algorithme en langage C.

Ce filtre est décrit par une équation de récurrence de la forme suivante :

$$y_n = \frac{1}{p+1} \sum_{i=0}^{p} x_{n-i} \ .$$

où

— x_i sont les entrées du filtre, c'est-à-dire les données fournies par le capteur
— y est la sortie du filtre ;
— p représente l'ordre du filtre, *i.e.*, le nombre d'entrées mémorisées.

NB : Les mémoires de x sont initialisées à zéro.

La structure d'un programme réactif est de la forme suivante :

```
n ← 0;
y ← 0;
while n < MaxN do
    Print(n, y);
    Read x;
    y ← Filtre(x);
    n ← n+1;
end
```

Question 1

Donner l'expression du filtre à l'ordre 3.

Solution:

$$y_n = \frac{1}{4} \left(x_n + x_{n-1} + x_{n-2} + x_{n-3} \right)$$

Question 2

Mettez en œuvre le filtre moyenne mobile à l'ordre 3, dans une fonction C. Vous utiliserez le mécanisme des variables `static` du langage C pour les mémoires.

Solution:

```
/* Filtre lineaire simple: moyenne mobile */
double filtre (double x) {
  /* Memoires */
  static double x1 = 0.0;
  static double x2 = 0.0;
  static double x3 = 0.0;

  /* Moyenne des entrees */
  double y = 1./4. * (x + x1 + x2 + x3);

  /* Mise a jour des memoires */
  x3 = x2;
  x2 = x1;
  x1 = x;

  /* Resultat */
  return y;
}
```

Question 3

Mettez en place la simulation de ce filtre avec comme entrée une fonction sinus bruitée avec une valeur aléatoire comprise en -0.5 et 0.5. Notez qu'il faut échantillonner la fonction sinus, c'est-à-dire, $\sin(n \cdot \Delta t)$. Dans cette question nous prendrons un pas d'échantillonnage $\Delta t = 0.1$.

Vous pourrez utiliser le logiciel `gnuplot` pour afficher le résultat de la simulation. Vous afficherez l'entrée du filtre et la sortie du filtre. En mettant le résultat du programme dans le fichier "res.txt", l'affichage se réalise par la commande suivante :

```
echo "plot 'res.txt' u 1 w lp, '' u 2 w lp" | gnuplot
```

Solution:

```
int main (int argc, char** argv)
{

  unsigned int n = 0;
  double x = 0.0;
  double y = 0.0;
  double z = 0.0;

  srand(time(NULL));

  for (n = 0; n < MAX_ITER; n++)
    {
      //printf ("%f %f %f\n", x, y, z);
      printf ("SSM::set Filtre/x %f\nSSM::cycle 1\n", x);

      x = sin(n * DT) + (rand() / (double) RAND_MAX) - 0.5;
      y = filtre (x);
      z = filtreN (x);

    }

  return EXIT_SUCCESS;
}
```

Question 4

Une façon d'avoir un ordre plus important est d'appeler le filtre sur le résultat d'un premier filtre.

Mettez en œuvre cette fonction et observer le comportement.

Solution:
Cela ne fonctionne pas à cause des mémoires partagées. Il faut dupliquer le code du filtre ou faire autrement pour les mémoires (cf exercice sur les machines à états).

Question 5

Nous voulons avoir un filtre d'ordre arbitraire à l'aide d'un tableau. Donnez une nouvelle implémentation du filtre à moyenne mobile à l'aide d'un tableau.

1. En faire à nouveau la simulation en fixant l'ordre du filtre à 20. Que constatez vous ?
2. Que se passe-t-il si l'ordre du filtre est de 63 ?

Solution:

```
/* Filtre lineaire simple: moyenne mobile d'ordre arbitraire */
double filtreN (double x) {
  /* Memoires */
  const unsigned int size = 19;
  static double xi[size];
  unsigned int i = 0;

  /* Moyenne des entrees */
  double y = x;
  for (i = 0; i< size; i++) {
    y += xi[i];
  }
  y *= (1.0 / size);

  /* Mise a jour des memoires */
  for (i = size - 1; i > 0; i--) {
    xi[i] = xi[i-1];
  }
  xi[0] = x;

  /* Resultat */
  return y;
}
```

1. Le signal est mieux filtré avec un ordre élevé mais cela crée un déphasage dû à la mémorisation plus importante.
2. Si le filtre est d'ordre 63, c'est-à-dire 2π, alors nous ne filtrons que le bruit qui a une moyenne nulle.

Exercice 2 – Machines à états

Dans les logiciels embarqués, une part importante des calculs est réservée à la prise de décisions. Une façon concise de modéliser ces décisions est donnée par des machines à états ou automates.

Une forme classique de code en langage C d'une machine à états est donnée dans listing 1.

Le but de cet exercice est de modéliser et simuler à l'aide d'une machine à états un voteur. En effet, certaines informations dans un système embarqué sont critiques et donc une duplication des données est nécessaire pour s'assurer d'avoir une valeur au moment voulu.

Nous prendrons un exemple simple de redondance en considérant un capteur qui est dupliqué. Nous avons donc deux capteurs qui renvoient normalement la même valeur à chaque instant. Dès que ces capteurs ne renvoient pas la même valeur alors une erreur est détectée. Le voteur est un algorithme qui prend en entrée les informations des capteurs et renvoie une seule valeur représentative des capteurs et informations d'erreur le cas échéant.

Le voteur est modélisé par l'automate donné à la figure 1. Les deux entrées sont nommées i_0 et i_1, la sortie est notée o alors que la sortie d'erreur est notée e. Les entrées i_0, i_1 et la sortie o sont des valeurs entières alors que la sortie erreur e est une valeur booléenne.

Listing 1 – Exemple de programmation de machine à états en C

```
#include <stdio.h>

enum states { STATE1, STATE2, STATE3 };

void step(enum states *state, int input, int* output)
{
  /* Calculs initiaux possibles */

  switch(*state) {
  case STATE1:
    /* Do something */
    *state = STATE2; /* Next state */
  }
    break;
 case STATE2:
   /* Do something */
   *state = STATE1; /* Next state */
   break;

 case STATE3:
   /* Do something and remain in this state */
   break;
  }

  /* Calculs finaux possibles */
}

int main(void)
{
  enum states state = STATE1;
  int input;
  int output = 0;

  while(true) {
    input = /* READ_SOMETHING */;
    step(&state, input, &output);
    printf ("output = %d\n", output);
  }

  return 0;
}
```

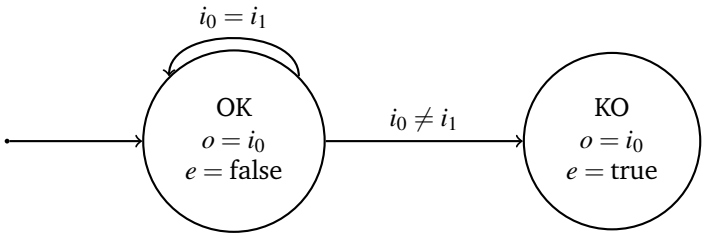

FIGURE 1 – Fonctionnement du voteur décrit par un automate

Question 1

Quels types de données utilisez vous pour modéliser les entrées et les sorties ?

Solution:
- Un tableau d'entiers pour les entrées
- Une structure pour la sortie contenant une valeur entière et une valeur booléenne pour l'erreur.

Question 2

Donner l'implémentation de l'automate de la figure 1 en langage C.

Solution:

```
struct output_t step (enum states_t *state, int* inputs)
{
  /* By default, the result is an error (defensive coding style). */
  struct output_t result = { 0, true } ;

  switch (*state) {
  case STATE1:
    /* All input agreed by default */
    if (inputs[0] == inputs[1]) {
      result.output = inputs[0];
      result.error = false;
      *state = STATE1;
    }

    if (inputs[0] != inputs[1]) {
      result.output = inputs[0];
      result.error = true;
      *state = STATE2;
    }
    break;

  case STATE2:
    /* Error detected */
    result.output = inputs[0];
    result.error = true;
    *state = STATE2;
    break;
  }
  return result;
}
```

Question 3

Pour faire la simulation d'un tel composant, il faut un générateur d'entrées. Écrivez une fonction qui génère aléatoirement des entrées qui dans 95% des cas renvoie deux valeurs d'entrées égales et autrement deux valeurs d'entrées qui sont différentes.

Solution:

```
void generateInput (int* input)
{
  int toto = rand ();

  double flag = rand () / (double) RAND_MAX;

  if (flag >= 0.95) {
    input[0] = toto;
    input[1] = rand();
  }
  else {
    input[0] = toto;
    input[1] = toto;
  }
}
```

Question 4

Simulez le comportement du composant et vérifiez graphiquement qu'une fois une erreur détectée la valeur de e vaut toujours vraie.

Solution:

```
int main (int argc, char** argv)
{

  unsigned int n = 0;
  enum states_t state = STATE1;
  struct output_t result;
  int inputs[2];

  srand(time(NULL));

  for (n = 0; n < MAX_ITER; n++) {
    generateInput(inputs);
    result = step(&state, inputs);
    printf ("%d %d\n", result.output, result.error);
  }

  return EXIT_SUCCESS;
}
```