La vache est dans le pré

Projet tutoré du semestre 4, année universitaire 2020-2021.

Version 2 du 27/04/21 – Modification du paragraphe 2.1.2

1 Le problème à traiter

Ceci est la triste histoire d'une vache victime des facéties d'un paysan taquin.

En effet, Fernand le paysan décida un beau jour de jouer quotidiennement la blague suivante à son voisin Raoul, également paysan : nuitamment, Fernand se rend dans le pré de Raoul, déplace tout ou partie des piquets de la clôture et positionne la vache dudit Raoul exactement au centre de gravité du pré ainsi modifié.

Dans un premier temps, quand il s'aperçut de la plaisanterie, Raoul se contenta de bougonner. Puis il se rendit compte que, suivant la forme du pré, le centre de gravité pouvait se situer en dehors de la clôture. Dans ce cas, la vache peut s'évader puisqu'elle n'est plus cernée par la clôture.

Raoul ne pouvant passer ses nuits à surveiller son pré et sa vache, il décida de faire appel à quelques siens amis, étudiants en seconde année de DUT Informatique à l'IUT d'Amiens, qui lui tinrent à peu près ce langage :

> *Nous pouvons concevoir un programme informatique qui indique si le centre de gravité se trouve à l'intérieur ou à l'extérieur du pré. Il suffira de fournir au programme les coordonnées des piquets.*

Fort heureusement, Raoul s'est jadis lié d'amitié avec un vieil hibou savant et noctambule. Cet hibou a accepté de relever à l'aide d'un smartphone la position des piquets de clôture juste après que Fernand ait changé leur position, puis de transmettre ces informations à Raoul. Ainsi, ce dernier pourra exécuter le programme et, selon la réponse, laisser sa vache au pré (si le centre de gravité se situe dans le pré) ou la rentrer à l'étable (si le centre de gravité se situe hors du pré).

Ces étudiants, c'est vous !

2 La programmation

Le programme à rédiger accepte en entrée le nombre de piquets formant la clôture, puis les coordonnées cartésiennes (x, y) de ces piquets. Elles sont données **dans l'ordre**, comme si l'on suivait le fil de la clôture, dans le sens inverse des aiguilles d'une montre. La clôture est fermée sur elle-même et les fils ne se croisent jamais.

Raoul ne possède que cinquante piquets. Le nombre de piquets utilisés ne peut donc dépasser cette valeur.

Le programme détermine les coordonnées du centre de gravité de la clôture, puis indique si celui-ci se trouve à l'intérieur.

On donne ci-dessous un exemple d'exécution du programme en mode console sous Unix :

```
$ ./vache
  Saisir le nombre de piquets
  4
  Saisir le piquet 0
  -1.1
  -1.5
  Saisir le piquet 1
  2.1
  3.012
  Saisir le piquet 2
  5.6
  -1.21
  Saisir le piquet 3
  1.97
  4.07
  Aire = 3.563150
```

```
  Centre de gravité : 1.978486, 1.903452
  Attention, la vache est hors du pré
$
```

2.1 Quelques éléments mathématiques

Attention : dans ce qui suit, on suppose que les n segments de la clôture sont numérotés de 0 à n-1.

2.1.1 Calcul du centre de gravité d'un polygone régulier.

Aire d'un polygone régulier Ce calcul nécessite de connaître l'aire **A** du polygone, donnée par la formule suivante :

$$A = \frac{1}{2}\sum_{i=0}^{n-1}(x_i \times y_{i+1} - x_{i+1} \times y_i)$$

i+1 est calculé modulo n afin de fermer la clôture du pré : le segment n-1 est combiné avec le segment 0.

Cette quantité **A** peut être négative. Dans le cas général, on prendrait la valeur absolue de **A** pour obtenir une aire positive. Cependant, dans les calculs qui suivent, il faut garder cette valeur signée.

Centre de gravité A partir de l'aire **A**, on calcule l'abscisse G_x du centre de gravité :

$$G_x = \frac{1}{6A}\sum_{i=0}^{n-1}(x_i + x_{i+1})(x_i \times y_{i+1} - x_{i+1} \times y_i)$$

L'ordonnée G_y du centre de gravité vaut :

$$G_y = \frac{1}{6A}\sum_{i=0}^{n-1}(y_i + y_{i+1})(x_i \times y_{i+1} - x_{i+1} \times y_i)$$

2.1.2 Appartenance d'un point à un polygone

Comment savoir si le centre de gravité **G** se situe dans le polygone ? Le problème d'appartenance d'un point à un polygone **P** est un problème de base en infographie. Ce problème se résout de plusieurs façons. Dans cette étude, on se propose d'utiliser la méthode expliquée ci-dessous. Elle est relativement simple mais gourmande en calculs puisqu'elle nécessite la détermination d'un arc cosinus.

Soit θ_i l'angle **signé** entre les segments de droite $[G, S_i]$ et $[G, S_{i+1}]$, où S_i et S_{i+1} sont deux sommets consécutifs et i+1 est calculé modulo n.

On peut montrer que :

$$\sum_{i=0}^{n-1}\theta_i = 0 \Rightarrow G \notin P$$

L'algorithme est le suivant :

```
booléen AppartenancePointPolygone (polygone P, point G)
{ La fonction retourne vrai si G appartient à P, faux sinon. }
VARIABLES
  Somme, Thetai : réel
  i : entier

DEBUT
  Somme <- 0.0

  POUR i ALLANT de 0 à n-1 FAIRE
```

```
    Calculer Thetai
    Somme <- Somme + Thetai
  FINPOUR

  SI Somme = 0 ALORS
    retourner faux
  SINON
    retourner vrai
  FINSI
FIN
```

Calcul de l'angle θ_i Pour calculer l'angle θ_i entre les segments de droite [**G**, S_i] et [**G**, S_{i+1}], on utilise la relation :

$$\theta_i = \arccos\left(\frac{\overrightarrow{GS_i} \bullet \overrightarrow{GS_{i+1}}}{|\overrightarrow{GS_i}| \times |\overrightarrow{GS_{i+1}}|}\right)$$

— $\overrightarrow{GS_i} \bullet \overrightarrow{GS_{i+1}}$ représente le produit scalaire des vecteurs $\overrightarrow{GS_i}$ et $\overrightarrow{GS_{i+1}}$
— $|\overrightarrow{GS_i}|$ représente la norme du vecteur $\overrightarrow{GS_i}$

Arc cosinus retourne la valeur de l'angle, mais **non signée**. Pour connaître le signe de l'angle (sens trigonométrique ou anti-trigonométrique), on calcule le déterminant des vecteurs $\overrightarrow{GS_i}$ et $\overrightarrow{GS_{i+1}}$: si $\text{Det}(\overrightarrow{GS_i}, \overrightarrow{GS_{i+1}})$ est négatif, l'angle est négatif ; s'il est positif, l'angle est positif (il est dans le sens trigonométrique).

Coordonnées d'un vecteur Les coordonnées du vecteur $\overrightarrow{AB}$ sont :

$$\overrightarrow{AB}(x_B - x_A, y_B - y_A)$$

Produit scalaire Le produit scalaire $\overrightarrow{U} \bullet \overrightarrow{V}$ de deux vecteurs $\overrightarrow{U}$ et $\overrightarrow{V}$ est égal à :

$$\overrightarrow{U} \bullet \overrightarrow{V} = x_u \times x_v + y_u \times y_v$$

Norme d'un vecteur La norme $|\overrightarrow{U}|$ d'un vecteur $\overrightarrow{U}$ est égale à :

$$|\overrightarrow{U}| = \sqrt{{x_u}^2 + {y_u}^2}$$

Déterminant de deux vecteurs Le déterminant $\text{Det}(\overrightarrow{U}, \overrightarrow{V})$ des deux vecteurs $\overrightarrow{U}$ et $\overrightarrow{V}$ est égal à :

$$\text{Det}(\overrightarrow{U}, \overrightarrow{V}) = x_u \times y_v - y_u \times x_v$$

Attention ! Il ne faut pas s'effrayer des formules mathématiques ci-dessus. Elles représentent des calculs simples qui se programment aisément.

Par exemple, la somme suivante S suivante :

$$S = \sum_{i=1}^{6} i = 1 + 2 + 3 + 4 + 5 + 6$$

se programme aisément. En Fortran, cela donnerait :

```
program somme
integer i, S

S = 0
do i = 1, 6
   S = S + i
enddo
```

2.2 Jeux d'essais

Nb piquets	*Coordonnées des piquets*	*Aire*	*Centre gravité*	*Position vache*
4	(-1, 1)(-1, -1)(1, -1)(1, 1)	4	(0, 0)	Intérieur
4	(-16.6, -20.1)(-12.6, -18.6)(-11.6, -16.6)(-15.1, -15.1)	13.125	(-14.226, -17.555)	Intérieur
4	(-1.1, -1.5)(2.1, 3.012)(5.6, -1.21)(1.97, 4.07)	3.563	(1.978, 1.903)	Extérieur

3 Modalités de rendu du projet tutoré

3.1 L'équipe de développeurs

Ce projet est à réaliser en monôme, binôme ou trinôme.

3.2 Contraintes techniques

Le programme sera codé indifféremment sous Unix ou Windows, en objet ou en procédural, avec un langage librement choisi, du moment qu'il existe des compilateurs libres ou gratuits (afin que votre enseignant puisse tester le programme).

De même, l'interface est choisie librement (console ou graphique). Dans tous les cas, elle doit permettre une saisie des coordonnées des piquets la plus simple possible.

3.3 Éléments à rendre

Il n'est pas demandé d'analyse ni d'algorithmes : uniquement le code source commenté selon les bonnes pratiques (voir annexe).

Ce code source sera placé sur un dépôt *git* (ou tout autre gestionnaire de version). Penser à mettre les droits idoines pour que votre enseignant puisse accéder audit dépôt !

Un rapport succinct sera envoyé par mail, au format *pdf*. Il contiendra :
- les noms et groupes d'appartenance des développeurs concernés
- l'URL du dépôt *git*
- la commande de compilation permettant de générer l'exécutable
- si besoin, des explications sur le programme si vous estimez qu'une technique mise en oeuvre nécessite des précisions
- une conclusion (intérêt du projet, difficultés, axes d'amélioration...)
- en annexe, le code source du programme, imprimé en police à chasse fixe de type `Courrier`, en taille 8

3.4 Critères et barème de notation

Les points suivants seront pris en compte pour la notation :
- La décomposition judicieuse en sous programmes ou en classes
- L'utilisation de structures de données/classes adéquates
- La clarté et la lisibilité du programme (commentaires pertinents, variables bien nommées, indentation correcte – voir annexe)

Barème de notation :
- Choix des structures de données : 2 pts
- Programme principal : 3 pts
- Calcul de l'aire : 3 pts
- Calcul du centre de gravité : 3 pts
- Appartenance à la forme : 3 pts
- Qualité du programme : 4 pts
- Rapport : 2 pts
- Bonus : +2 pts (si le programme comporte une fonctionnalité supplémentaire, s'il utilise des techniques follement élégantes, s'il est codé à travers un langage inattendu ou rare, votre enseignant peut ajouter un bonus pouvant aller jusque 2 points)

Attention Le partage de code à la manière des logiciels libres est très noble. Cependant, dans le cas d'un travail noté comme ce projet, cela s'apparente à une action de fraude, comme si un étudiant en salle d'examens passait sa copie à son voisin. Les codes identiques seront donc sanctionnés.

3.5 Date de rendu

Le code source devra être disponible sur le *git* et le rapport devra être rendu par mail à

arnaud.clerentin@u-picardie.fr

avant le

lundi 17 mai 2021, 18h

Les retards seront pénalisés.

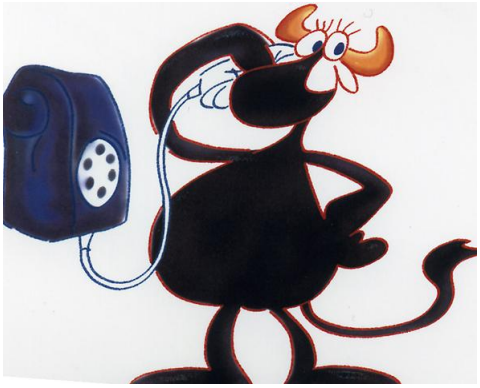

Figure 1 – La vache dont il est question dans cette étude.

Ici, mécontente des plaisanteries dont elle est l'objet, elle téléphone à son docteur.

Annexe : les bonnes pratiques de programmation

1 Coder proprement

1.1 Quelques principes

Un développeur est un **auteur**
- **Écrire** est sa profession
- Un auteur écrit pour être lu
- Un auteur classique a des outils :
  - chapitres, paragraphes, figures de style...
- Un développeur a des outils :
  - langages, classes, méthodes et procédures...

Lors de la lecture du code, notre cerveau est le compilateur : simplifions lui la tâche !

→ Principe de **moindre surprise** : le comportement d'une classe/fonction doit être évident.

1.2 Nommage

Le nom d'une variable, fonction, classe doit répondre à plusieurs questions :
- la raison de son existence
- son rôle et son utilisation

Si un nom exige un commentaire, c'est qu'il ne répond pas à ces questions.

```
int d;     // Temps écoulé en jours        NON !
int tempsEcouleEnJours;                 // OUI !
```

Autre exemple :

```
for (int j=0; j<34; j++)
   s += (t[j]*4)/5;
```

→ Illisible !

```
for (int j=0; j<nombreDeTaches; j++)
{
  totalJourTache = tache[j] * nbJourIdeal;
  tachesParJour = totalJourTache / nbJourSemaine;
  somme += tachesParJour;
}
```

→ Beaucoup plus lisible. On comprend qu'on calcule la quantité de tâches effective dans la semaine. Puis on divise ce nombre par le nombre réel de jour dans la semaine. Ce qui donne le nombre réel de tâches réalisées dans la journée.

Ne pas hésiter à scinder les lignes de code ou utiliser des constantes.

On privilégie la lisibilité des noms de variables par rapport à la concision (*prixHorsTaxe* est préférable à *pxHT*).

1.3 Les commentaires

Dans la plupart des livres sur les langages, la première instruction présentée est le **commentaire**. Mais on ne commente pas n'importe comment : le commentaire doit apporter quelque chose de plus que ce que présente le code.

On évite donc les commentaires :

— Inappropriés, c'est à dire intéressants mais ailleurs : gestion de version, bugs, auteurs..
— Sans intérêt

```
int pif = 5;      // déclaration d'un entier nommé pif initialisé à 5
```

— Obsolètes
— D'excuse, d'avertissement...

1.4 DRY

Faire quelque chose **une** fois, c'est **une** chance de se tromper. Faire quelque chose x fois, c'est x chance de se tromper !

Le **DRY** (Don't Repeat Yourself) est une philosophie de développement consistant à éviter la **redondance de code**, issue par exemple de copier/coller intempestifs. Ceci passe notamment par l'utilisation de sous programmes.

1.5 KISS

Le principe **KISS** (*Keep it simple, stupid*) est une ligne directrice de conception qui préconise la simplicité dans la conception et que toute complexité non indispensable devrait être évitée dans la mesure du possible. La complexité est en effet une source de coûts de conception et de maintenance, ainsi qu'une source potentielle d'erreurs.

Il est important de noter que le principe KISS proscrit les seules complexités non indispensables. Paradoxalement, tenter d'utiliser des moyens simples pour résoudre un problème complexe peut conduire à une complexité encore plus grande. En outre, la complexité est parfois utile pour assurer de bonnes performances. Dans ce cadre, l'idée est de ne pas optimiser quoi que ce soit avant de maîtriser totalement une version simple de ce que l'on crée.

« *La simplicité est la sophistication suprême.* » — Léonard de Vinci

2 Normes et conventions

L'heure n'étant plus à l'artisanat, l'informatique tend à **normaliser** ses processus (codage, test...)

Ce besoin de normalisation est apparu suite à des constats simples :
— 80% du temps passé dans le cycle de vie d'un logiciel intervient pendant le cycle de maintenance
— Il est très rare qu'un code soit maintenu par son auteur original
→ De bonnes **conventions adoptées par tous** permettent à un développeur de s'y retrouver plus facilement dans un code inconnu

Les règles de codage s'articulent autour de plusieurs thèmes, les plus courants étant :
— le nommage et l'organisation des fichiers,
— le style d'indentation,
— les conventions de nommage,
— les commentaires et documentation,
— recommandations sur la déclaration des variables, l'écriture des instructions...

2.1 Taille des fichiers source

Un fichier ne doit pas dépasser 1000 lignes de codes

2.2 Taille des méthodes

Il est déconseillé d'écrire des méthodes de plus de 15 lignes.

Certains traitements peuvent néanmoins justifier un dépassement de ce nombre de lignes, mais des méthodes trop longues traduisent un mauvais découpage algorithmique et conduisent généralement à l'obtention d'un code incompréhensible.

2.3 Taille des lignes

La lecture du code lors des phases de maintenance se fait de bas en haut et l'utilisation d'une barre de défilement horizontal est pénalisante à la lecture.

→ Au maximum 80 caractères par ligne.

En C#, il est possible d'insérer un retour à la ligne au milieu d'une instruction.

```
int retour = maMethode(parametre1, parametre2, parametre3,
      parametre4, parametre5);
```

2.4 Documentation des classes et des méthodes

Ils permettent de spécifier le comportement de l'applicatif.

Pour les méthodes, la documentation décrit :
- La liste des paramètres d'entrée
- Le retour attendu
- Les exceptions/erreurs renvoyées (selon les langages)
- Un court descriptif de la méthode

Pour les classes, la documentation la décrit.

Sous Visual Studio, la documentation est introduite par trois slashes « `///` » tandis que des logiciels comme *Doxygen* permettent d'en tirer une documentation technique au format *pdf* ou *html*. Cette documentation est également utilisée par le système Intellisense de Visual Studio quand le programmeur fait appel aux classes ou aux méthodes ainsi documentées.

Exemple.

```
/// <summary>
/// Calcul le prix d'un ensemble de baguettes
/// </summary>
/// <returns>Prix total des baguettes</returns>
/// <param name="nbBaguettes">Nombre de baguettes</param>
/// <param name="prixUneBaguette">Prix d'une baguette</param>
public static double PrixBaguettes(int nbBaguettes, double prixUneBaguette)
{
   ...
}
```

2.5 Une norme de nommage : CamelCase

Visual Studio suit la norme de nommage *Camel Case* : les différents mots constituant un identificateur commencent par une majuscule et ne comportent pas de caractère de séparation (exemple : méthode *DecodageCarte*). Il n'est pas recommandé d'utiliser des caractères accentués (tous les systèmes d'exploitation ne les gèrent pas correctement).

Pour les identificateurs de variables, le premier mot commence par une minuscule. Par exemple : *int largeurPixel ;*

2.6 Les déclarations de variables

On conseille **une seule** déclaration par ligne.

```
string nom, prenom;          // ce type de déclaration n'est pas recommandé
```

On conseille d'éviter de déclarer des variables de types différents sur la même ligne même si cela est accepté par le compilateur.

```
int age, notes[];            // ce type de déclaration est à éviter
```

Il est préférable de rassembler toutes les déclarations d'un bloc au début de ce bloc (un bloc est un morceau de code entouré par des accolades).

2.7 Les séparateurs

L'usage des séparateurs tels que les retours à la ligne, les lignes blanches, les espaces... permet de rendre le code moins dense et donc plus lisible.

2.7.1 Les lignes blanches

Elles permettent de définir des sections dans le code pour effectuer des séparations logiques.

Une ligne blanche devrait toujours être utilisée dans les cas suivants :
- avant la déclaration d'une méthode
- entre les déclarations des variables locales et la première ligne de code
- avant un commentaire d'une seule ligne
- avant chaque section logique dans le code d'une méthode

2.7.2 Les instructions

Même s'il est possible en C# de mettre plusieurs instructions sur une ligne, chaque ligne devrait n'en contenir qu'une seule.

```
prix = 5.32; nb++;           // à éviter
```