2.7 Ellipse

Solutions p. 223

Une ellipse est l'ensemble des points dont la somme des distances à deux points fixes F et F', dits foyers, est constante. Dans un repère orthonormé du plan affine, dont les vecteurs directeurs sont parallèles aux axes de l'ellipse et où l'ellipse est centrée à l'origine, l'équation paramétrique de celle-ci est :

$$\begin{cases} x &= a \times \cos\theta \\ y &= b \times \sin\theta \end{cases}$$

où $a \in \mathbb{R}^{+*}$ est le demi-grand axe, $b \in \mathbb{R}^{+*}$ est le demi-petit axe et $\theta \in [0, 2\pi[$. Dans ce cas, on a :

$$M(x,y) \in \mathcal{E} \quad \Leftrightarrow \quad \exists \theta \in [0, 2\pi[,\ x = a\cos\theta \wedge y = b\sin\theta$$

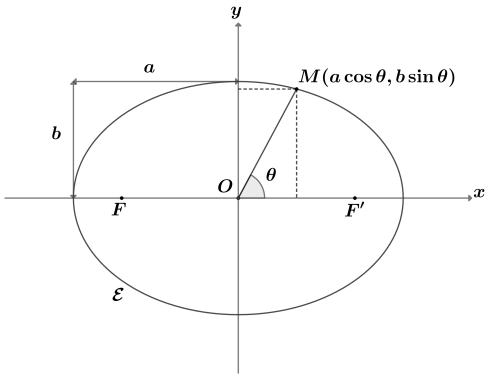

Schéma d'une ellipse

La formule générale pour calculer le périmètre P d'une ellipse est égale à :

$$P = 4\int_0^{\frac{\pi}{2}} \sqrt{x'(\theta)^2 + y'(\theta)^2}\, d\theta = 4a\int_0^{\frac{\pi}{2}} \sqrt{1 - \left(1 - \frac{b^2}{a^2}\right)\sin^2\theta}\, d\theta$$

Le souci avec cette formule, c'est qu'on ne peut pas déterminer la valeur exacte de cette intégrale. Le but de ce projet est d'approcher le périmètre d'une ellipse de différentes façons.

Exercice 1 : Fichiers

1) Créer un fichier nommé `Nom_Ellipse.ml`.

2) Écrire une expression principale qui ne fait qu'afficher le texte « Ellipse. ». Cette expression sera l'expression principale qui servira de test. Aucun affichage ne doit avoir lieu dans une autre fonction.

3) Écrire un `Makefile`. Tester le programme.

Partie A. Formule de Ramanujan

Ramanujan (1887 - 1920) est un mathématicien indien qui a travaillé principalement sur les fonctions elliptiques et sur la théorie analytique des nombres. Il est connu comme « grand créateur de formules mathématiques ».

Le périmètre d'une ellipse peut être approchée avec souvent plus d'une dizaine de chiffres exacts dans la partie décimale avec la deuxième formule de Ramanujan :

$$P \approx \pi(a+b)\left(1+\frac{3h}{10+\sqrt{4-3h}}\right) \text{ où } h=\frac{(a-b)^2}{(a+b)^2}.$$

<u>Exercice 2</u> : Formule de Ramanujan

1) Écrire une fonction de signature

```
val ramanujan : float -> float -> float
```

qui prend en paramètres les deux demi-axes $a \in \mathbb{R}^{+*}$ et $b \in \mathbb{R}^{+*}$ et qui renvoie une approximation du périmètre de l'ellipse associée. Si $a \leqslant 0$ ou $b \leqslant 0$ alors la fonction doit renvoyer une erreur.

2) Tester cette fonction et afficher le résultat pour avoir 10 chiffres après la virgule.

Partie B. Archimède

Une autre façon d'approcher le périmètre d'une ellipse consiste à adapter la méthode d'Archimède, en utilisant des polygones inscrits ayant $p = 2^n$ côtés avec $n \in \mathbb{N}$ et $n \geqslant 2$.

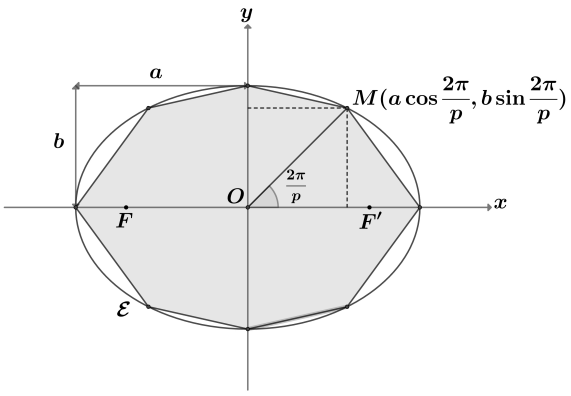

Illustration de la méthode d'Archimède pour $n = 3$

Dans cette partie, on travaille avec des points choisis sur une ellipse qui correspondent aux sommets du polygone inscrit. L'idée est d'additionner les longueurs des côtés de ce polygone pour approcher le périmètre de l'ellipse par défaut.

<u>Exercice 3</u> : Programmation impérative

Un point sur l'ellipse est représenté par une paire de nombres flottants : l'abscisse et l'ordonnée de ce point dans un repère orthonormé.

1) Écrire une fonction de signature

```
val distance : float * float -> float * float -> float
```

qui renvoie la distance entre deux points donnés en paramètre.

2) Donner les coordonnées des sommets M_i, avec $0 \leqslant i < 2^n$, du polygone inscrit à l'ellipse, à $p = 2^n$ côtés, en fonction de l'angle $(\overrightarrow{0F'}, \overrightarrow{OM_i})$.

3) Écrire, en utilisant la programmation impérative, une fonction de signature

```
val approx_perim_ellipse1 : int -> float -> float -> float
```

qui prend en paramètres un entier $n \geqslant 2$ et deux nombres flottants a et b qui correspondent aux demi-axes d'une ellipse. La fonction calcule le périmètre du polygone inscrit à 2^n côtés et renvoie une erreur si $n < 2$.

4) Calculer une approximation du nombre π en utilisant la fonction précédente.

5) Afficher, de $n = 5$ à $n = 29$, la différence entre l'approximation du nombre π calculé et le nombre π fourni dans le module `Float`, avec 20 chiffres après la virgule. Discuter des résultats. *Afin d'afficher les résultats des calculs au fur et à mesure, sans attendre la fin des instructions, on peut utiliser la fonction* `flush stdout` *de* `OCaml`.

Exercice 4 : Programmation fonctionnelle

Afin de résoudre le problème observé à la fin de l'exercice précédent, lié à l'utilisation de la programmation impérative en partie, on reprend la même approximation en adoptant une approche fonctionnelle différente.

Soient $l_0, l_1, l_2, \ldots l_{2^n-1}$ les longueurs des côtés du polygone, où l_i représente la longueur du segment reliant M_i à M_{i+1}. Plutôt que d'ajouter directement chaque l_i à un accumulateur, on modifie l'ordre des opérations : on commence par additionner les longueurs deux par deux, puis on additionne deux par deux les résultats obtenus précédemment, et ainsi de suite, jusqu'à obtenir la somme finale.

L'idée est de diviser l'angle au centre de l'ellipse n fois, en additionnant à chaque étape récursive les arcs gauche et droite. Cette approche réorganise l'ordre des additions et fournit ainsi une approximation plus précise que celle obtenue avec la version impérative.

1) Écrire, en utilisant cette nouvelle approche, une fonction de signature

```
val approx_perim_ellipse2 : int -> float -> float -> float
```

qui prend en paramètres un entier $n \geqslant 2$ et deux nombres flottants a et b qui correspondent aux demi-axes d'une ellipse. La fonction calcule le périmètre du polygone inscrit à 2^n côtés et renvoie une erreur si $n < 2$.

2) Calculer une approximation du nombre π en utilisant la fonction précédente.

3) Afficher, de $n = 5$ à $n = 29$, la différence entre l'approximation du nombre π calculé et le nombre π fourni dans le module `Float`, avec 20 chiffres après la virgule. Discuter des résultats.

2.8 Ellipse

Énoncé p. 220

Exercice 1 : Fichiers

1)
```
$ touch Grimaud_Ellipse.ml
```

2)
```
let ()=
  Printf.printf "Ellipse\n"
```

3)
```
SRC := Grimaud_Ellipse
EXE := ellipse
.PHONY: clean
$(EXE): $(SRC).ml
    ocamlopt -o $@ $<
clean:
    rm *.cmi *.cmo *.cmx *.o $(EXE)
```

```
$ make
$ ./ellipse
```

Exercice 2 : Formule de Ramanujan

1)
```
let ramanujan a b =
  if (a<=0.) || (b<=0.) then
    failwith "ramanujan. Wrong parameters, need a>0 and b>0."
  else
    let h=((a-.b)*.(a-.b)) /. ((a+.b)*.(a+.b)) in
    Float.pi *.(a +. b)*.(1. +. 3.*.h /. (10. +. sqrt(4. -. 3. *. h)))
```

2)
```
let a=6. and b=3. in
  Printf.printf "Périmètre de l'ellipse pour a=%.2f et b=%.2f avec la
    formule de Ramanujan vaut %.10f.\n" a b (ramanujan a b)
```

Exercice 3 : Programmation impérative

1)
```
let distance point1 point2 =
  let x1 = fst point1 in
  let y1 = snd point1 in
  let x2 = fst point2 in
  let y2 = snd point2 in
  sqrt((x1 -. x2) *. (x1 -. x2) +. (y1 -. y2) *. (y1 -. y2))
```

2) $M_i\left(a\cos\dfrac{2i\pi}{p}, b\sin\dfrac{2i\pi}{p}\right)$.

3)
```
let approx_perim_ellipse1 n a b =
  if n<2 then
    failwith "approx_perim_ellipse1. Wrong parameter, need n>=2."
  else
    let p = int_of_float(2.**float_of_int(n)) in
    let perim = ref 0. in
    for i=0 to p-1 do
      let a1 = float_of_int(i) *. 2. *. Float.pi /. float_of_int(p) in
      let a2 = float_of_int(i+1) *. 2. *. Float.pi /. float_of_int(p) in
      let point1 = (a *. (cos a1), b *. (sin a1)) in
      let point2 = (a *. (cos a2), b *. (sin a2)) in
      perim := !perim +. (distance point1 point2)
    done;
    !perim
```

4)
```
let n = 5 and a=1. and b=1. in
  let approx_pi = ((approx_perim_ellipse1 n a b)/. 2.) in
  Printf.printf "n=%d. Pi : %.20f.\n" n approx_pi
```

5)
```
let a=1. and b=1. in
  for n=5 to 29 do
    let approx_pi1 = ((approx_perim_ellipse1 n a b)/. 2.) in
    Printf.printf "n=%d. Difference with pi :%.20f\n" n (Float.pi -.
    approx_pi1);
    flush stdout
  done
```

Au fur et à mesure que la valeur de n augmente jusque $n = 23$, l'approximation de π devient de plus en plus proche à la valeur donnée par `Float.pi`. En revanche, la valeur calculée fluctue à partir de $n = 24$, dépassant parfois π. L'ajout des toutes « petites » valeurs de segments provoque des erreurs. L'accumulation de ces erreurs de calcul due à la représentation des nombres flottants explique cette incohérence.

Exercice 4 : Programmation fonctionnelle

1)
```
let approx_perim_ellipse2 n a b =
  if n<2 then
    failwith "approx_perim_ellipse2. Wrong parameter, need n>=2."
  else
    let rec aux n a b theta1 theta2 =
      if n=0 then
        let point1 = (a *. (cos theta1), b *. (sin theta1)) in
        let point2 = (a *. (cos theta2), b *. (sin theta2)) in
        distance point1 point2
      else
        (aux (n-1) a b theta1 ((theta1+.theta2)/.2.)) +.
        (aux (n-1) a b ((theta1+.theta2)/.2.) theta2)
    in
    aux n a b 0. (2. *. Float.pi)
```

2)
```
let n = 5 and a=1. and b=1. in
  let approx_pi = ((approx_perim_ellipse2 n a b)/. 2.) in
  Printf.printf "n=%d. Pi : %.20f.\n" n approx_pi
```

3)
```
for n=5 to 29 do
    let approx_pi2 = ((approx_perim_ellipse2 n a b)/. 2.) in
    Printf.printf "n=%d. Difference with pi : %.20f\n" n (Float.pi -.
    approx_pi2);
    flush stdout
  done
```

Au fur et à mesure que la valeur de n augmente, l'approximation de π devient de plus en plus proche à la valeur donnée par `Float.pi`, comme cela est attendu. La programmation fonctionnelle, surtout la récursivité change l'ordre de réalisation des calculs, équilibrant les additions entre des valeurs de taille homogène.