DM1 Bec-clemente Elio

Exercice 1

1.

La dérivée de f_1 vaut :

$$f_1' : x \mapsto a_i \times i \times x^{i-1}$$

La dérivée de f_2 vaut :

$$f_2' : x \mapsto \sum_{i=0}^{n} a_i \times i \times x^{i-1}$$

2.

```
let rec f x coefs =
        match coefs with
                | [] -> 0.
                | t::s -> t +. (f x s) *. x
;;
```

3.

On initialise la fonction récursive aux qui calcule les coefficients de la dérivée d'un monôme.

```
let rec aux coefs n =
        match coefs with
                | [] -> []
                | t::s -> [t *. float_of_int(n-(List.length coefs))] @ aux
s n
;;

let derive coef =
        let n = List.length coef in
                match coef with
                        | [] -> []
                        | t::s -> aux s n;;
;;
```

4.

```
let descGradient coefs x0 pas n =
        let derivee = derive coefs in
                let x = ref x0 in
                        for i = 0 to (n-1) do
                                x := (!x) -. pas *. f (!x) (derivee);
                        done;
                        (!x, f !x coefs);;
```

5.

```
let newton coefs x0 n =
        let derivee = derive coefs in
                let x = ref x0 in
                        for i = 0 to n-1 do
                                x := !x -. ((f (!x) (coefs)) /. (f (!x)
(derivee)));
                        done;
                        !x;;
```

Exercice 2

1.

```
type position = Gauche | Centre | Droite ;;
```

2.

```
let rec deplacements_hanoi n posD posI posA =
        if n = 0 then
                []
        else
                (deplacements_hanoi (n-1) posD posA posI) @
                ([(posD, posA)] @
                (deplacements_hanoi (n-1) posI posD posA))
;;
```

3.

```
type etat = {mutable gauche : int list;
                        mutable centre : int list;
                        mutable droite : int list}
;;
```

4.

On initialise 2 fonctions auxiliaires :

```
(*Fonction qui prend en paramètre un entier et renvoie une liste dont les éléments varient de 1 a len avec un pas de 1 lorsqu'on parcours la liste de gauche a droite*)

let rec createList len =
        if len = 0 then
                []
        else
                (createList (len-1)) @ ([len])
;;
```

```
(*Fonction qui prend en paramètre une liste et un entier et qui renvoie la valeur de l'élément de la liste a l'indice index.*)

let rec valeur liste index =
        match index with
        | 0 -> List.hd liste
        | _ -> valeur (List.tl liste) (index-1)
```

Pour pouvoir faire fonctionner la fonction finale :

```
(*Fonction finale*)
let etats_hannoi n posD posI posA =
        let etat = {gauche = []; centre = []; droite = []} in
                if posD = Gauche then
                        etat.gauche <- createList n
                else if posD = Centre then
                        etat.centre <- createList n
                else
                        etat.droite <- createList n;

        let deplacements = deplacements_hanoi n posD posI posA in
                let listeEtat = ref [{gauche = etat.gauche; centre = etat.centre; droite = etat.droite}] in
                        for i = 0 to ((List.length deplacements) -1) do
                                match valeur deplacements i with

| (Gauche, Centre) ->
listeEtat := !listeEtat @ [{gauche = List.tl etat.gauche; centre = [List.hd etat.gauche] @ etat.centre; droite = etat.droite}] ;
etat.centre <- [List.hd etat.gauche] @ etat.centre ;
etat.gauche <- List.tl etat.gauche
```

```
| (Gauche, Droite) ->
listeEtat := !listeEtat @ [{gauche = List.tl etat.gauche; centre =
etat.centre; droite = [List.hd etat.gauche] @ etat.droite}] ;
etat.droite <- [List.hd etat.gauche] @ etat.droite ;
etat.gauche <- List.tl etat.gauche

| (Centre, Gauche) ->
listeEtat := !listeEtat @ [{gauche = [List.hd etat.centre] @ etat.gauche;
centre = List.tl etat.centre; droite = etat.droite}] ;
etat.gauche <- [List.hd etat.centre] @ etat.gauche ;
etat.centre <- List.tl etat.centre

| (Centre, Droite) ->
listeEtat := !listeEtat @ [{gauche = etat.gauche; centre = List.tl
etat.centre; droite = [List.hd etat.centre] @ etat.droite}] ;
etat.droite <- [List.hd etat.centre] @ etat.droite ;
etat.centre <- List.tl etat.centre

| (Droite, Gauche) ->
listeEtat := !listeEtat @ [{gauche = [List.hd etat.droite] @ etat.gauche;
centre = etat.centre; droite = List.tl etat.droite}] ;
etat.gauche <- [List.hd etat.droite] @ etat.gauche ;
etat.droite <- List.tl etat.droite

| (Droite, Centre) ->
listeEtat := !listeEtat @ [{gauche = etat.gauche; centre = [List.hd
etat.droite] @ etat.centre; droite = List.tl etat.droite}] ;
etat.centre <- [List.hd etat.droite] @ etat.centre ;
etat.droite <- List.tl etat.droite

| _ -> listeEtat := !listeEtat @ [etat]

done;
!listeEtat;;
```