Différentiation algorithmique

30 avril 2022

Partie I

Question I.1.

```
let identite n =
  let m array.make_matrix n n 0.0 in
  for i = 0 to (n-1) do m.(i).(i) <- 1.0 done;
  m;;
```

Question I.2.

```
let scalaire u v =
  let ps = ref 0.0 in
  for i = 0 to (Array.length u -1) do
    ps := !ps +. u.(i) *. v.(i) done;
  !ps;;
```

Question I.3.

```
let mul_possible a b =
  Array.length a.(0) = Array.length b ;;
```

Question I.4.

```
let mul a b =
  let m = Array.length a in
  let n = Array.length b in
  let p = Array.length b.(0) in
  let c = Array.make_matrix m p 0.0 in
  for i = 0 to (m-1) do
    for j = 0 to (p-1) do
      for k =0 to (n-1) do
        c.(i).(j) <- c.(i).(j) +. a.(i).(k) *. b.(k).(j) done
    done
  done;
  c;;
```

Partie II

Question II.1. $B^1 \times B^2$ demande $k_0k_1k_2$ multiplications et donne une matrice appartenant à $\mathcal{M}_{k_0,k_2}(\mathbb{R})$; on en déduit que le calcul de $(B^1 \times B^2) \times B^3$ nécessite au total $k_0k_1k_2 + k_0k_2k_3 = k_0k_2(k_1 + k_3)$ multiplications.
De même $B^1 \times (B^2 \times B^3)$ nécessite $k_1k_3(k_0+k_2)$ multiplications. L'ordre d'évaluation du produit $B^1 \times B^2 \times B^3$ ne donne pas toujours le même nombre d'opérations.

Question II.2.

```
let rec muld liste =
  match liste with
  |[] -> failwith "La liste ne doit pas être vide"
  |[m] -> m
  |m::reste -> mul m (muld reste);;
```

Question II.3. le récursivité terminale permet d'inverser les opérations

```
let mulg liste =
  let aux_mul reste prod =
    mach reste with
    |[] -> prod
    |m::q -> aux_mul q (mul prod m) in
  match liste with
  |[] -> failwith "La liste ne doit pas être vide"
  |m::reste -> aux_mul reste m;;
```

Question II.4.

```
let min_mul k =
  let n = Array.length k in
  let rec pm i j =
    if i = j
    then 0
    else begin
      let p0 = ref ((pm i (j-1)) + k.(i-1)*k.(j-1)*k.(j)) in
      for r = i to (j-2) do
        let p = (pm i r)+(pm (r+1) j)+k.(i-1)*k.(r)*k.(j) in
        p0 := min !p0 p done;
      !p0 end in
  pm 1 (n-1);;
```

On compte, par exemple, le nombre d'appels récursifs pour le calcul de `pm i j`.
Ce nombre ne dépend que $h = j - i$, on le note $C(h)$.
On a $C(0) = 0$, $C(1) = 4$ (une affectation, une addition et 2 multiplications) et, pour $h \geqslant 2$
$C(h) = C(h-1) + 3 + \sum_{k=0}^{h-2} \big(C(k) + C(h-k-1) + 6\big) = 6h - 3 + 2\sum_{k=1}^{h-2} C(k) + C(h-1).$
On a donc $C(h+1) - C(h) = C(h) + C(h-1) + 6$ pour $h \geqslant 2$,
donc $C(h) = A \cdot (1+\sqrt{2})^h + B \cdot (1-\sqrt{2})^h + 3$ pour $h \geqslant 2$ avec $A > 0$ et $1 + \sqrt{2} > 2$.
La complexité est bien exponentielle.

Question II.5.

```
let min_mul k =
  let n = Array.length k in
  let p = Array.make_matrix n n 0 in
  for h = 1 to (n-2) do
    for i = 1 to (n-1-h) do
      let j = i + h in
      p.(i).(j) <- p.(i).(j-1) + k.(i-1)*k.(j-1)*k.(j);
      for r = i to (j-2) do
      let pp = p.(i).(r)+p.(r+1).(j)+k.(i-1)*k.(r)*k.(j) in
        p.(i).(j) <- min p.(i).(j) pp done done done;
  p.(1).(n-1);;
```

La complexité temporelle est en $\Theta(n^3)$, la complexité spatiale est en $\Theta(n^2)$

Question II.6. On a besoin de connaître les découpages qui permettent le produit optimum

```
let decoupage k =
  let n = Array.length k in
  let p = Array.make_matrix n n 0 in
  let dec = Array.make_matrix n n 0 in
  for h = 1 to (n-2) do
    for i = 1 to (n-1-h) do
      let j = i + h in
      p.(i).(j) <- p.(i).(j-1) + k.(i-1)*k.(j-1)*k.(j);
      dec.(i).(j) <- j
      for r = i to (j-2) do
      let pp = p.(i).(r)+p.(r+1).(j)+k.(i-1)*k.(r)*k.(j) in
        if pp < p.(i).(j)
        then begin
          p.(i).(j) <- pp;
          dec.(i).(j) <- r end done done done;
  p.(1).(n-1);;
```

On peut alors cacluler le produit de manière optimale en calculant récursivement les $B^{i,j}$.

```
let mul_opt m =
  let n = Array.length m in
  let k = Array.make (n+1) 0 in
  for i = 0  to (n-1) do k.(i) <- Array.length m.(i) done;
  k.(n) <- Array.length m.(n-1).(0);
  let dec = decoupage k in
  let rec prodB  i j =
    if i = j then m.(i-1)
    else let k = dec.(i).(j) in
         mul (prodB i (k-1)) (prodB k j) in
  prodB 1 n;;
```

Partie III

Question III.1. La fonction $\overrightarrow{\mathrm{D}} f$ est définie sur $\mathbb{R}^n \times \mathcal{F}(\mathbb{R}^p, \mathbb{R}^n)$ et à valeurs dans $\mathbb{R}^m \times \mathcal{F}(\mathbb{R}^p, \mathbb{R}^m)$ où $\mathcal{F}(A, B)$ désigne l'ensembles des fonctions de A vers B.

$$\begin{aligned}\left(\overrightarrow{\mathrm{D}} g \circ \overrightarrow{\mathrm{D}} f\right)(x,h) &= \overrightarrow{\mathrm{D}} g\left(f(x), (\mathrm{D}_{\mathbf{x}} f) \circ h\right) = \left(g(f(x)), (\mathrm{D}_{f(\mathbf{x})} g) \circ (\mathrm{D}_{\mathbf{x}} f) \circ h\right) \\ &= \left((g \circ f)(x), (\mathrm{D}_{\mathbf{x}}(g \circ f)) \circ h\right) = \overrightarrow{\mathrm{D}}(g \circ f)(x,h)\end{aligned}$$

Ainsi $\overrightarrow{\mathrm{D}} g \circ \overrightarrow{\mathrm{D}} f = \overrightarrow{\mathrm{D}}(g \circ f)$.

Question III.2. On a $(f^1(\mathbf{x}), \mathrm{D}_{\mathbf{x}} f^1) = \overrightarrow{\mathrm{D}}((x, \mathrm{Id}_{\mathbb{R}_{k_0}})$ donc, d'après la question précédente,

$$\begin{aligned}\overrightarrow{\mathrm{D}} f^2\left(f^1(\mathbf{x}), \mathrm{D}_{\mathbf{x}} f^1\right) &= \left(\overrightarrow{\mathrm{D}} f^2 \circ \overrightarrow{\mathrm{D}} f^1\right)(x, \mathrm{Id}_{\mathbb{R}_{k_0}}) = \overrightarrow{\mathrm{D}}(f^2 \circ f^1)(x, \mathrm{Id}_{\mathbb{R}_{k_0}}) \\ &= \left((f^2 \circ f^1)(x), \mathrm{D}_{\mathbf{x}}(f^2 \circ f^1) \circ \mathrm{Id}_{\mathbb{R}_{k_0}}\right) = \left((f^2 \circ f^1)(\mathbf{x}), \mathrm{D}_{\mathbf{x}}(f^2 \circ f^1)\right)\end{aligned}$$

Question III.3. La question ci-dessus nous incite à calculer les images des points en plus de la jacobienne. On initie par $(x, \mathrm{Id}_{\mathbb{R}_{k_0}})$.

```
let forward x foncs diff =
  let rec aux_fwd fns  =
    match fns with
    |[] -> (x, identite (Array.length x))
    |f::reste -> let y, jac = aux_fwd reste in
                 (f y, mul (diff f y) jac) in
  let _, jac = aux_fwd foncs in jac;;
```

On peut économiser un calcul de produit si on ajoute le cas d'une liste réduite à un élément

```
let forward x foncs diff =
  let rec aux_fwd fns  =
    match fns with
    |[] -> (x, identite (Array.length x))
    |[f] -> (f x, diff f x)
    |f::reste -> let y, jac = aux_fwd reste in
                 (f y, mul (diff f y) jac) in
  let _, jac = aux_fwd foncs in jac;;
```

Avec la question **III.2.**, on prouve qu'à chaque étape `aux_fwd [fi; ...; fn]` calcule bien $\overrightarrow{\mathrm{D}}(f_i \circ \cdots f_n)(x, \mathrm{Id}_{\mathbb{R}_{k_0}})$ donc la seconde composante du résultat final est bien $\mathrm{D}_{\mathbf{x}}(f_1 \circ \cdots f_n) \circ \mathrm{Id}_{\mathbb{R}_{k_0}}$, c'est la jacobienne demandée.

Question III.4. Pour chaque fonction, de $\mathbb{R}^{k_{i-1}}$ vers $\mathbb{R}^{k_i}$, `f y` a une complexité en $\Omega(k_{i-1})$, `diff f y` a une complexité en $\Theta(k_{i-1} \cdot k_i)$ et la jacobienne reçue est de taille $k_{i-1} \times k_0$ donc la multiplication a une complexité en $\Theta(k_0 \cdot k_{i-1} \cdot k_i)$. La complexité est donc en $\Theta\left(k_0 \cdot \sum_{i=1}^{n} k_{i-1} \cdot k_i\right)$.

Pour $k_i = 2^i$, on aboutit à $\sum_{i=1}^{n} 2^{2i-1} = \frac{2(4^n - 1)}{3}$ d'où une complexité en $\Theta(4^n)$.

Pour $k_i = k$, on aboutit à $k \cdot \sum_{i=1}^{n} k^2 = nk^3$ d'où une complexité en $\Theta(n)$.

Question III.5. On veut la multiplication des jacobiennes via un parenthésage à gauche. On va donc construire la liste des jacobiennes et utiliser `mulg`.

```
let backward x foncs diff =
  let rec aux_bwd fns =
    match fns with
    |[] -> x, []
    |f::reste -> let y, jac = aux_bwd reste in
                 (f y, (diff f y) :: jac) in
  let _, jac = aux_bwd foncs in mulg jac;;
```

Question III.6. Dans le cas d'un parenthésage à gauche le nombre de multiplications dans le produit des jacobiennes on multiplie des matrices de tailles respectives $k_n \times k_{n-1}$, $k_{n-1} \times k_{n-2}$, ..., $k_2 \times k_1$ puis $k_1 \times k_0$. Les produits successifs sont alors des produits de matrices de tailles $k_n \times k_{n-1}$ et $k_{n-1} \times k_{n-2}$ puis $k_n \times k_{n-2}$ et $k_{n-2} \times k_{n-3}$ jusqu'à $k_n \times k_1$ et $k_1 \times k_0$. Le nombre de multiplication est donc proportionnelle à $\sum_{i=1}^{n-1} k_n \cdot k_i \cdot k_{i-1}$.

Pour $k_i = 2^i$, on aboutit à $2^n \sum_{i=1}^{n-1} 2^{2i-1} = \frac{2^{n+1}(4^{n-1}-1)}{3}$ d'où une complexité en $\Theta(2^{3n})$: le calcul est moins efficace que `forward`.

Pour $k_i = 2^{n-i}$ les nombres d'opérations sont intervertis et `backward` est plus efficace que `forward`.

Question III.7. Cette question semble indiquer que la solution de la question **III.5.** ci-dessus n'est pas celle qui est attendue. En effet, il suffit de convertir la liste des jacobiennes en tableau et remplacer `mul` par `mul_opt`.

Partie IV

Question IV.1.

Indice	Fonction	Valeur	Différentielle
0	x	$\frac{\pi}{4}$	1
1	y	1	1
2	$\cos(x)$	$\cos(\frac{\pi}{4}) = \frac{\sqrt{2}}{2}$	$(-\sin(\frac{\pi}{4})) \times 1 = \frac{-\sqrt{2}}{2}$
3	$(\cos(x))^2$	$(\frac{\sqrt{2}}{2})^2 = \frac{1}{2}$	$(2 \cdot \frac{\sqrt{2}}{2}) \times 1 = \sqrt{2}$
4	$(\cos(x))^3$	$(\frac{\sqrt{2}}{2})^3 = \frac{\sqrt{2}}{4}$	$(3 \cdot \frac{\sqrt{2}}{2}) \times 1 = \frac{3\sqrt{2}}{2}$
5	$\sqrt{2} \cdot y$	$\sqrt{2} \cdot 1 = \sqrt{2}$	$\sqrt{2} \times 1 = \sqrt{2}$
6	$(\cos(x))^3 \times (\sqrt{2} \cdot y)$	$\frac{\sqrt{2}}{4} + \sqrt{2} = \frac{5\sqrt{2}}{4}$	$\sqrt{2} \times \frac{3\sqrt{2}}{2} + \frac{\sqrt{2}}{4} \times \sqrt{2} = \frac{7}{2}$
7	$(\cos(x))^2 + (\cos(x))^3 \times (\sqrt{2} \cdot y)$	$\frac{1}{2} + \frac{5\sqrt{2}}{4}$	$1 \times \sqrt{2} + 1 \times \frac{7}{2} = \sqrt{2} + \frac{7}{2}$

Question IV.2.

```
let collecte propag ind =
  let n = Array.length ind in
  let valeur = Array.make n 0.0 in
  let dif = Array.make n 0.0 in
  for i = 0 to (n-1) do
    let v, d = propag.(ind.(i)) in
    valeur.(i) <- v;
    dif.(i) <- d done;
  valeur, dif;;
```

Question IV.3.

```
let propagation_avant g_av x d =
  let m = Array.length x in
  let n = Array.length g_av in
  let propag = Array.make n (0.0, 0.0) in
  for i = 0 to (m-1) do
    propag.(i) <- (x.(i), d.(i)) done;
  for i = m to (n-1) do
    let ind = g_av.(i).pred in
    let f, df = g_av.(i).fct in
    let valeur, dif = collecte propag ind in
    propag.(i) <- (f valeur, scalaire (df valeur) dif) done;
  let _, dif = propag.(n-1) in  dif;;
```

Question IV.4. Pour gérer le cas d'une fonction linéaires on pourrait créer un type somme

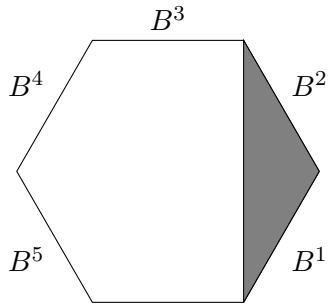
```
type sommet_avant =
   |Fonction of fonction_elementaire * int array};;
   |Lineaire of (float array -> float) * int array;;
```

On peut alors modifier la fonction de propagation

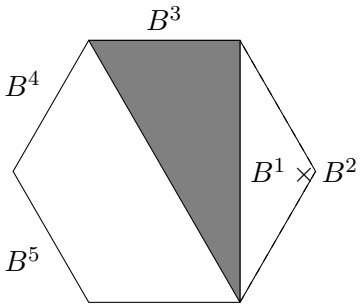
```
let propagation_avant g_av x d =
  let m = Array.length x in
  let n = Array.length g_av in
  let propag = Array.make n (0.0, 0.0) in
  for i = 0 to (m-1) do
    propag.(i) <- (x.(i), d.(i)) done;
  for i = m to (n-1) do
    match g_av.(i) with
    |Fonction ((f, df), ind)
      -> let val, dif = collecte propag ind in
         propag.(i) <- (f val, scalaire (df val) dif) done;
    |Lineaire (f, ind)
      -> let val, dif = collecte propag ind in
         propag.(i) <- (f val, f dif) done;
  let _, dif = propag.(n-1) in  dif;;
```

Partie V

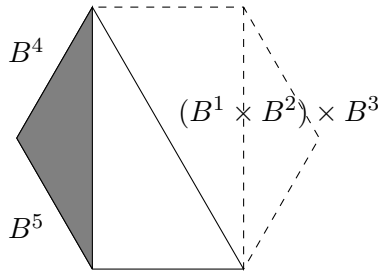
Question V.1. Pour un produit de n matrices matrices de flottants $(B^i)_{1\leqslant i\leqslant n}$ telles que $B^i \in \mathcal{M}_{k_{i-1},k_i}(\mathbb{R})$, on considère le polygone régulier à $n+1$ sommets $(S^i)_{0\leqslant i\leqslant n}$; à chaque sommet S_i est associé le poids k_i. À chaque arête (S_i, S_j) avec $i < j$ on peut associer le produit de matrice $B^{i+1} \times \cdots \times B^j$; en particulier B_i est associée à (S_{i-1}, S_i).
Un calcul du produit des matrices revient à calculer $n-1$ fois un produit de deux matrices de la forme $B^{i+1} \times \cdots \times B^r$ et $B^{r+1} \times \cdots \times B^j$. Un tel produit est associé au triangle (S_i, S_r, S_k). Le coût de ce produit en produits de flottants est $k_i \cdot k_r \cdot k_j$ qui est le coût du triangle.
Ainsi calculer un produit de matrice revient à ôter un triangle au polygone pour aboutit à la fin au seul segment (S_0, S_n), on a défini ainsi une triangulation.
Par exemple le produit $\big((B^1 \times B^2) \times B^3\big) \times (B^4 \times B^5)$ donne la triangulation



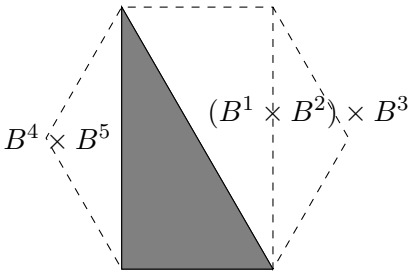
$B^1 \times B^2$



$(B^1 \times B^2) \times B^3$



$B^4 \times B^5$



$\big((B^1 \times B^2) \times B^3\big) \times (B^4 \times B^5)$

Inversement, si on part d'une triangulation du polygone défini ci-dessus, elle comporte p triangles qui sont bordés par $3p$ segments. Il y a $n+1$ segments qui bordent le polynômes et $p-1$ segments qui forment le découpage, chacun d'eux borde 2 triangles.
Ainsi on a $3p = n + 1 + 2(p-1)$ d'où $p = n - 1$.

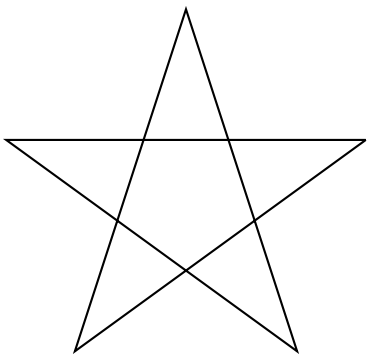
Les n arêtes du polygone distinctes de (S_0, S_n) bordent chacune un des $n-1$ triangles donc au moins un triangle est bordé par deux arêtes qui seront alors consécutives. Ce triangle (S_{i-1}, S_i, S_{i+1}) est associé à un produit $B^i \times B^{i+1}$.
On peut alors continuer en enlevant le sommet S_i ; la triangulation se restreint à une triangulation du nouveau polygone et on construit alors, pas à pas, une suite de produit qui aboutit au produit de toutes les matrices.
On a ainsi une bijection entre les produits possibles et les triangulations et le coût de la triangulation est la somme des coûts des triangles donc la somme des nombres de produits de flottants : la coût minimum d'une triangulation est bien le nombre minimal de produits de flottants dans le produits des matrices.

Question V.2. Avec la construction précédente, les matrices B^1 à B^{n-1} définissent un polygone à n sommets de poids k_0, k_1, …, k_{n-1} et les B^n, B^1 à B^{n-2} définissent un polygone à n sommets de poids $kn-1$, k_0, k_1, …, k_{n-2}.
Les deux polygone ont donc la même suite de poids en faisant subir une rotation aux sommets donc le coût minimum d'une triangulation est le même ; on en déduit que les deux produits de matrices ont le même nombre minimal de multiplications de flottants nécessaires à leur calcul.

Question V.3. On admet qu'un polygone simple $(S_1, S_2, \ldots, S_n)$ est convexe si et seulement si les angles entre deux droites portant deux arêtes successives ont le même signe, c'est à dire si et seulement si le déterminant de $(\overrightarrow{S_{i-1}S_i}, \overrightarrow{S_iS_{i+1}})$ est de signe constant

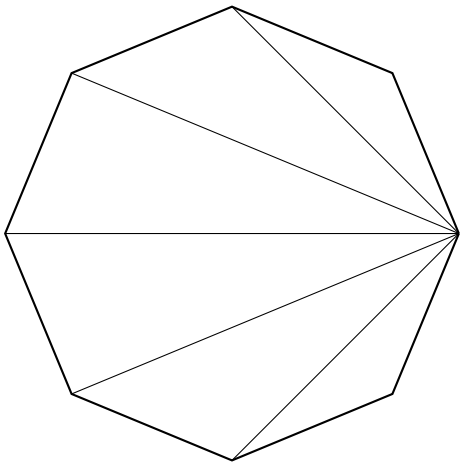
```
let det (a,b) (c, d) (e, f) =
  (c -. a) *. (f -. d) -. (d -. b) *. (e -.c);;

let est_convexe poly =
  let n = Array.length poly in
  let s0 = det poly.(n-2) poly.(n-1) poly.(0) in
  let s1 = det poly.(n-1) poly.(0) poly.(1) in
  let cvx = ref s0 *. s1 > 0 in
  let i = ref 0 in
  while cvx && !i < n-2 do
    if s0 *. (det poly.(!i) poly.(!i+1) poly.(!i+2) < 0
    then cvx := false;
    incr i done;
  !cvx;;
```

La propriété n'est pas valide pour un polygone non simple.



Question V.4. Cette question semble indiquer que ma réponse à la question **V.1.** n'est pas optimale car elle contient la réponse à la question **V.4.** On y a montrer qu'une triangulation d'un polygone à n sommets comportait $n-2$ triangles donc $n-3$ segment supplémentaires.

Question V.5. Le plus simple est une triangulation en éventail : on relie un sommet donné avec tous les sommets qui ne sont pas ses voisins.



```
let triangule_convexe poly =
  let n = Array.length poly in
  let rec aux k =
    if k = 1
    then []
    else (0, k) :: (aux (k-1)) in
  aux (n-2);;
```

Question V.6.

```
let separe_polygone  poly =
  let n = Array.length poly in
  let x0, y0 = poly.(0) in
  let y_min = ref y0 in
  let y_max = ref y0 in
  let i_min = ref 0 in
  let i_max = ref 0 in
  for i = 1 to (n-1) do
    let x, y = poly.(i) in
    if y < !y_min then (y_min := y; i_min := i);
    if y > !y_max then (y_max := y; i_max := i) done;
  let rec decouper a b i =
    if i = n
    then [], []
    else let l1, l2 = decouper a b (i+1) in
      if i < a || i >= b
      then (i :: l1), l2
      else l1, (i::l2) in
  if !i_min < !i_max
  then decouper !i_min !i_max 0
  else decouper !i_max !i_min 0;;
```

Question V.7. Il s'agit d'un algorithme donné initialement par
Michael R. Garey, David S. Johnson, Franco P. Preparata, Robert E. Tarjan,
Triangulating a simple polygon, Information Processing Letters,
1978, Volume 7, Issue 4, Pages 175-179,
https ://doi.org/10.1016/0020-0190(78)90062-5.
L'exposer dépasse mes compétences