ECOLE POLYTECHNIQUE
ECOLES NORMALES SUPERIEURES

CONCOURS D'ADMISSION 2022

MARDI 26 AVRIL 2022
14h00 - 18h00

FILIERE MP - Epreuve n° 4

INFORMATIQUE A (XULSR)

Durée : 4 heures

L'utilisation des calculatrices n'est pas autorisée pour cette épreuve

Cette composition ne concerne qu'une partie des candidats de la filière MP, les autres candidats effectuant simultanément la composition de Physique et Sciences de l'Ingénieur.
Pour la filière MP, il y a donc deux enveloppes de Sujets pour cette séance.

Différentiation algorithmique

Le sujet comporte 12 pages, numérotées de 1 à 12.

Début de l'épreuve.

Dans ce sujet, on s'intéresse au problème de la différentiation algorithmique. Il s'agit de calculer efficacement des différentielles d'expressions mathématiques vues comme des composées de fonctions élémentaires. C'est l'une des pierres fondatrices du calcul formel, ainsi que de la descente de gradient dans les réseaux de neurones.

Ce sujet est constitué de cinq parties. La première partie est formée de préliminaires utiles dans le reste du sujet, mais les fonctions qui y sont définies peuvent être admises pour traiter les parties suivantes. La troisième partie est largement indépendante de la deuxième, à l'exception de la question III.7. La quatrième partie nécessite d'avoir compris les enjeux de la troisième partie. Enfin, la cinquième et dernière partie est nettement indépendante du reste du sujet, seule les questions V.1 et V.2 font référence aux sujets abordés avant.

Notations et définitions

— On note $(\mathbf{e}_1, \ldots, \mathbf{e}_n)$ la *base canonique* de $\mathbb{R}^n$ (pour n un entier strictement positif), de sorte que pour tout $\mathbf{x} = (x_1, \ldots, x_n) \in \mathbb{R}^n$, $\mathbf{x} = \sum_{i=1}^n x_i \mathbf{e}_i$.

— Pour une fonction $f : \mathbb{R}^n \to \mathbb{R}^m$ (avec $n, m \in \mathbb{N}^*$), on note $f_i : \mathbb{R}^n \to \mathbb{R}$ (pour $1 \leqslant i \leqslant m$) la *projection* de f sur la coordonnée i : $f_i : \mathbf{x} \mapsto f(\mathbf{x}) \cdot \mathbf{e}_i$.

— Pour une fonction $f : \mathbb{R}^n \to \mathbb{R}^m$ (avec $n, m \in \mathbb{N}^*$), on note $\mathrm{D}_{\mathbf{x}} f(\mathbf{v})$ la *différentielle* de la fonction f en un point $\mathbf{x}$ évaluée en un vecteur $\mathbf{v}$ (cette notation présuppose que f est différentiable en $\mathbf{x}$). On rappelle que la fonction $\mathrm{D}_{\mathbf{x}} f$ est *linéaire* en son argument $\mathbf{v}$.

— On rappelle également la *règle de la chaîne* : pour $n, m, p \in \mathbb{N}^*$, $f : \mathbb{R}^n \to \mathbb{R}^m$, $g : \mathbb{R}^m \to \mathbb{R}^p$ et $\mathbf{x} \in \mathbb{R}^n$, on a :

$$\mathrm{D}_{\mathbf{x}}(g \circ f) = \mathrm{D}_{f(\mathbf{x})} g \circ \mathrm{D}_{\mathbf{x}} f,$$

où $\circ$ représente la composition de fonctions.

— On note également $\frac{\partial f}{\partial x_i} = \mathrm{D}_{\mathbf{x}} f(\mathbf{e}_i)$ la dérivée de f en $\mathbf{x}$ suivant le vecteur $\mathbf{e}_i$ de la base canonique ; pour $\mathbf{v} = (v_1, \ldots, v_n)$ et par linéarité de la différentielle, on a donc :

$$\mathrm{D}_{\mathbf{x}} f(\mathbf{v}) = \sum_{i=1}^{n} v_i \frac{\partial f}{\partial x_i}.$$

— On note $\mathcal{M}_{m,n}(\mathbb{R})$ l'ensemble des matrices à m lignes, n colonnes, et coefficients dans $\mathbb{R}$. On supposera toujours $m \geqslant 1$, $n \geqslant 1$.

— Pour $n, m \in \mathbb{N}^*$, la *matrice jacobienne* d'une fonction $f : \mathbb{R}^n \to \mathbb{R}^m$ en un point $\mathbf{x} \in \mathbb{R}^n$, notée $\mathrm{J}_f(\mathbf{x})$, est la matrice de $\mathcal{M}_{m,n}(\mathbb{R})$ définie par :

$$\mathrm{J}_f(\mathbf{x}) = \begin{pmatrix} \frac{\partial f}{\partial x_1} & \cdots & \frac{\partial f}{\partial x_n} \end{pmatrix} = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \cdots & & \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{pmatrix}.$$

— En appliquant la règle de la chaîne, on observe que pour deux fonctions $f : \mathbb{R}^n \to \mathbb{R}^m$ et $g : \mathbb{R}^m \to \mathbb{R}^p$ et pour un point $\mathbf{x} \in \mathbb{R}^n$, on a : $\mathrm{J}_{g \circ f}(\mathbf{x}) = \mathrm{J}_g(f(\mathbf{x})) \times \mathrm{J}_f(\mathbf{x})$.

Complexité

Par complexité en temps d'un algorithme $\mathcal{A}$, on entend le nombre d'opérations élémentaires (comparaison, addition, soustraction, multiplication, division, affectation, test, etc.) nécessaires à l'exécution de $\mathcal{A}$ dans le cas le pire.

Par complexité en mémoire d'un algorithme $\mathcal{A}$, on entend l'espace mémoire total utilisé par l'exécution de $\mathcal{A}$ dans le cas le pire, en supposant que l'espace mémoire occupé pour stocker une valeur d'un type de base (entier, booléen, flottant, caractère) est une constante.

Lorsque la complexité (en temps ou en espace) dépend d'un paramètre κ, on dit que $\mathcal{A}$ a une complexité (en temps ou en espace) en $O(f(\kappa))$ s'il existe une constante $C > 0$ telle que, pour toutes les valeurs de κ suffisamment grandes (c'est-à-dire plus grandes qu'un certain seuil), pour toute instance du problème de paramètre κ, la complexité (en temps ou en espace) est au plus $C \cdot f(\kappa)$; on dit que $\mathcal{A}$ a une complexité (en temps ou en espace) en $\Omega(f(\kappa))$ s'il existe une constante $C > 0$ telle que, pour toutes les valeurs de κ suffisamment grandes, pour toute instance du problème de paramètre κ, la complexité (en temps ou en espace) est au moins $C \cdot f(\kappa)$. Enfin, on a une complexité en $\Theta(f(\kappa))$ quand elle est à la fois en $O(f(\kappa))$ et en $\Omega(f(\kappa))$.

Quand l'on demande la complexité d'un algorithme, il s'agit de trouver une expression en $\Theta(\cdot)$ la plus simple possible. On dit qu'un algorithme est linéaire en κ si sa complexité en temps est en $\Theta(\kappa)$.

Rappels OCaml

- On rappelle les opérations de base sur les nombres en virgule flottante (ou *flottants*), que l'on utilisera pour représenter des nombres réels dans OCaml :
  - `1.0` et `0.0` (ou `1.` et `0.`) représentent les nombres 1 et 0 respectivement.
  - L'addition entre deux flottants `a` et `b` s'écrit `a +. b`.
  - La multiplication entre deux flottants `a` et `b` s'écrit `a *. b`.
- On rappelle quelques opérations de base sur les tableaux :
  - **`Array.length: 'a array -> int`** donne la longueur d'un tableau.
  - **`Array.make: int -> 'a -> 'a array`** permet de créer un tableau : **`Array.make m r`** crée un tableau de taille m dont toutes les cases sont initialisées à r.
  - **`Array.make_matrix: int -> int -> 'a -> 'a array array`** permet de créer une matrice : **`Array.make_matrix m n r`** crée une matrice de taille $m \times n$ dont toutes les cases sont initialisées à r.
- Enfin, une opération de base sur les listes :
  - **`List.rev: 'a list -> 'a list`** renvoie le miroir de la liste passée en argument (cette fonction a une complexité linéaire en la taille de la liste).

Dans l'ensemble du sujet, il sera fréquemment question de matrices de flottants. Pour simplifier l'écriture du type des fonctions manipulant de telles matrices, nous définissons le type `matrix` :

```
type matrix = float array array;;
```

Partie I

Question I.1. Donner une fonction OCaml

```
identite: int -> matrix
```

prenant en entrée un entier $n \in \mathbb{N}^*$ et renvoyant la matrice identité de $\mathcal{M}_{n,n}(\mathbb{R})$.

Question I.2. Donner une fonction OCaml

```
scalaire: float array -> float array -> float
```

prenant en entrée deux vecteurs u et v et renvoyant le produit scalaire $u \cdot v$.

Question I.3. Donner une fonction OCaml

```
mul_possible: matrix -> matrix -> bool
```

prenant en entrée deux matrices A et B et renvoyant `true` si et seulement si les dimensions de A et B sont telles que $A \times B$ est bien définie.

Question I.4. Donner une fonction OCaml

```
mul: matrix -> matrix -> matrix
```

qui calcule le produit $A \times B$ des deux matrices A et B de flottants passées en argument. Si A est de dimension $m \times n$ et B de dimension $n \times p$, on impose que la fonction `mul` effectue $m \times n \times p$ multiplications de flottants.

Dans l'ensemble du sujet, on utilisera la fonction `mul` à chaque fois qu'il s'agit de calculer le produit de deux matrices, sans chercher à rendre ce calcul plus efficace.

Partie II

Dans cette partie, on s'intéresse à la multiplication d'une suite finie de matrices réelles $(B^i)_{1\leqslant i\leqslant n}$. Soit $k_0, \ldots, k_n$ la suite finie d'entiers tels que $B^i \in \mathcal{M}_{k_{i-1},k_i}(\mathbb{R})$. On cherche à écrire un programme effectuant le parenthésage optimal sur les B^i pour un calcul efficace de leur multiplication $B^1 \times \cdots \times B^n$. Pour chaque $1 \leqslant i \leqslant j \leqslant n$, on note $B^{i,j}$ le produit de matrices $B^i \times \cdots \times B^j$, et $p^{i,j}$ le nombre minimum de multiplications de flottants nécessaires pour calculer $B^{i,j}$. On rappelle que l'on réutilisera la fonction `mul` de la partie précédente implémentant la multiplication de deux matrices, que l'on ne cherchera pas à améliorer.

Question II.1. On suppose (dans cette question uniquement) $n = 3$. Comparer le nombre de multiplications de flottants effectuées en évaluant $(B^1 \times B^2) \times B^3$ avec `mul` et en évaluant $B^1 \times (B^2 \times B^3)$ avec `mul`. Que faut-il en conclure sur l'ordre d'évaluation du produit $B^1 \times B^2 \times B^3$?

Question II.2. Donner une fonction OCaml

```
muld: matrix list -> matrix
```

qui prend en entrée une liste non vide de matrices $(B^1, \ldots, B^n)$ et renvoie leur multiplication (en utilisant `mul`) via un parenthésage à droite : $B^1 \times (B^2 \times (\ldots))$.

Question II.3. Donner une fonction OCaml

```
mulg: matrix list -> matrix
```

qui prend en entrée une liste non vide de matrices $(B^1, \ldots, B^n)$ et renvoie leur multiplication (en utilisant `mul`) via un parenthésage à gauche : $((\ldots) \times B^{n-1}) \times B^n$.

Question II.4. Donner une fonction OCaml `min_mul: int array -> int`, qui prend en entrée le tableau $k = [k_0, \ldots, k_n]$ des tailles de chacune des matrices, et renvoie $p^{1,n}$ en faisant des appels récursifs pour calculer les $p^{i,j}$, sans stockage d'information supplémentaire. Montrer que si, pour tout i, $k_i \geqslant 2$, la complexité en temps de `min_mul` est au moins exponentielle en n, c'est-à-dire en $\Omega(2^n)$.

Question II.5. Donner une fonction OCaml `min_mul_opt: int array -> int` qui calcule le même nombre, mais où une matrice auxiliaire de taille $n \times n$, initialement nulle, permet de stocker les valeurs de $p^{i,j}$ déjà calculées. Quelle est la complexité en temps de `min_mul_opt` en fonction de n ? En espace ?

Question II.6. Donner une fonction OCaml `mul_opt: matrix array -> matrix` qui prend en entrée le tableau des matrices $[B^1, \ldots, B^n]$ et calcule $B^{1,n}$ en $p^{1,n}$ multiplications de flottants.

Partie III

On se propose d'étudier deux manières de calculer la différentiation de la composition d'une liste de fonctions. Soient $f^1, f^2, \ldots, f^n$ des fonctions de type $f^i : \mathbb{R}^{k_{i-1}} \to \mathbb{R}^{k_i}$ pour $k_0, \ldots, k_n \in \mathbb{N}^*$. On note $f = f^n \circ \cdots \circ f^1$. Il s'agit d'optimiser le calcul successif de la valeur de f^i et de celle de sa différentielle en un point. On note $J^i_{\mathbf{x}} \in \mathcal{M}_{k_i,k_{i-1}}(\mathbb{R})$ la matrice jacobienne de f^i en $\mathbf{x} \in \mathbb{R}^{k_{i-1}}$. Pour tout i, on note J^i la fonction

$$J^i : \begin{cases} \mathbb{R}^{k_{i-1}} \to \mathcal{M}_{k_i,k_{i-1}}(\mathbb{R}) \\ \quad \mathbf{x} \mapsto J^i_{\mathbf{x}}. \end{cases}$$

Question III.1. Pour une fonction $f : \mathbb{R}^n \to \mathbb{R}^m$ avec $n, m \in \mathbb{N}^*$, on note $\overrightarrow{\mathrm{D}} f$ la fonction :

$$(\mathbf{x}, h) \mapsto (f(\mathbf{x}), (\mathrm{D}_{\mathbf{x}} f) \circ h).$$

Si on impose que le domaine de la fonction h est $\mathbb{R}^p$ pour un certain $p \in \mathbb{N}^*$, quels sont l'ensemble de départ (domaine) et l'ensemble d'arrivée (codomaine) de la fonction $\overrightarrow{\mathrm{D}} f$? Montrer que l'opérateur $\overrightarrow{\mathrm{D}}$ vérifie :

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

Question III.2. Pour deux fonctions $f^1 : \mathbb{R}^{k_0} \to \mathbb{R}^{k_1}$ et $f^2 : \mathbb{R}^{k_1} \to \mathbb{R}^{k_2}$ et un point $\mathbf{x} \in \mathbb{R}^{k_0}$, montrer que $\overrightarrow{\mathrm{D}} f^2(f^1(\mathbf{x}), \mathrm{D}_{\mathbf{x}} f^1) = ((f^2 \circ f^1)(\mathbf{x}), \mathrm{D}_{\mathbf{x}}(f^2 \circ f^1))$.

Question III.3. Donner une fonction OCaml `forward` de type :

```
float array ->
(float array -> float array) list ->
((float array -> float array) -> float array -> matrix) ->
matrix
```

qui prend en entrée un point $\mathbf{x} \in \mathbb{R}^{k_0}$, une liste de fonctions $f^1, \ldots, f^n$ de type $f^i : \mathbb{R}^{k_{i-1}} \to \mathbb{R}^{k_i}$, et une fonction

```
diff: (float array -> float array) -> float array -> matrix
```

renvoyant la jacobienne d'une fonction en un point passé en argument, et calcule la matrice représentant $\mathrm{D}_{\mathbf{x}}(f^n \circ \cdots \circ f^1)$ via un parenthésage à droite de la forme $\mathrm{D}_{\mathbf{x}}(f^n \circ (f^{n-1} \circ \ldots))$. On s'appuiera sur la définition de $\overrightarrow{\mathrm{D}}$ et sur la propriété établie à l'aide de la question précédente. On pourra s'aider d'une fonction récursive auxiliaire. Prouver la correction de cette fonction.

Question III.4. Quelle est la complexité en temps de cette fonction dans le cas où $k_i = 2^i$? Dans le cas où pour tout i, $k_i = k$ avec $k \geqslant 2$ une constante quelconque ? On supposera que les appels à la fonction `diff` se font en temps proportionnel à la taille de la matrice jacobienne renvoyée par cette fonction.

Question III.5. Donner une fonction `backward` de type identique à `forward` qui prend en entrée un point $\mathbf{x} \in \mathbb{R}^{k_0}$, une liste de fonctions $f^1, \ldots, f^n$ de type $f^i : \mathbb{R}^{k_{i-1}} \to \mathbb{R}^{k_i}$, et une fonction

```
diff: (float array -> float array) -> float array -> matrix
```

renvoyant la jacobienne d'une fonction en un point passé en argument, et calcule la matrice représentant $\mathrm{D}_{\mathbf{x}}(f^n \circ \cdots \circ f^1)$ via un parenthésage à gauche de la forme $\mathrm{D}_{\mathbf{x}}((\cdots \circ f^2) \circ f^1)$.

Question III.6. Pour une liste de fonctions de longueur n arbitraire, donner un exemple de cas où `backward` est plus efficace que `forward`, ainsi que de cas où `forward` est plus efficace que `backward`.

Question III.7. Sans écrire explicitement de fonction, expliquer comment adapter les techniques des questions de la partie II pour écrire une fonction `diff_opt` qui prend en entrée $\mathbf{x} \in \mathbb{R}^{k_0}$, une liste de n fonctions $f^1, \ldots, f^n$ de type $f^i : \mathbb{R}^{k_{i-1}} \to \mathbb{R}^{k_i}$, et un tableau de n fonctions $[J^1, \ldots, J^n]$ et calcule avec un minimum de multiplications de flottants la matrice jacobienne de $f = f^n \circ \cdots \circ f^1$ en $\mathbf{x}$.

Partie IV

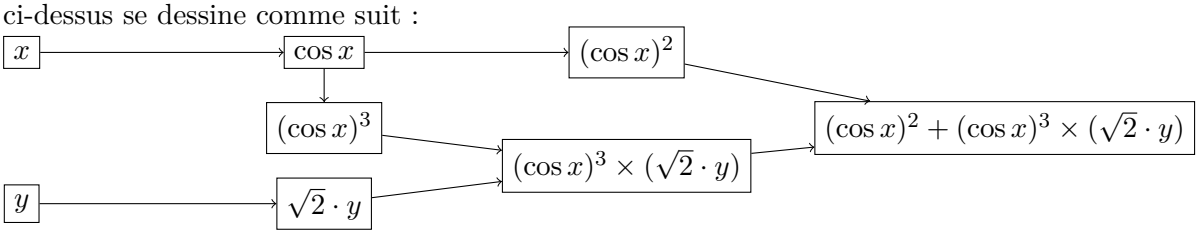
La représentation d'une suite d'instructions mathématiques comme une composition de fonctions est assez peu concise. En particulier, elle ne tient pas compte des dépendances *linéaires* entre les instructions. Considérons par exemple la fonction :

$$f : \begin{cases} \mathbb{R}^2 \to \mathbb{R} \\ (x, y) \mapsto (\cos x)^2 + (\cos x)^3 \times (\sqrt{2} \cdot y) \end{cases}$$

On peut l'écrire comme une composition $f = f^4 \circ f^3 \circ f^2 \circ f^1$ avec :

$$\begin{array}{lrl} f^1 : & (x, y) & \mapsto (\cos x, \sqrt{2} \cdot y) \\ f^2 : & (x, y) & \mapsto (x^2, x^3, y) \\ f^3 : & (x, y, z) & \mapsto (x, y \times z) \\ f^4 : & (x, y) & \mapsto x + y \end{array}$$

Propager la différentielle des fonctions f^i pour $1 \leqslant i \leqslant 4$ est une perte de temps. En effet, on manipule des fonctions qui ne font pas usage de certaines de leur variables, et on différencie des fonctions potentiellement linéaires, comme la fonction f^4. On peut représenter plus finement les suites d'instructions mathématiques comme des graphes acycliques. Par exemple, la fonction f ci-dessus se dessine comme suit :



On décrit un tel graphe comme un tableau T de sommets. Chaque sommet u contient deux informations : la fonction élémentaire qu'il représente et le tableau de ses prédécesseurs (les sommets v tels qu'il y a une arête de v à u). Comme le graphe est acyclique, on peut faire l'hypothèse que s'il y a une arête de v à u, l'indice de v dans T est strictement inférieur à l'indice de u dans T. On suppose par ailleurs que l'on ne manipule que des fonctions élémentaires à valeurs *réelles* $\mathbb{R}^n \to \mathbb{R}$. Ces fonctions élémentaires sont alors représentées par une paire de type `(float array -> float) * (float array -> float array)`, où le deuxième élément représente la différentielle du premier élément (comme la fonction est à valeurs réelles, sa différentielle en un point est représentée directement par un vecteur et non par une matrice). On définit donc les types OCaml suivants :

```
type fonction_elementaire =
  (float array -> float) * (float array -> float array);;

type sommet_avant = {fct : fonction_elementaire; pred : int array};;

type graphe_avant = sommet_avant array;;
```

Lors du calcul par propagation avant de la différentielle d'une fonction, on parcourt le graphe de cette fonction *de la gauche vers la droite*, c'est-à-dire des sommets correspondant aux variables au sommet correspondant à la fonction complète, en traitant les prédécesseurs

d'un sommet avant celui-ci. On stocke dans un *tableau de propagation avant*, au fur et à mesure, les valeurs de chaque fonction élémentaire et de sa différentielle en les points pertinents. Pour les sommets d'un graphe de fonction correspondant aux variables, le contenu du champ `fct` n'importe pas, car ils n'ont pas de prédécesseurs.

On peut par exemple coder le graphe de la fonction exemple f comme suit :

```
let mafonction =
  let id t = t.(0) in
  let did t = [|1.|] in
  let cube t = (t.(0)) ** 3. in
  let dcube t = [|3. *. ((t.(0)) ** 2.)|] in
  let scaler t = sqrt 2. *. (t.(0)) in
  let dscaler t = [|sqrt 2.|] in
  let costab t = (cos (t.(0))) in
  let dcostab t = [| -1. *. sin (t.(0)) |] in
  let carre t = (t.(0)) ** 2. in
  let dcarre t = [|2. *. t.(0)|] in
  let mult t = t.(0) *. t.(1) in
  let dmult t = [|t.(1);t.(0)|] in
  let plus t = t.(0) +. t.(1) in
  let dplus t = [|1.;1.|] in
  let n0 : sommet_avant = {fct = (id,did); pred = [| |]} in
  let n2 : sommet_avant = {fct = (costab, dcostab); pred = [|0|]} in
  let n3 : sommet_avant = {fct = (carre , dcarre); pred = [|2|]} in
  let n4 : sommet_avant = {fct = (cube, dcube); pred = [|2|]} in
  let n5 : sommet_avant = {fct = (scaler, dscaler); pred = [|1|]} in
  let n6 : sommet_avant = {fct = (mult , dmult); pred = [|4;5|]} in
  let n7 : sommet_avant = {fct = (plus , dplus); pred = [|3;6|]} in
  [|n0;n0;n2;n3;n4;n5;n6;n7|];;
```

Question IV.1. On souhaite calculer la valeur de la différentielle de la fonction f en $(\frac{\pi}{4}, 1)$ évaluée en le point $(1, 1)$. Pour ce faire, calculer à la main l'ensemble des valeurs du *tableau de propagation avant* en détaillant le calcul. Les trois premières lignes du tableau sont :

Indice	Fonction	Valeur	Différentielle
0	x	$\frac{\pi}{4}$	1
1	y	1	1
2	$\cos x$	$\cos \frac{\pi}{4} = \frac{\sqrt{2}}{2}$	$(-\sin \frac{\pi}{4}) \times 1 = \frac{-\sqrt{2}}{2}$

Question IV.2. Donner une fonction intermédiaire OCaml

```
collecte : (float * float) array -> int array -> float array * float array
```

prenant en entrée un tableau de propagation avant A (c'est-à-dire un tableau de couples (valeur_fonction, valeur_différentielle)), un tableau I de n indices de sommets et renvoyant un couple formé :

— d'un vecteur des n valeurs de fonctions dans A correspondant aux sommets de I ;

— d'un vecteur des n valeurs de différentielles dans A correspondant aux sommets de I.

Pour l'exemple de la fonction f, les trois premiers éléments du tableau A sont de la forme :

```
[| (0.785398163397448279, 1.);
   (1., 1.);
   (0.707106781186547573, -0.707106781186547573) |]
```

Question IV.3. En utilisant la fonction `collecte`, écrire une fonction Ocaml

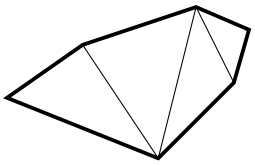
```
propagation_avant: graphe_avant -> float array -> float array -> float
```

prenant en entrée un graphe représentant une fonction f, un tableau représentant un point $\mathbf{x}$, un tableau représentant un vecteur $\mathbf{v}$, et qui calcule par propagation avant le flottant $\mathrm{D}_{\mathbf{x}}f(\mathbf{v})$.

Question IV.4. On voudrait traiter de manière particulière les fonctions linéaires, comme la dernière opération $(x, y) \mapsto x + y$ et la première opération $z \mapsto \sqrt{2} \cdot z$ dans l'exemple ci-dessus, car elles sont leur propres différentielles et peuvent donc être traitées plus simplement qu'une fonction arbitraire. Comment peut-on modifier la structure du graphe de la fonction dans ce sens ?

Partie V

Une *triangulation* d'un polygone correspond à la partition du polygone en triangles dont les sommets sont des sommets du polygone.



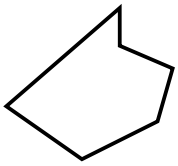
Question V.1. On considère des polygones convexes dont chaque sommet est étiqueté par un poids (entier naturel non nul). On définit le coût d'un triangle comme le produit des poids des trois sommets, et le coût d'une triangulation d'un polygone comme la somme des coûts de tous ses triangles. Montrer que le nombre minimum de multiplications de flottants nécessaires pour calculer la multiplication de $n \geqslant 2$ matrices (au sens de la partie II) est égal au coût minimum d'une triangulation d'un certain polygone convexe.

Question V.2. On reprend les notations de la partie II : soit une suite de matrices de flottants $(B^i)_{1\leqslant i\leqslant n}$ et $k_0, \ldots, k_n$ la suite d'entiers tels que $B^i \in \mathcal{M}_{k_{i-1},k_i}(\mathbb{R})$. On suppose de plus $k_0 = k_n$. Montrer que le nombre minimal de multiplications de flottants nécessaires au calcul de $B^1 \times \cdots \times B^{n-1}$ est égal au nombre minimal de multiplications de flottants nécessaires au calcul de $B^n \times B^1 \times \cdots \times B^{n-2}$.

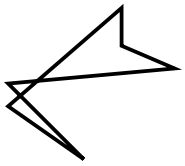
Oublions désormais les poids sur les sommets de la triangulation d'un polygone. On représente un polygone comme un tableau de points dans $\mathbb{R} \times \mathbb{R}$. Il y a une arête entre deux sommets successifs du tableau, ainsi qu'entre la première et la dernière valeur.

```
type polygone = (float * float) array
```

Un polygone est dit simple si deux arêtes non consécutives ne se croisent pas, et deux arêtes consécutives n'ont en commun que l'un de leurs sommets. On ne considéra que des polygones simples et on ne cherchera pas à vérifier leur simplicité.



Polygone simple



Polygone non-simple

On formalise la triangulation d'un polygone comme une liste d'arêtes à ajouter au polygone. Les sommets du polygone sont désignés par leur place dans le tableau représentant le polygone : le premier sommet est celui en position 0 dans le tableau, etc. Une arête entre le i-ème sommet du polygone et le j-ème sommet est décrite comme un couple d'entiers (i, j) avec $i < j$.

Question V.3. Donner une fonction OCaml testant la convexité d'un polygone simple :

`est_convexe: polygone -> bool`.

Indication : On pourra utiliser l'orientation de produits vectoriels de vecteurs bien choisis.

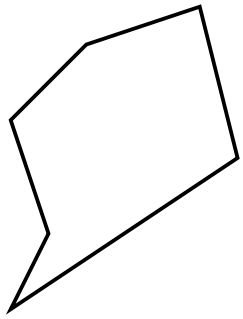
Question V.4. Combien d'arêtes la triangulation d'un polygone convexe à n cotés introduit-elle ? Justifier.

Question V.5. Donner une fonction OCaml

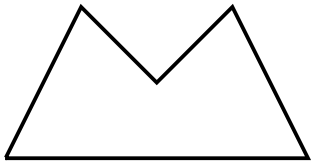
`triangule_convexe: polygone -> (int * int) list`

qui prend en entrée un polygone supposé convexe et renvoie une triangulation de ce polygone sous la forme d'une liste d'arêtes en un temps linéaire en le nombre de sommets du polygone.

On considère désormais des polygones non-convexes. Un polygone simple est dit *strictement monotone* par rapport à l'axe des ordonnées si toute ligne horizontale ne coupe le polygone qu'en au plus deux points.



Polygone simple strictement monotone



Polygone simple non strictement monotone

Ainsi, si on imagine un point se déplaçant le long d'un polygone simple, de son sommet le plus haut vers son sommet le plus bas, il ne fera que descendre ou se déplacer horizontalement, jamais il ne remontera.

Question V.6. Donner une fonction OCaml

`separe_polygone: polygone -> int list * int list`

qui prend en entrée un polygone simple strictement monotone et sépare ses sommets (représentés par leurs indices) en deux listes, l'une contenant les sommets du plus haut (inclus) vers le plus bas (exclus) en suivant l'un des deux chemins à partir du sommet le plus haut, l'autre les sommets du plus haut (exclus) vers le plus bas (inclus) en suivant l'autre chemin. Ainsi, tout sommet du polygone doit être classé une et une seule fois dans l'une des listes en sortie, et chaque liste est triée par ordre décroissant de deuxième coordonnée. La fonction donnée doit être linéaire en le nombre de sommets du polygone.

Question V.7. Proposer un algorithme qui prend en entrée un polygone supposé simple strictement monotone et renvoie une triangulation de ce polygone en un temps linéaire en le nombre de sommets du polygone. Cet algorithme pourra commencer par utiliser l'algorithme implémenté par la fonction `separe_polygone`.

Fin du sujet.