MINISTÈRE
DE L'ÉDUCATION
NATIONALE ET
DE LA JEUNESSE

Liberté • Égalité • Fraternité
RÉPUBLIQUE FRANÇAISE

EAE SIN 3

SESSION 2020

AGRÉGATION
CONCOURS EXTERNE

Section : SCIENCES INDUSTRIELLES DE L'INGÉNIEUR

Option : SCIENCES INDUSTRIELLES DE L'INGÉNIEUR ET INGÉNIERIE INFORMATIQUE

CONCEPTION PRÉLIMINAIRE D'UN SYSTÈME, D'UN PROCÉDÉ OU D'UNE ORGANISATION

Durée : 6 heures

Calculatrice électronique de poche - y compris calculatrice programmable, alphanumérique ou à écran graphique – à fonctionnement autonome, non imprimante, autorisée conformément à la circulaire n° 99-186 du 16 novembre 1999.

L'usage de tout ouvrage de référence, de tout dictionnaire et de tout autre matériel électronique est rigoureusement interdit.

Si vous repérez ce qui vous semble être une erreur d'énoncé, vous devez le signaler très lisiblement sur votre copie, en proposer la correction et poursuivre l'épreuve en conséquence. De même, si cela vous conduit à formuler une ou plusieurs hypothèses, vous devez la (ou les) mentionner explicitement.

NB : Conformément au principe d'anonymat, votre copie ne doit comporter aucun signe distinctif, tel que nom, signature, origine, etc. Si le travail qui vous est demandé consiste notamment en la rédaction d'un projet ou d'une note, vous devrez impérativement vous abstenir de la signer ou de l'identifier.

Tournez la page S.V.P.

Ce document est composé de 48 pages :

- d’un dossier sujet de la page 1 à la page 21 ;
- d’un dossier technique (DT) de la page 22 à la page 42 ;
- d’un dossier réponse (DR) de la page 43 à la page 48.

Conseils aux candidats :

Les différentes parties du sujet sont indépendantes.

Un parcours attentif de l’ensemble du document est conseillé avant de composer.

La présentation des programmes doit respecter les mots clés du langage cible ainsi que l’indentation des structures algorithmiques.

Les réponses doivent être présentées avec clarté, rigueur et concision.

INFORMATION AUX CANDIDATS

Vous trouverez ci-après les codes nécessaires vous permettant de compléter les rubriques figurant en en-tête de votre copie

Ces codes doivent être reportés sur chacune des copies que vous remettrez.

Concours	Section/option	Epreuve	Matière
EAE	1417A	103	1268

FRIPON, réseau de surveillance du ciel français pour pister les météorites

Présentation

L'explosion le 15 février 2013 d'une très grosse météorite au-dessus de la ville russe de Tchéliabinsk a surpris le monde entier et a déclenché une véritable prise de conscience auprès de l'opinion publique et des pouvoirs publics : un tel événement peut se reproduire n'importe où, n'importe quand.

De façon générale, la grande majorité des bolides et météores se désintègre totalement en pénétrant dans l'atmosphère terrestre et finit en poussière sans même avoir atteint le sol. Il est possible qu'un objet céleste de quelques mètres puisse produire des météorites capables d'atteindre le sol. Les estimations actuelles sont d'une dizaine de météorites par an en France. Parmi ces objets touchant le sol, seul un est retrouvé en moyenne tous les 10 ans. Étonnamment ce taux était cinq fois plus important au XIX^{e} siècle.

Sur ce constat, trois chercheurs François Colas (chercheur CNRS à l'Observatoire de Paris), Brigitte Zanda (enseignante-chercheuse au Muséum national d'Histoire naturelle) et Sylvain Bouley (enseignant-chercheur à l'Université Paris-Sud) ont mis sur pied depuis 2013 un dispositif du nom de FRIPON, acronyme pour « Fireball Recovery InterPlanetary Observation Network » (en français : réseau de recherche de bolides et de matière interplanétaire) dans le but de détecter et organiser les recherches des météorites.

Le principe

Trois à neuf caméras sont implantées par région à des distances respectives de 50 à 100 kilomètres. La figure 1 montre l'implantation dense des caméras en France et son extension en cours en Europe.

Figure 1. Carte européenne partielle des sites du réseau FRIPON

Simples d'installation et d'utilisation, les caméras sont dotées d'un objectif capturant une demi-sphère permettant une vue très large à 360° de la voûte céleste. Les données sont transmises sous la forme d'une seule image rectangulaire présentant un effet *fisheye (exemple visible en figure 2).*

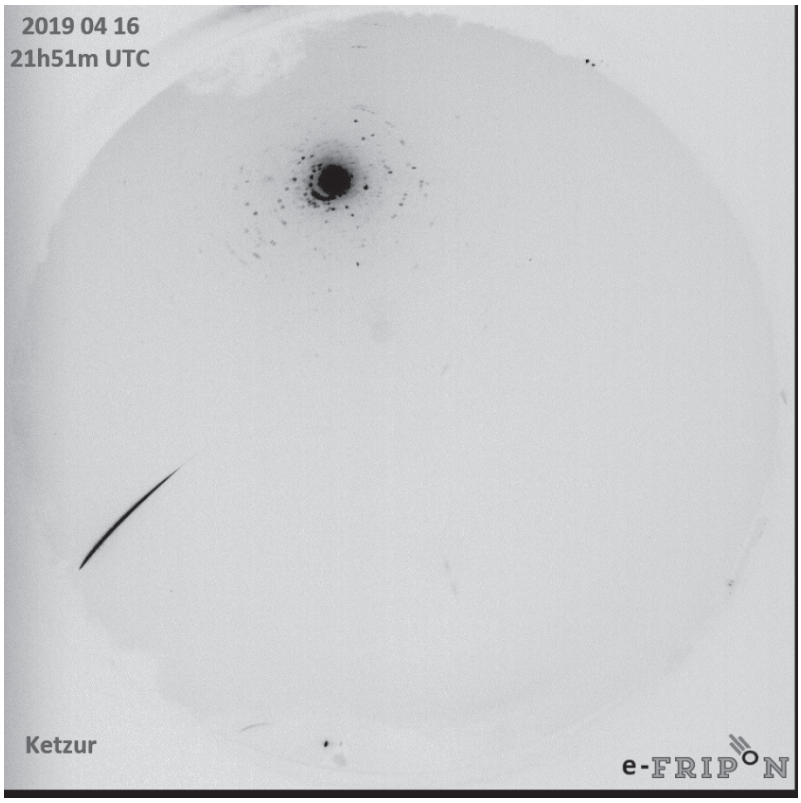


Figure 2. Exemple d'image prise par une caméra avec objectif fisheye

Figure 3. Photo d'une caméra implantée à l'Observatoire de Paris

Sur chaque site d'implantation (figure 3), la caméra est raccordée à une unité informatique munie d'un logiciel (*Freeture)* développé spécialement pour analyser les images et détecter les événements lumineux. Lorsqu'une détection survient, une alerte est transmise au calculateur central situé à Marseille, qui recueille les données de tout le réseau en temps différé. Sur la base d'une observation il est ainsi possible de déclencher une campagne de recherche de l'impact sur le terrain en une journée environ.

Architecture de l'ensemble du système

La figure 4 présente une vue d'ensemble de l'architecture matérielle du système.

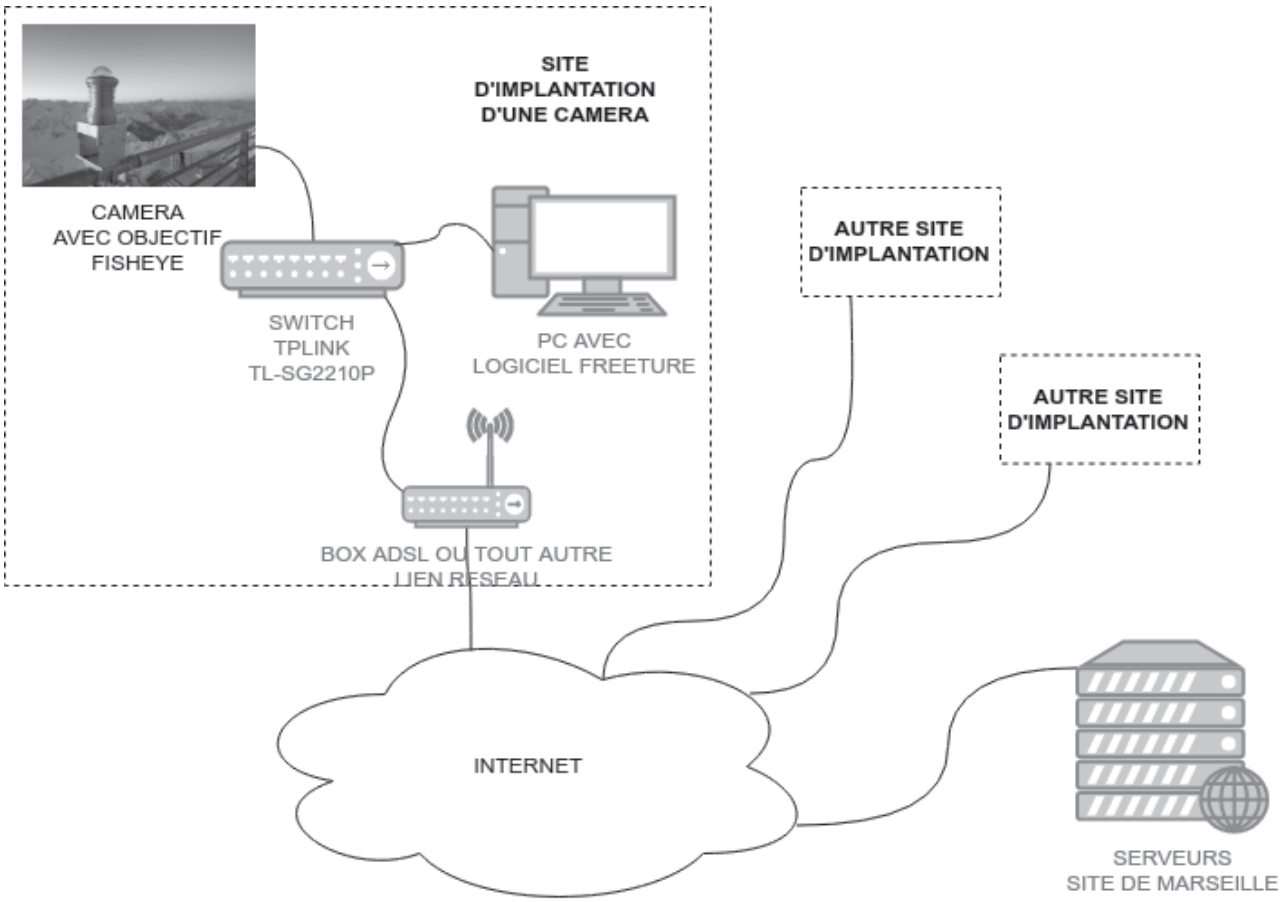


Figure 4. Vue d'ensemble du système FRIPON

L'architecture globale est la suivante :

- une caméra est connectée à un *switch* (commutateur réseau) ;
- l'unité informatique locale (PC embarqué compact de type Intel NUC) héberge le logiciel *Freeture* ;
- ce logiciel récupère les images de la caméra, puis les traite ;
- lors de la détection d'un événement, les données sont sauvegardées localement et un mail est envoyé aux serveurs du réseau pour validation ;
- si plusieurs stations géographiquement proches détectent des événements au cours de la même période, un opérateur peut démarrer une procédure logicielle pour récupérer l'ensemble des images sauvegardées par les stations concernées.

Ce sujet propose l'étude de ce système ; il est composé de trois parties indépendantes :

- la première porte sur le logiciel *Freeture* qui détecte et alerte le site principal lors de la détection d'un événement ;
- la deuxième porte sur l'interconnexion de tous les sites locaux au site principal de Marseille ;
- la dernière porte sur le traitement réalisé sur toutes les images d'un événement afin de calculer la trajectoire d'une météorite et en déduire son point de chute potentiel.

Partie 1. Étude du logiciel Freeture

Cette partie porte sur l'étude du logiciel « *Freeture »* de détection des météorites. Ce logiciel doit analyser toutes les images disponibles issues d'une caméra et transmettre au système central de Marseille les éventuelles détections d'événement sans interrompre l'analyse des images suivantes.

Sous-partie 1.1. Architecture multi-tâches

Les contraintes temporelles dans ce système sont les suivantes :

- la caméra est configurée pour transmettre 30 images par seconde ;
- la détection pour filtrer les faux positifs nécessite en moyenne 20 ms mais peut quelques fois dépasser 1 s ;
- l'envoi d'un email pour prévenir d'une détection requiert en moyenne 500 ms.

Q1.1. Justifier la conception multi-tâche de cette application.

L'application multitâche parallélise trois traitements et une structure de stockage appelée *buffer*, chacun implémenté sous la forme d'un *thread* écrit dans une classe distincte :

- « tAcqThread » exécute l'acquisition des images et les stocke dans le *buffer* ;
- « tDetThread » exécute la détection des événements sur les images lues depuis le *buffer* et réalise l'envoi d'emails ;
- « tStackThread » exécute la sauvegarde sur le disque de toutes images présentes dans le *buffer* après détection d'un événement en vue d'une analyse détaillée par les chercheurs.

D'autres classes sont nécessaires à la gestion des ressources :

- la classe « tCamera » gère les accès à la caméra, notamment la récupération des images, cette classe n'est pas « *thread-safe* » ;
- la classe « tCircularBuffer » implémente le stockage *buffer* circulaire des images en attente de traitement, cette classe n'est pas « *thread-safe* » ;
- la classe « tCfgParam » stocke tous les paramètres d'exécution de l'application (calibrations et options de fonctionnement) sous forme de fichiers textes ;
- la classe « tMail » gère l'envoi d'un email, cette classe n'est pas « thread-safe » ;
- la classe « tThreadGeneric » mutualise les méthodes basiques de création, de démarrage, d'arrêt et de destruction d'un *thread*.

Q1.2. Donner les caractéristiques d'une conception multi-tâche basée sur des processus communicants indépendants et celles d'une conception multi-tâche basée sur un processus unique comportant plusieurs fils d'exécution.

Q1.3. Expliquer les dispositifs de synchronisation et de communication nécessaires au fonctionnement global pour ces deux architectures. Parmi ces deux propositions, choisir et justifier le choix de l'architecture multi-tâches pour cette application.

Q1.4. Justifier la nécessité de synchroniser l'exécution des *threads* d'acquisition, de sauvegarde et de détection pour transférer les images. Expliquer les problèmes possibles en l'absence de synchronisation.

Q1.5. Compléter le diagramme de classes du document réponse DR1 en ajoutant les liens décrits dans la description du système et des classes disponibles.

Sous-partie 1.2. Conception d'une synchronisation

La suite de l'étude est limitée à deux *threads* partageant le *buffer* circulaire de stockage des images :

- le *thread* d'acquisition récupère les images de la caméra et les stocke dans le *buffer* ;
- le *thread* de détection récupère les images du *buffer* pour les analyser.

Le *thread* d'acquisition ne peut écrire dans le *buffer* que s'il reste de la place. Le *thread* de détection n'est sollicité que si des images à traiter sont présentes dans le *buffer*. Les éléments de synchronisation nécessaires sont :

- un sémaphore VIDE ;
- un sémaphore PLEIN ;
- un *mutex* d'accès au *buffer* ACCES.

Le *buffer* a une capacité de N places. Les fonctions primitives pour utiliser les *mutex*/sémaphores sont celles habituellement décrites dans la littérature du domaine, à savoir :

- **P()** permet de verrouiller un *mutex* : si le *mutex* est déjà verrouillé par un autre *thread*, l'appelant est suspendu par le noyau jusqu'au déverrouillage de ce *mutex*. Sinon, le *mutex* est verrouillé pour l'appelant au retour de la fonction ;
- **P()** appliqué sur un sémaphore suspend l'appelant si son compteur interne vaut zéro, sinon le compteur interne est simplement décrémenté de 1. La suspension pour l'appelant prend fin lorsqu'un autre *thread* incrémente ce sémaphore ;
- **V()** appliqué sur un *mutex* le déverrouille et réveille un autre *thread* en attente de cet objet. Dans tous les cas, aucune suspension possible pour le *thread* appelant ;
- **V()** appliqué sur un sémaphore incrémente son compteur interne. Dans tous les cas, aucune suspension possible pour le *thread* appelant.

Par construction de l'algorithme du document réponse DR2, les compteurs des sémaphores VIDE et PLEIN sont mis à jour de façon cohérente. Un emplacement est compté « plein » une fois écrit, un emplacement est compté « vide » une fois libéré.

Q1.6. Compléter les cases du document réponse DR2 afin d'ajouter les instructions gérant les sémaphores VIDE et PLEIN ainsi que le *mutex* ACCES. La solution doit respecter la description faite en début de sous-partie.

Q1.7. Montrer succinctement l'absence d'interblocage entre les *threads* de cette solution.

Tournez la page S.V.P.

Sous-partie 1.3. Implémentation de la solution

L'implémentation de cette solution est réalisée en utilisant la bibliothèque *pthread*. La norme C++11 introduit de nouveaux mécanismes de synchronisation implémentés avec une orientation objet. Ceci répond à des problématiques de rapidité, de maintenance et de robustesse du code.

Pour comprendre l'implémentation réelle de la synchronisation, une partie du code C++ est fournie dans le document technique DT1 où les lignes synchronisées des deux threads sont encadrées.

Les classes utilisées dans ce mécanisme sont *condition_variable* (document technique DT2)*, unique_lock* (document technique DT3), et *mutex* (structure opaque de mutuelle exclusion). L'utilisation de la classe *condition_variable* doit prendre en compte le phénomène de « *spurious wakeup* » décrit dans le document technique DT4.

Q.1.8. Expliquer l'intérêt de la classe `unique_lock` et décrire le rôle de la variable booléenne pointée par `detSignal`. Préciser le rôle de l'appel de la méthode `wait()` de la classe `condition_variable` par le *thread* appelant.

Dans le document technique DT1, deux blocs de code sont encadrés dans le fil d'exécution du *thread* d'acquisition et deux blocs sont encadrés dans le fil d'exécution du *thread* de détection.

Q.1.9. Décrire le rôle de ces quatre blocs de code encadrés.

Q1.10. Montrer que l'implémentation présentée assure la synchronisation des deux *threads*, y compris si le *thread* de détection s'exécute avant le *thread* d'acquisition.

Q.1.11. Expliquer le fonctionnement du *thread* de détection lorsque celui-ci détecte un événement.

Sous-partie 1.4 Conclusion

Q1.12. Par une synthèse, conclure sur l'architecture multitâche du logiciel Freeture.

Partie 2. Étude du réseau d'un système d'acquisition

Chaque système d'acquisition est hébergé par une université, un institut partenaire, un club d'astronomie ou un particulier… Le référent local, ayant rarement les compétences informatiques nécessaires, réalise l'installation matérielle du système mais ne se charge pas de son administration. Il est donc nécessaire que le système informatique puisse se configurer de manière autonome et être accessible à distance depuis le site de Marseille.

Chaque caméra est reliée via un *switch* à un PC embarqué (Intel NUC) réalisant les premiers traitements. Le switch est connecté au réseau local de l'hébergeur afin de communiquer avec les serveurs centraux de Fripon.

Le principe de fonctionnement du système d'acquisition est le suivant :

- la caméra communique par IP avec le PC embarqué pour transférer les images ;
- le PC embarqué reçoit toutes les images, les traite et sélectionne uniquement les observations ayant un intérêt (observation d'un bolide lumineux, voir partie précédente avec Freeture) en sauvegardant les images correspondantes sur son disque ;
- le logiciel de traitement prévient le site de Marseille par envoi d'emails ;
- si un événement a été repéré par plusieurs stations, un opérateur peut télécharger les images précédemment sauvegardées sur le disque local vers les serveurs centraux de Fripon.

Le schéma réseau résumant l'architecture du système est donné figure 4.

Les exigences de fonctionnement sont les suivantes :

1. l'administration du système est réalisée intégralement à distance dans le but de changer sa configuration réseau, de mettre à jour les logiciels et les systèmes d'exploitation. Les ingénieurs en charge de la maintenance générale du réseau Fripon se connectent à distance sur le PC embarqué afin d'accéder aux autres composants ;
2. étant donné le grand nombre de systèmes d'acquisition connectés à l'infrastructure, les échanges avec le réseau central Fripon sont limités. Les échanges volumineux de données entre la caméra et le PC embarqué ne transitent pas via le réseau central ;
3. le système d'acquisition n'utilise le réseau local de l'hébergeur (généralement une université, un institut partenaire) que pour les communications vers le réseau central Fripon. L'accès direct à d'autres adresses/réseaux est interdit ;
4. les échanges entre la caméra et le PC embarqué ne transitent pas par le réseau local de l'hébergeur. Autrement dit, le domaine de collision entre la caméra et le PC est différent de celui du réseau d'accueil ;
5. le système d'acquisition n'est pas accessible depuis le réseau local de l'hébergeur. Un utilisateur connecté au réseau local de l'hébergeur ne peut communiquer ni avec la caméra ni avec le PC embarqué.

Les trois sous-parties suivantes sont indépendantes et peuvent être traitées séparément.

Tournez la page S.V.P.

Sous-partie 2.1. Configuration des VLAN et des ports Ethernet

Le système d'acquisition est constitué de différents équipements (figure 4). Leurs configurations sont réalisées avant l'envoi du matériel à l'hébergeur par transporteur.

Pour le fonctionnement, deux VLAN (*Virtual Local Area Network*) sont définis :

- le VLAN « vision » véhicule les images issues de la caméra ;
- le VLAN « hébergeur » relie le système d'aquisition au réseau local de l'hébergeur.

Q2.1 Compléter les cases à cocher du document réponse DR3 pour affecter les ports dans les différents VLAN et permettre le fonctionnement et de valider l'exigence 4.

Le switch connectant les différents éléments du système d'acquisition et le réseau local de l'hébergeur a les caractéristiques suivantes :

- 8 ports LAN, 1 port WAN (*Wide Area Network*) et 1 port d'administration interne ;
- support des VLAN avec affectation possible d'un port dans plusieurs VLAN ;
- possibilité de taguer l'entête des paquets Ethernet (IEEE 802.1Q).

La notion de « tag » *Ethernet* est standardisée dans la norme IEEE 802.1Q :

- la configuration « tagged » indique que l'identificateur du VLAN est ajouté dans l'entête de chaque paquet transitant via ce port. La configuration « untagged » ne transmet pas l'identificateur du VLAN dans ces paquets (utilisation de l'entête Ethernet classique) ;
- les configurations des ports reliant deux équipements doivent être cohérentes : un appareil « untagged » ne peut être relié qu'à un port configuré « untagged » du switch (et inversement) ;
- un port « tagged » peut être affecté à plusieurs VLAN. Un port « untagged » ne peut être affecté qu'à un seul VLAN.

Q2.2 Compléter les cases à cocher dans le document réponse DR4 afin d'indiquer la configuration des ports du switch et du PC embarqué.

Au démarrage, c'est-à-dire avant l'établissement du VPN détaillé dans les sous-parties suivantes, la configuration des interfaces réseaux et la table de routage du PC embarqué sont précisées dans le document technique DT5. L'interface « eth0 » se trouve dans le VLAN « hébergeur », l'interface « eth0.2 » est dans le VLAN « vision ».

Q2.3. Extraire les valeurs numériques des identificateurs des deux VLAN. Pour chaque VLAN, préciser si la configuration est en IPv4 ou IPv6 et calculer le nombres d'équipements physiques pouvant être connectés, préciser les adresses de broadcast.

Q2.4. Préciser ce qu'indique la partie « fe:80 :: » avec l'élément « Scope : Link » des interfaces *eth0* et *eth0.2*.

Le switch est administrable à distance par protocole IP. Par mesure de simplicité, son administration sera réalisée via une interface d'administration accessible sur le VLAN « vision ».

Q2.5. Lister les adresses IP disponibles sur ce VLAN.

Q2.6. Préciser les notions d'adresses IP routables et non-routables. Pour chaque VLAN, préciser si les adresses IP des VLAN sont routables ou non et indiquer si le trafic peut être routé vers un autre réseau.

Sous-partie 2.2. Liaison sécurisée VPN côté client

La configuration réseau du PC embarqué est automatiquement modifiée après l'établissement de la liaison VPN. Les nouveaux paramètres sont présents dans le document technique DT6. L'interface « tun0 » est une interface réseau logicielle créée par le client VPN réécrivant les trames d'un réseau virtuel « VPN » vers un réseau réel réalisant le transfert physique des paquets cryptés.

Afin de faciliter l'administration à distance (SSH) et de sécuriser les communications du système d'acquisition vers le serveur central, un tunnel sécurisé est à concevoir **depuis le PC embarqué**. Il existe principalement deux technologies distinctes pour se connecter à un réseau distant :

- un réseau privé virtuel (VPN) permet le transfert bidirectionnel de paquets IP. L'établissement des connexions est également bidirectionnel. Un VPN fait le lien entre différentes stations en encapsulant leurs échanges dans une connexion sécurisée. Un VPN peut re-router automatiquement tout le trafic entre les différents sites ;
- un proxy sécurisé (Proxy SSL) centralise les connections entre différentes applications clients et serveurs. Il fonctionne uniquement pour certains protocoles (par exemple HTTP) et sa mise en place nécessite des clients compatibles. Seules les connexions entre ce client et le serveur transiteront par le proxy.

Q2.7. Justifier la nécessité de créer un tunnel sécurisé **depuis la station locale** vers le serveur.

Q2.8. En s'appuyant sur la structure du système Fripon, justifier le choix d'une solution VPN. Préciser la ou les couches du modèle OSI classifiant ce service.

Le logiciel client Freeture ouvre un port client réseau sur l'adresse IP locale « 10.8.0.218 » et communique avec le serveur central dont l'adresse est « 10.8.1.10 ». La liaison VPN est établie vers le serveur 1.2.3.4 (adresse « anonyme » du serveur réel).

Q2.9. Vérifier que le routage d'un paquet du client vers le serveur est possible en précisant les opérations de routage.

Q2.10. Les serveurs de mise à jour du PC embarqué ont les adresses IP 212.211.132.250 et 217.196.149.233. Justifier les règles de routages explicites pour ces adresses IP.

Sous-partie 2.3. Logiciel client VPN

Du côté client, la mise en place du VPN suit la séquence suivante :

1. le client VPN sur le PC embarqué ouvre une connexion (tunnel) vers le serveur VPN dont l'adresse IP est 1.2.3.4 (adresse « anonyme » du système Fripon réel) ;
2. cette connexion nécessite la négociation de protocoles de chiffrement et de sécurisation gérés au niveau de la couche « application » ;

3. le client VPN crée une interface réseau virtuelle « tun0 » dont il relaie tout le trafic pour l'encapsuler dans le tunnel sécurisé ;
4. le client DHCP du système d'exploitation s'exécute pour cette interface réseau virtuelle et obtient, par l'intermédiaire du tunnel, l'adresse IP locale 10.8.0.218 et la passerelle 10.8.0.217 ;
5. le client VPN ajoute des routes afin d'orienter le trafic par l'interface réseau sécurisée.

Le code de la fonction « main » du programme « vpnc » (client VPN – licence GPL) est donné dans le document technique DT7.

Q2.11. Résumer les principes de la licence GPL. Préciser si cette licence autorise la reproduction du code dans ce sujet. Relever les numéros des lignes du programme réalisant les étapes 1 à 3 de la séquence décrivant la mise en place du VPN.

Q2.12. Le protocole de transport utilisé pour le tunnel entre le client et le serveur VPN est basé sur UDP, justifier l'utilisation de la boucle entre les lignes 3234 et 3244.

Q2.13. À partir des mots clés présents dans ce code source, déterminer le protocole de sécurisation de la liaison et l'algorithme de chiffrement utilisé.

Sous-partie 2.4. Vérification des exigences

Q2.14. Valider la conception de la solution technique vis-à-vis des cinq exigences énoncées en début de partie.

Q2.15. Proposer une modification des règles de routage présentées dans le document technique DT6 afin de renforcer l'exigence 3 et rediriger inconditionnellement tout le trafic vers « internet » via le tunnel sécurisé.

Q2.16. Proposer une solution technique à appliquer sur le PC embarqué renforçant l'exigence 5 décrite dans l'introduction de la partie 2.

Partie 3. Détermination de la trajectoire d'un météore

Sous-partie 3.1. Présentation

L'objectif de cette partie est d'estimer la zone de chute de la météorite sur la Terre. Pour cela, il faut estimer la trajectoire à partir des clichés dans un repère cartésien géocentrique (repère lié à la Terre). Cette trajectoire est divisée en deux phases successives schématisées dans la figure 5 :

- la première phase consiste en l'entrée dans l'atmosphère à haute vitesse, le météore est visible dans le ciel (phase de combustion) dès que son altitude est inférieure à 80 km. Au cours de cette phase relativement courte (quelques secondes), le rayon de courbure dû à l'attraction terrestre est très grand et la vitesse du bolide est très élevée. La trajectoire du météore est approximée par une droite prolongeant la trajectoire d'orbite. Le météore devient lumineux (il est alors nommé *fireball*) dès que l'atmosphère a suffisamment chauffé sa surface (forces de frottement) et « s'éteint » lorsqu'il a suffisamment décéléré ;
- la deuxième phase consiste en la trajectoire lors de la phase de « *dark flight »*, le météore a suffisamment décéléré pour être assimilé à un corps en chute libre dans l'atmosphère avec une vitesse initiale. Cette phase peut durer plusieurs minutes et ne permet pas d'observation optique directe car l'objet est alors assimilable à un corps noir de petite taille. L'objet touchant le sol s'appelle une météorite.

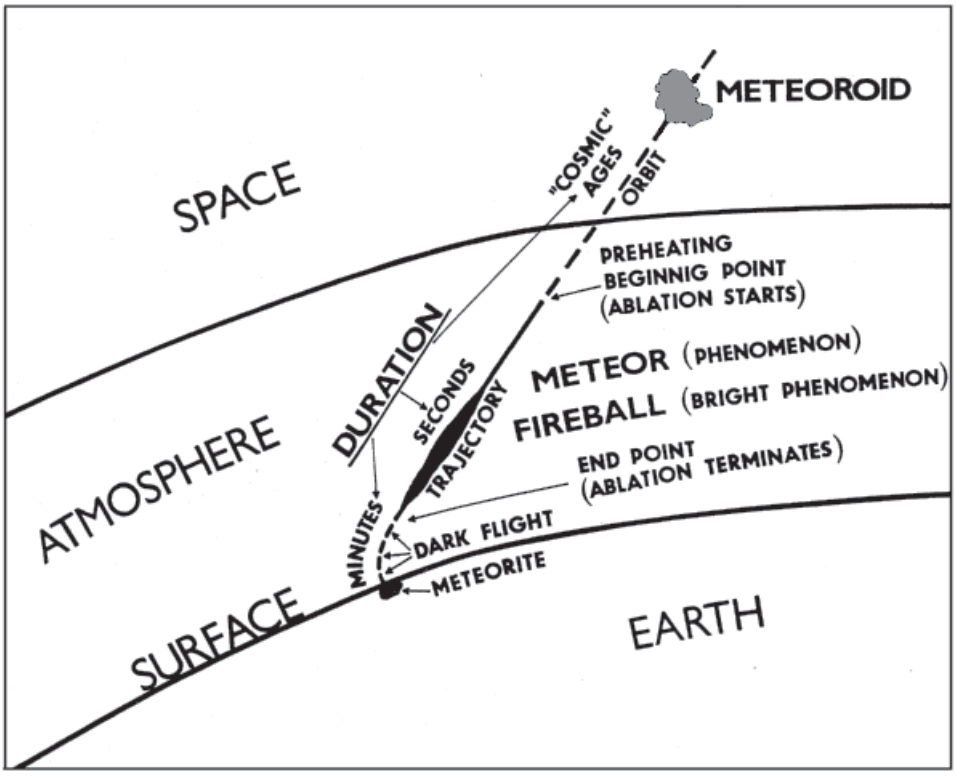


Figure 5. Illustration des différentes phases de la chute

Le point de chute ne peut pas être obtenu précisément car selon les caractéristiques propres du météore (forme, constitution...), les interactions avec l'atmosphère peuvent présenter des variations. Celles-ci sont assez importantes surtout lors du « *dark flight »* (phase en chute libre) au cours de laquelle aucune mesure ne peut être réalisée. Le résultat donne une zone de recherche souhaitée la plus petite possible. Cette zone est ensuite fouillée lors de recherches sur le terrain faites par des volontaires.

Le logiciel *Freeture* présent dans chaque station d'observation déclenche un événement lors de l'observation d'une météorite. Plusieurs observations simultanées dans une zone géographique rapprochée (quelques centaines de kilomètres au maximum) sont nécessaires à l'obtention de la trajectoire. Au moins deux stations proches doivent détecter le même événement afin d'appliquer la méthode étudiée dans cette partie.

Lorsqu'au moins deux événements sont détectés dans le même intervalle temporel par deux stations proches, il est possible d'estimer la trajectoire du météorite grâce aux images provenant des caméras. Il est nécessaire d'appliquer plusieurs traitements avant d'estimer la position du bolide dans un repère géocentrique. Une triangularisation permet de déterminer la trajectoire. Cette méthode consiste à repérer des points de la trajectoire par plusieurs caméras aux mêmes instants. À partir de ces points, il est possible de déterminer analytiquement l'équation de la trajectoire.

Q3.1. Justifier la nécessité d'obtenir au moins deux observations proches pour localiser précisément la météorite. Discuter de la précision temporelle nécessaire pour réaliser une triangularisation point-à-point en se basant sur les différentes images simultanées des caméras.

Sous-partie 3.2. Méthode de calcul de la trajectoire en phase de « combustion »

Le calcul analytique par triangularisation de la trajectoire en phase de combustion impose des contraintes temporelles fortes dans l'acquisition des images. De meilleurs résultats sont obtenus grâce à un algorithme heuristique de type *optimisation par essaim particulaire* afin de déterminer la trajectoire de la météorite : celle-ci est obtenue « au mieux » en se basant sur les trajectoires apparentes captées par les caméras.

Chaque image observant une météorite lumineuse permet de déterminer une ligne de vue. Celle-ci est modélisée dans le programme par la position de la caméra (point de passage à trois coordonnées de la droite) et par une direction et un sens dans l'espace (vecteur directeur à trois coordonnées issu de la vision de la caméra). Ce modèle de représentation des droites dans l'espace est préféré pour simplifier les calculs ultérieurs.

Les images successives d'une caméra observant le déplacement du point lumineux construisent donc un ensemble de droites concourantes (la position de la caméra est supposée fixe pendant le court instant d'observation). Ces lignes de vues successives visent la trajectoire observée du bolide à différents instants (figure 6).

Plusieurs caméras observant la même météorite produiront plusieurs ensembles de droites concourantes toutes dirigées vers l'unique trajectoire (représentation dans les figures 6 et 7).

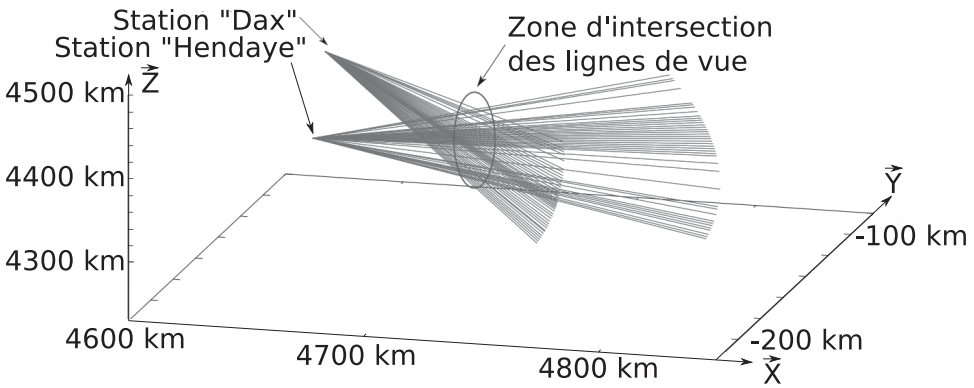


Figure 6. Lignes de vue de deux stations (Dax et Hendaye) pour un unique météore.

Toutes les mesures issues des observations sont altérées par la discrétisation spatiale des images, l'imprécision dans la mesure du centre du météore, le halo lumineux lié à la traînée dans l'atmosphère… L'intersection des lignes de vues et de la trajectoire linéaire du bolide n'est pas calculable analytiquement. La méthode a pour but de chercher la droite modélisant la trajectoire passant au mieux dans la zone d'intersection des droites modélisant les lignes de vues. L'algorithme heuristique aura pour objectif de minimiser la distance entre chaque trajectoire potentielle et chacune des lignes de vue afin d'estimer l'expression de la droite représentant la trajectoire du météore au mieux.

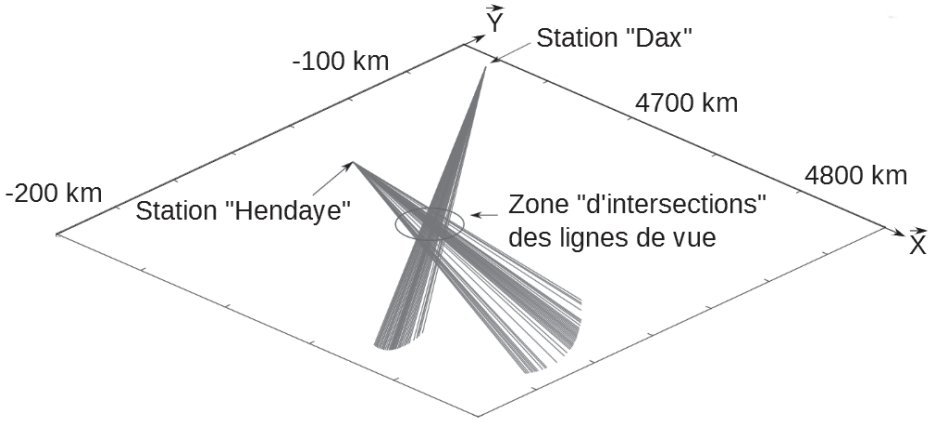


Figure 7. Autre point de vue montrant les lignes se croisant dans une zone spatiale.

Le modèle des droites étant basé sur un point de passage et un vecteur directeur, il est nécessaire de connaître précisément les positions des stations dans un repère cartésien, d'obtenir la direction de chaque ligne de vue dans ce même repère et enfin de d'appliquer l'*algorithme par essaim particulaire* afin d'obtenir les caractéristiques de la trajectoire. Ces différentes étapes sont étudiées dans la suite du sujet.

Q3.2. Évaluer la robustesse de cette méthode vis-à-vis des imprécisions de mesure du centre des météores dans les images et vis-à-vis de la précision temporelle de ces observations.

Sous-partie 3.3. Coordonnées des caméras

Pour faciliter le développement, le repère cartésien choisi $(O, \vec{x}, \vec{y}, \vec{z})$ est orthonormé, géocentré (origine au centre de la Terre) et fixe par rapport à celle-ci.

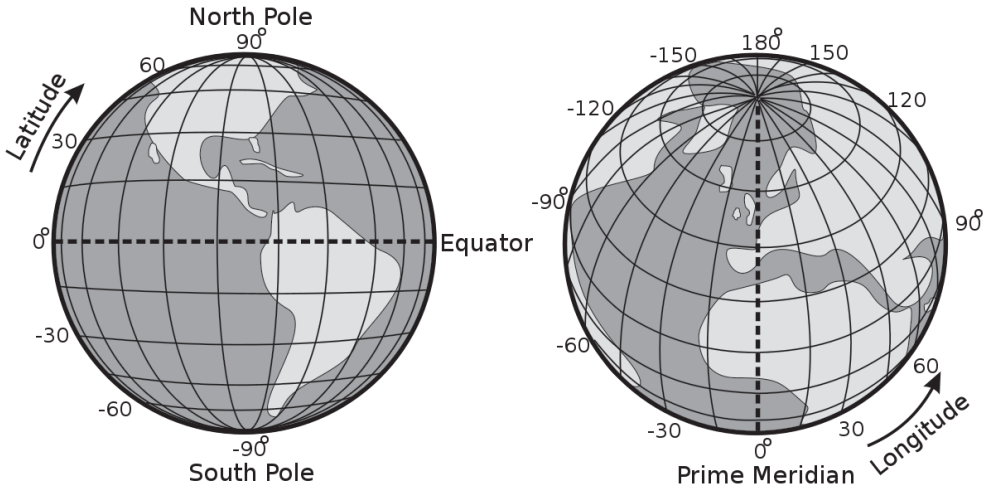


Figure 8. Définition du repère géographique (source Wikipédia – Creative Commons)

Tournez la page S.V.P.

La position GPS de chaque caméra est connue dans un repère géographique (figure 8) exprimé en longitude (donnée en degrés par rapport au méridien « zéro » de Greenwitch), en latitude (donnée en degrés par rapport au plan de l'équateur) et en hauteur (ou altitude) par rapport à la surface de référence de la planète.

Le modèle de la surface de la planète utilisé par le système GPS est une ellipsoïde (sphère aplatie aux pôles) telle que spécifiée dans le système WGS84. Cette surface et les axes cartésiens sont illustrés dans la figure 9. Dans ce modèle, la longitude est notée λ, la latitude φ et l'altitude *h* par rapport à l'ellipsoïde.

Cette ellipsoïde est alignée sur le repère géocentrique. Son origine, notée O, est le centre de masse de la Terre, l'axe $\vec{z}$ est orienté vers le Nord selon l'axe de rotation de la planète, le plan de l'équateur $(O, \vec{x}, \vec{y})$ est perpendiculaire à l'axe $\vec{z}$ et enfin l'axe $\vec{x}$ passe par l'intersection de l'équateur et du méridien « zéro ».

Le modèle mathématique de cette surface ellipsoïde de référence peut être simplifié en transformant les coordonnées en un repère sphérique équivalent pour lequel le rayon de la planète dépend de la latitude. Ce repère intermédiaire est décrit dans la figure 10. Dans cette représentation sphérique, l'angle polaire habituel est substitué par une latitude virtuelle désignée φ', la longitude sphérique est notée λ'.

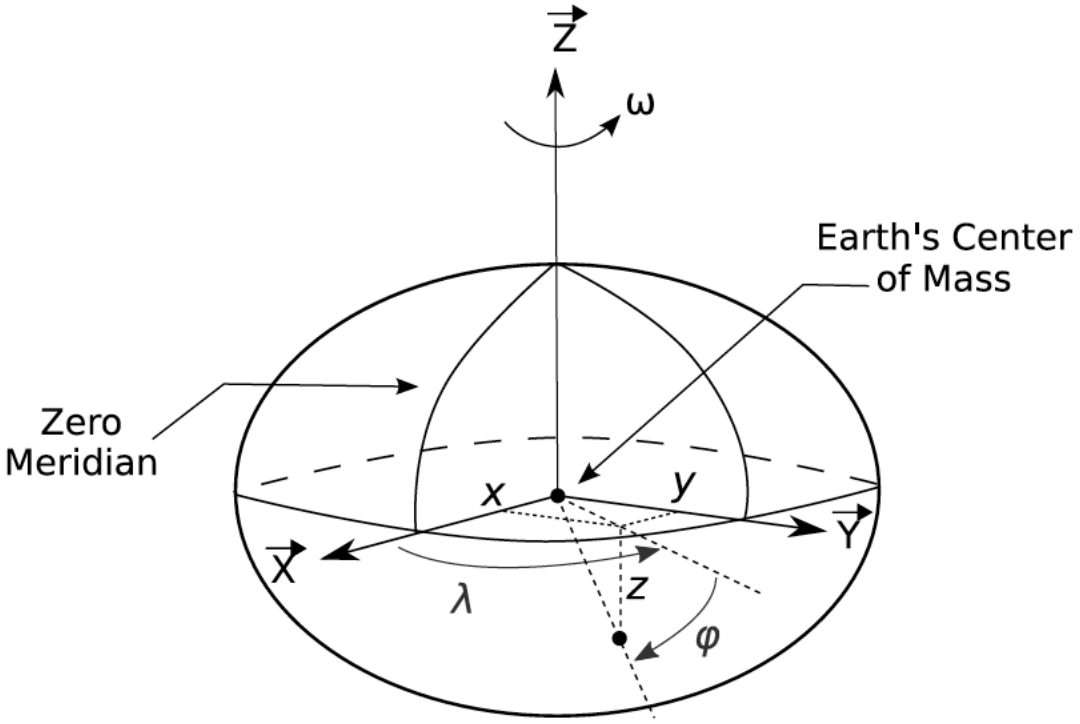


Figure 9. Définition du repère WGS84 (Source NIMA report 8350.2 avec modifications)

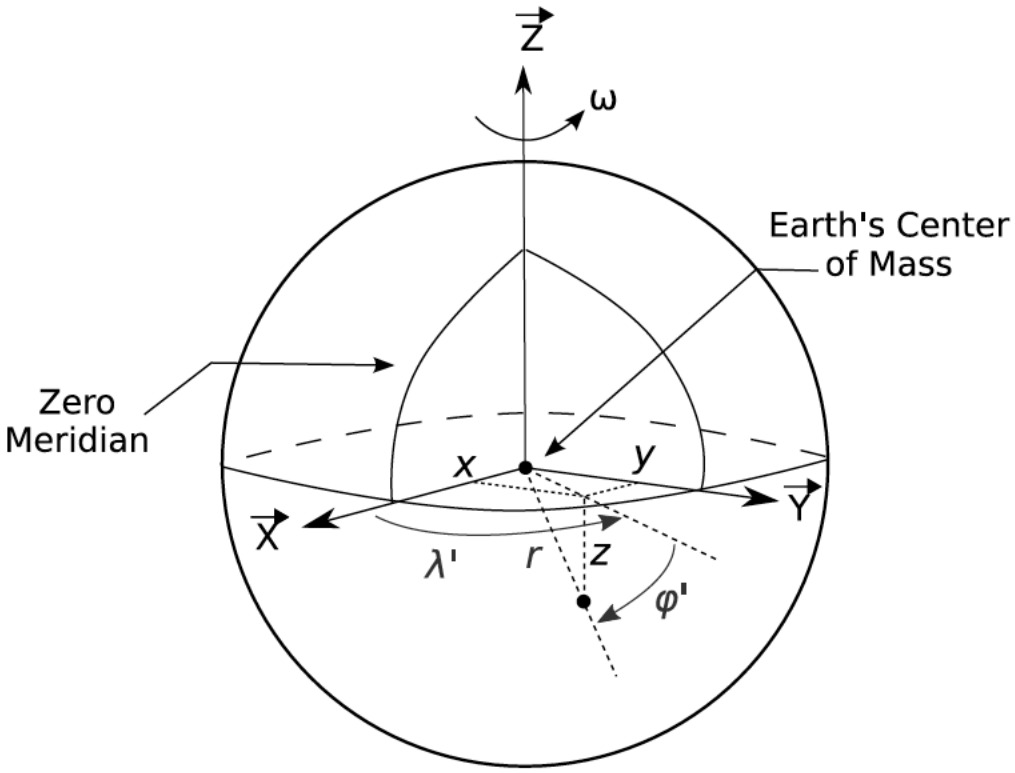


Figure 10. Définition du repère sphérique (Source NIMA report 8350.2 avec modifications)

Les relations suivantes transforment les coordonnées du repère géographique en de nouvelles coordonnées dans le repère sphérique :

- $\rho = \frac{r}{a}$
- $\tan(u) = (1-f)\cdot\tan(\varphi)$
- $\rho\cdot\cos(\varphi') = \cos(u) + \frac{h}{a}\cdot\cos(\varphi)$
- $\rho\cdot\sin(\varphi') = (1-f)\cdot\sin(u) + \frac{h}{a}\cdot\sin(\varphi)$
- $\lambda' = \lambda$

avec :

- ρ : distance au centre de la Terre normalisée par le rayon de la Terre à l'équateur (sans unité) ;
- r : distance au centre de la Terre (sphérique) en mètre ;
- u : angle de correction entre les coordonnées géographiques (GPS) et géocentriques (sphériques) ;
- φ' : latitude géocentrique (sphérique) en degré ;
- λ' : longitude géocentrique (sphérique) en degré ;
- φ : latitude géographique (GPS) en degré ;
- λ : longitude géographique (GPS) en degré ;
- h : altitude géographique (GPS) par rapport à l'ellipsoïde de référence (WGS84) en mètre ;
- a : rayon de la Terre à l'équateur (WGS84) en mètre ;
- f : aplatissement de l'ellipsoïde terrestre (sans dimension).

Q3.3. Donner les relations permettant de calculer les coordonnées sphériques r, φ' et λ' en fonction des coordonnées GPS de la station h, φ et λ. Exprimer les coordonnées cartésiennes x, y et z de la station en fonction de ses coordonnées sphériques r, φ' et λ'.

Q3.4. Compléter dans le document réponse DR5 le corps de la fonction de conversion des coordonnées. L'utilisation des fonctions mathématiques de la libc est encouragée.

Sous-partie 3.4. Détermination du vecteur directeur de la ligne de vue

L'obtention de ce vecteur dirigé de la caméra vers la météorite nécessite l'étude de l'optique visuelle et des dimensions des images numérisées.

Les objectifs *fisheye* ont la particularité d'avoir une distance focale très courte et un angle de vision très grand (ici 180°). L'image obtenue est distordue (voir la figure 2 dans la partie introductive). Cette distorsion est spécifique à l'objectif utilisé.

La distance focale est notée *f*, la distance d'observation de l'objet au centre de l'image *r* et l'angle Θ est l'angle d'observation de l'objet.

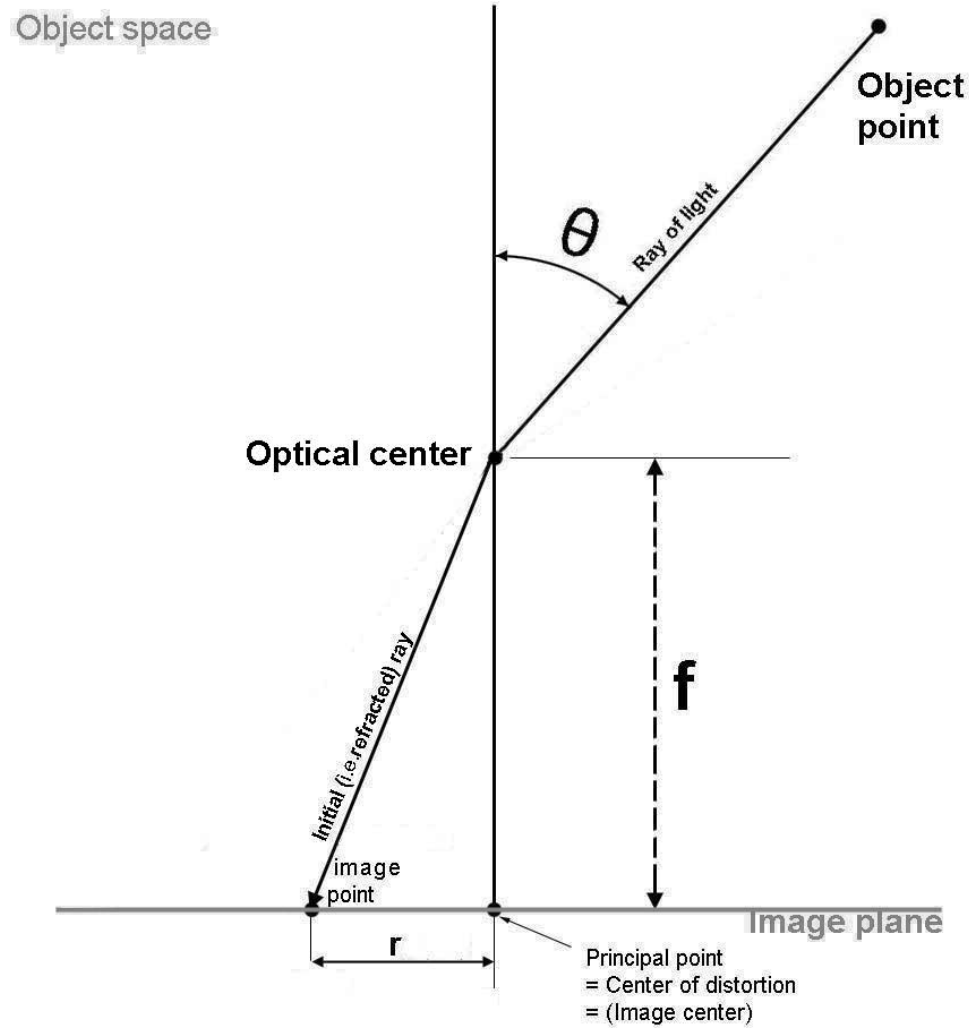


Figure 11. Schéma montrant le trajet d'un rayon avec un objectif optique fisheye

Source http://michel.thoby.free.fr/Fisheye_history_short/Projections/Models_of_classical_projections.html

La fonction de représentation des objectifs fisheye du projet FRIPON est donnée par la loi suivante :

$$r = f.\sin(\theta)$$

En pratique, même si ces fonctions sont fournies par chaque fabricant, les caméras doivent être calibrées afin de prendre en compte et compenser les imperfections propres à la partie optique et à la pose de la caméra. Cette phase de calibration est réalisée une fois celle-ci installée définitivement sur le site d'observation. Elle permet d'aboutir à une *fonction de représentation inverse*, définie comme la transformation de la distance au centre de l'image en un angle Θ. Celle-ci est approximée par un polynôme de degré 9 dont la précision est jugée suffisante. D'autres paramètres physiques sont à prendre en compte et sont réunis dans l'ensemble de données suivant pour chaque caméra appellé « paramètres astronomiques » :

- `x0, y0` : centre de l'image obtenue ;
- `V, S, D, P et Q` : coefficients du polynôme d'ordre 9 limité aux coefficients impairs tel que $P(x) = V \cdot x + S \cdot x^3 + D \cdot x^5 + P \cdot x^7 + Q \cdot x^9$;
- $\theta_x, \theta_y, \theta_z$: angles de positionnement du repère de la caméra par rapport au repère géocentrique sphérique lié à la terre selon chaque axe ;
- `ani, phi` : coefficients de correction du décalage entre la focale de la lentille et le centre de l'image.

Q3.5. Proposer une méthode de calibration de la *fonction de transformation inverse* d'une caméra avec un objectif *fisheye* en étudiant les images obtenues par les caméras observant le ciel. Justifier que tous les coefficients des monomes de puissance paire du polynôme approximant la *fonction de transformation inverse* soient nuls.

Les fonctions calculant les composantes du vecteur directeur d'une ligne de vue en fonction de la position d'un pixel sur l'image sont fournies dans le document réponse DR6.

Q3.6. Commenter sur le document réponse DR6 les différentes étapes de l'algorithme. Définir et tracer le repère sphérique utilisé dans la fonction « *pix_to_cam_vector* ». Justifier l'utilité des rotations effectuées dans la fonction « *pix_to_earth_vector* ».

Q3.7. Ecrire le prototype et une implémentation en langage C de la fonction « *rot_z* » calculant la rotation des coordonnées du vecteur autour de son axe $\vec{z}$.

Sous-partie 3.5. Analyse des images

Les premières étapes du traitement des images (non étudiées) extraient la position centrale (en coordonnées de pixels) de la météorite pour chaque image. Le flux lumineux de la météorite est réparti sur plusieurs pixels (tâche ou halo lumineux), le traitement stocke les coordonnées cartésiennes moyennes de chaque position dans un fichier texte. Un exemple partiel de fichier est donné ci-dessous :

```
1  507.1438 643.815  508.8941 2018-12-22T21:12:01.605
2  335.4258 642.1721 514.4301 2018-12-22T21:12:01.672
3  1294.918 641.3181 517.2942 2018-12-22T21:12:01.706
4  1677.496 637.8014 529.1404 2018-12-22T21:12:01.840
5  23008.71 632.0119 551.5469 2018-12-22T21:12:02.073
6  31629.78 631.23   554.1618 2018-12-22T21:12:02.106
7  34204.68 630.2981 557.5591 2018-12-22T21:12:02.140
8  78016.21 628.4183 564.1457 2018-12-22T21:12:02.206
9  112373.9 626.4995 570.485  2018-12-22T21:12:02.273
```

Les différents champs sont séparés par un espace et représentent les données dans l'ordre :

- numéro de l'enregistrement ;
- brillance totale (luminosité totale des pixels représentant une météorite) ;
- abscisse moyenne de la position (en pixel) ;
- ordonnée moyenne de la position (en pixel) ;
- heure/date de l'image.

L'objectif de cette partie est la mise au point d'une fonction transformant chaque ligne de ce fichier en un objet de type *tDroite* représentant une ligne de vue. Ces objets sont ensuite utilisés par l'algorithme d'*optimisation par essaim particulaire*. Le diagramme des classes est tracé dans le document technique DT8.

Q3.8. Décrire et justifier les relations entre les classes *tDroite, tPoint* et *tVecteur*.

Les questions suivantes visent à charger pour chaque station le fichier texte contenant les résultats, puis pour chaque observation, créer la droite modélisant la ligne de vue d'observation. L'utilisation des méthodes des classes des bibliothèques standards et des classes *tDroite*, *tPoint* et *tVecteur* est obligatoire.

Les trois vecteurs fournis en paramètre de la fonction *calculerLignesDeVue*() du document réponse DR7 sont de tailles identiques. Les chemins des fichiers de données sont contenus dans les éléments du tableau *chemins*, les coordonnées des stations dans le paramètre *coordsCartStations*, les paramètres de calibration (paramètres astronomiques) des stations dans le paramètre *paramsStations*.

Chaque élément du tableau *chemins* correspond à un élément du tableau *coordsCartStations*. Par exemple, le premier chemin est le nom du fichier produit par la première station ayant observé un événement et les coordonnées de cette station sont stockées dans le premier élément du tableau *coordsCartStations.*

Pour chaque fichier, pour chaque ligne, il faut déterminer le vecteur directeur de chaque ligne de vue en convertissant les positions des pixels contenues dans le fichier et l'ajouter à la liste des lignes de vues considérées par l'algorithme d'*optimisation par essaim particulaire*.

La classe *fstream* est détaillée partiellement dans le document technique DT9 ; la classe générique (« *class template* ») *vector* l'est également dans le document technique DT10.

Q3.9. Compléter le premier bloc encadré de code du document réponse DR7 implémentant l'extraction de la position de la météorite de chaque ligne d'un fichier. L'utilisation des méthodes C++ est fortement encouragée (classe fstream).

Q3.10. Compléter le deuxième bloc encadré du document réponse DR7 pour implémenter la conversion des coordonnées de chaque pixel en vecteur directeur de ligne de vue, créer une instance de la classe *tDroite* et l'ajouter au vecteur *lignesDeVue*.

Sous-partie 3.6. Algorithme heuristique d'optimisation de la trajectoire

Cette partie vise à estimer la trajectoire du météore à partir des images des caméras.

Présentation de l'algorithme

Ce type d'algorithme est inspiré du comportement d'un groupe d'oiseaux recherchant de la nourriture ou d'autres animaux ayant un comportement du même genre. Il permet de résoudre en optimisant itérativement un problème numérique.

Les particules représentent les solutions possibles à un problème. Chaque particule a une vitesse de mouvement conditionnée par sa meilleure position et par la meilleure position trouvée par tout l'essaim. Le résultat espéré est la convergence du nuage de particules vers la « meilleure » solution globale.

Il faut bien faire attention au fait que le terme « position » représente une solution possible à un problème. Celle-ci peut être de plusieurs dimensions. En prenant l'exemple simple de la recherche de minimum de la fonction f : $(x,y) \rightarrow x^2+y^2+1$ (situé au point de coordonnées (0,0)), alors chaque particule aura deux dimensions. De même pour le terme « vitesse », celui-ci ne représente pas réellement une vitesse au sens physique, mais le taux de variation d'une des dimensions de la particule. Ainsi, de manière générale, la position représente le vecteur d'état des données d'une particule, et la vitesse représente le vecteur des variations des différentes dimensions de l'état.

Pour le problème présenté, une particule représente une droite avec six dimensions (trois pour la position d'un point de passage et trois pour la direction du vecteur directeur). Il y a également pour chaque particule six vitesses, chacune liée à une dimension de la particule.

Implémentation de l'optimisation par essaim particulaire

Le pseudo-code de l'algorithme est listé dans le document technique DT11, il se décompose en deux parties distinctes : l'initialisation et les itérations (representées par la lettre *k* par la suite).

Il commence par une initialisation de chaque particule. La valeur de chaque dimension du point de la particule est un nombre aléatoire compris dans l'espace de recherche. La vitesse sera initialisée également à une valeur aléatoire.

Les itérations consistent à répéter les étapes suivantes :

- étape 1 : le coût de chaque particule est calculé, il mesure numériquement la qualité de la solution représentée par cette particule. Chaque particule possède sa propre meilleure valeur. Si la mesure actuelle est meilleure que les précédentes de cette particule, alors la particule mémorise cette nouvelle valeur ainsi que sa position (i.e. le vecteur de valeurs correspondantes) ;
- étape 2 : parmi toutes les particules, la meilleure valeur globale de coût est mémorisée. Si pour toutes les valeurs de coût de l'itération actuelle une meilleure valeur est trouvée, alors celle-ci est mémorisée ainsi que sa position ;
- étape 3 (schématisée dans la figure 13) : la nouvelle valeur de vitesse $v_{id}(k+1)$ de chaque dimension d de chaque particule i est calculée selon la relation ci-dessous : $v_{id}(k+1)=w \cdot v_{id}(k)+c_1 \cdot rand_1 \cdot (\widehat{x_{id}}(k)-x_{id}(k))+c_2 . rand_2 . (g_{id}(k)-x_{id}(k))$

Tournez la page S.V.P.

où *:*

- $v_{id}(k)$ est la vitesse de la dimension d de la particule i ;
- w est un facteur « d'inertie » pondérant la vitesse actuelle de la particule. Cette valeur constante est en général fixée à 0.8 dans la littérature ;
- c_1 et c_2 sont deux constantes ajustant les contributions de l'écart entre la position actuelle $x_{id}(k)$ et la meilleure position personnelle $\widehat{x_{id}}(k)$ ainsi que de l'écart entre la position actuelle $x_{id}(k)$ et la meilleure position globale $g_{id}(k)$. Elles sont inférieures à 1 *;*
- $rand_1$ et $rand_2$ sont deux valeurs tirées aléatoirement entre 0 et 1 pour chaque dimension de chaque particule à chaque itération ;

- étape 4 : la position de chaque dimension de chaque particule est mise à jour selon la vitesse selon la relation suivante : $x_{id}(k+1)=x_{id}(k)+v_{id}(k+1)$.

La figure 13 représente l'évolution d'une position en deux dimensions d'une particule pendant une itération de l'algorithme. La nouvelle position de la particule « *Position after update* » est obtenue à partir de la position actuelle « *Position before update* » et d'une évolution dépendant de sa vitesse propre « *Current motion* », de sa meilleure position passée « *Individual best solution* » ainsi que de la meilleure position globale « *Global best candidate solution* ».

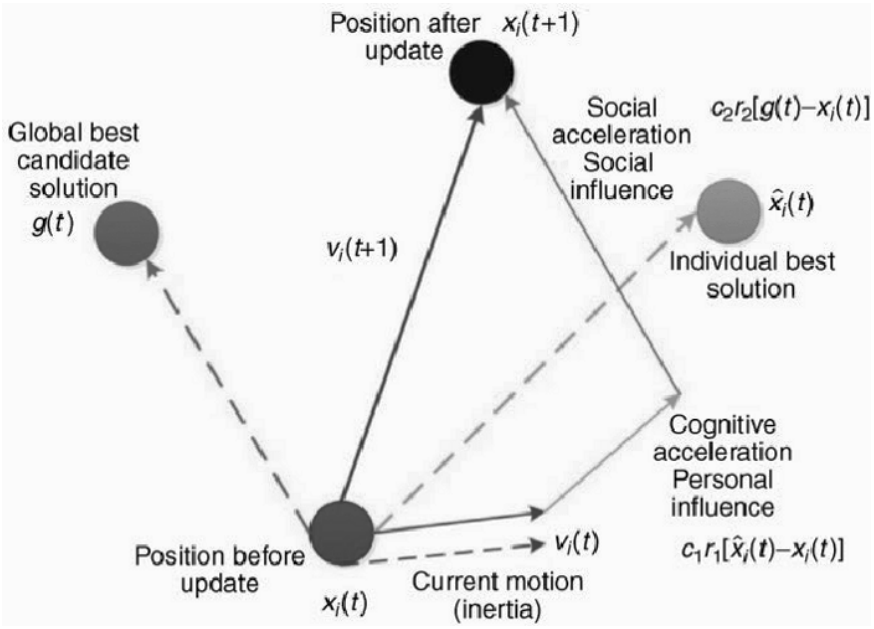


Figure 13. Schématisation de l'évolution itérative de la position d'une particule i (en 2 dimensions)

Source https://medium.com/analytics-vidhya/implementing-particle-swarm-optimization-pso-algorithm-in-python-9efc2eb179a6

Application de l'algorithme au cas de Fripon

Les questions suivantes reposent sur le contenu du document technique DT8. Chaque droite est représentée par six valeurs numériques : trois pour les coordonnées cartésiennes du point de passage et trois pour les coordonnées cartésiennes du vecteur directeur de la droite.

Q3.11. Décrire et justifier le lien entre les classes *tParticule* et *tDroite*. Justifier le rôle des attributs de la classe *tParticule*.

La fonction de calcul du coût est basée sur le calcul de la somme des distances d'une particule (donc d'une droite) avec toutes les lignes de vue (des droites). L'algorithme converge vers une solution où la distance avec toutes les lignes de vue est la plus faible possible. La relation mathématique permettant de calculer la distance entre deux droites D et D' est :

$$d = \frac{\left\| \overrightarrow{AA'}, \vec{u}, \vec{v} \right\|}{\|\vec{u} \wedge \vec{v}\|}$$

avec :

- A est un point appartenant la droite D ;
- A' est un point appartenant à la droite D' ;
- $\vec{u}$ est le vecteur directeur de la droite D ;
- $\vec{v}$ est le vecteur directeur de la droite D' ;
- $\left\| \overrightarrow{AA'}, \vec{u}, \vec{v} \right\|$ est la norme cartésienne du produit mixte de relation $\left(\overrightarrow{AA'} \wedge \vec{u} \right) \cdot \vec{v}$;
- $\|\vec{u} \wedge \vec{v}\|$ est la norme cartésienne du produit vectoriel entre $\vec{u}$ et $\vec{v}$.

Q3.12. En utilisant les attributs et les méthodes disponibles, écrire le code de la méthode *tDroite::calculDistance()*.

Q3.13. Nommer la caractéristique particulière signifiée par le soulignement de 5 attributs de la classe *tParticule*. Justifier ce choix au regard de la description du fonctionnement de l'algorithme.

Q3.14. Ecrire le contenu du fichier entête *tParticule.h*. déclarant la classe *tParticule*.

Le coût d'une particule est calculée par la méthode *tParticule::updateFitness()*. Conformément à la description générale de l'algorithme, cette fonction calcule la somme des distances avec toutes les lignes de vue (contenues dans le vecteur *tabDroites*), et met à jour les variables *bestLocalValue*, *bestLocalPos*, *bestGlobalValue*, *bestGlobalPos* si la distance actuelle est la meilleure.

Q3.15. Écrire le code C++ de la méthode *tParticule::updateFitness()* pour parcourir le tableau *tabDroites*, calculer (à l'aide de la fonction *tParticule::calculDistance*) et accumuler les distances.

La dernière partie de l'algorithme est implémentée par la méthode *tParticule::updatePosition()*. Pour chaque dimension, le calcul de la nouvelle vitesse est suivi d'une mise à jour de la position.

Q3.16. Écrire le code C++ de la méthode *tParticule::updatePosition()* sachant que la fonction *float alea(void)* renvoie une valeur aléatoire entre 0 et 1.

Sous-partie 3.7. Validation du fonctionnement

Q3.17. Valider la fonctionnalité du logiciel et la réponse aux objectifs exprimés. Proposer d'autres utilisations possibles de la trajectoire obtenue.

La phase de « dark flight » est définie par la chute libre de la météorite.

Q3.18. Évaluer les étapes nécessaires à l'obtention de la trajectoire complète du météore pendant la phase « *dark flight* ». Indiquer les phénomènes physiques agissant sur ce déplacement et nommer une méthode de résolution numérique permettant de calculer la position du point de chute.

Document technique DT1. Code de la synchronisation

Ce document technique comporte deux pages.

Déclarations des variables partagées entre les threads

```
mutex *detSignal_mutex;
bool *detSignal;
condition_variable *detSignal_condition;
mutex *frameBuffer_mutex;
```

Code partiel du fil d'exécution du thread d'acquisition

```
void AcqThread::run()
{
  tFrame* newFrame;
  while (!stop)
  {
    if (camera.getNextFrame(newFrame))
    {
      unique_lock verrouCB(*frameBuffer_mutex);
      cirBuf->push(newFrame);
      verrouCB.unlock();

      unique_lock verrouSig(*detSignal_mutex);
      *detSignal = true;
      detSignal_condition->notify_one();
      verrouSig.unlock();
    }
  }
}
```


Tournez la page S.V.P.

Code partiel du fil d'exécution du thread de détection

```
void DetThread::run()
{
  bool eventToComplete = false;
  while(!stop)
  {
    unique_lock verrouSig(*detSignal_mutex);
    while (!*detSignal)
    {
      detSignal_condition->wait(verrouSig);
    }
    *detSignal = false;
    verrouSig.unlock();
```

```
    unique_lock verrouCB(*frameBuffer_mutex);
    tFrame* frame = cirBuf->back();
    verrouCB.unlock();
```

```
    chrono::high_resolution_clock::time_point tref;

    //pDetMthd est un pointeur donnant accès à
    //la fonction de détection.
    //Elle renvoie true si il y a détection d'un météore, false sinon.
    if(pDetMthd->runDetection(lastFrame) && !eventToComplete)
    {
      eventToComplete = true;
      NbDetection++;
      tref = chrono::high_resolution_clock::now();
    }

    if (eventToComplete)
    {
      auto diff = chrono::high_resolution_clock::now() - tref;
      auto diff_ms = chrono::duration_cast<chrono::microseconds>(diff);
      if (diff_ms > param.diff)
      {
        eventToComplete = false;
        // la suite du code est dédiée à la sauvegarde des images de
        // l'événement.
      }
    }
  }
}
```

Document technique DT2. Classe std::condition_variable

Cette annexe contient la documentation partielle de la classe, de ses membres et de son utilisation, issue de https://en.cppreference.com/w/cpp/thread/condition_variable (avec modifications).

Description générale de la classe std::condition_variable

The `condition_variable` class is a synchronization primitive that can be used to block a thread, or multiple threads at the same time, until another thread both modifies a shared variable (the *condition*), and notifies the `condition_variable`.

The thread that intends to modify the variable has to:

1. acquire a `std::mutex`
2. perform the modification while the lock is held
3. execute notify_one on the `std::condition_variable` (the lock does not need to be held for notification)

Even if the shared variable is atomic, it must be modified under the mutex in order to correctly publish the modification to the waiting thread.

Any thread that intends to wait on `std::condition_variable` has to:

1. acquire a `std::unique_lock`, on the same mutex as used to protect the shared variable
2. execute `wait()`. The wait operations atomically release the mutex and suspend the execution of the thread.
3. When the condition variable is notified, a timeout expires, or a spurious wakeup occurs, the thread is awakened, and the mutex is atomically reacquired. The thread should then check the condition and resume waiting if the wake up was spurious.

Condition variables permit concurrent invocation of the `wait()` and `notify_one()` member functions.

The effects of `notify_one()` and `wait()` (unlock+wait, wakeup, and lock) take place in a single total order that can be viewed as modification order of an atomic variable: the order is specific to this individual condition_variable. This makes it impossible for `notify_one()` to, for example, be delayed and unblock a thread that started waiting just after the call to `notify_one()` was made.

Tournez la page S.V.P.

Description de la méthode std::condition_variable::notify_one

If any threads are waiting on `*this`, calling `notify_one` unblocks one of the waiting threads.

Parameters (none)

Return value (none)

Notes:

The notifying thread does not need to hold the lock on the same mutex as the one held by the waiting thread(s); in fact doing so is a pessimization, since the notified thread would immediately block again, waiting for the notifying thread to release the lock. However, some implementations (in particular many implementations of pthreads) recognize this situation and avoid this "hurry up and wait" scenario by transferring the waiting thread from the condition variable's queue directly to the queue of the mutex within the notify call, without waking it up.

Notifying while under the lock may nevertheless be necessary when precise scheduling of events is required, e.g. if the waiting thread would exit the program if the condition is satisfied, causing destruction of the notifying thread's condition_variable. A spurious wakeup after mutex unlock but before notify would result in notify called on a destroyed object.

Description de la méthode std::condition_variable::wait

```
void wait( std::unique_lock& lock );
```

`wait` causes the current thread to block until the condition variable is notified or a spurious wakeup occurs, optionally looping until some predicate is satisfied.

Atomically unlocks `lock`, blocks the current executing thread, and adds it to the list of threads waiting on *this. The thread will be unblocked when `notify_one()` is executed. It may also be unblocked spuriously. When unblocked, regardless of the reason, `lock` is reacquired and `wait` exits. If this function exits via exception, `lock` is also reacquired.
Note that `lock` must be acquired before entering this method, and it is reacquired after `wait(lock)` exits, which means that `lock` can be used to guard access to `pred()`.

Parameters:

`lock` an object of type `std::unique_lock<std::mutex>`, which must be locked by the current thread

Return value: (none)

Notes:

Calling this function if lock.mutex() is not locked by the current thread is undefined behavior.

Calling this function if lock.mutex() is not the same mutex as the one used by all other threads that are currently waiting on the same condition variable is undefined behavior.

Document technique DT3. Classe std::unique_lock

Cette annexe contient la documentation partielle de la classe, de ses membres et de son utilisation, issue de https://en.cppreference.com/w/cpp/thread/unique_lock (avec modifications).

Description générale de la classe std::condition_variable

unique lock is an object that manages a *mutex object* with *unique ownership* in both states: *locked* and *unlocked*.

On construction (or by move-assigning to it), the object acquires a *mutex object*, for whose locking and unlocking operations becomes responsible.

The object supports both states: *locked* and *unlocked*.

This class guarantees an unlocked status on destruction (even if not called explicitly). Therefore it is especially useful as an object with *automatic duration*, as it guarantees the *mutex object* is properly unlocked in case an exception is thrown.

Note though, that the unique_lock object does not manage the lifetime of the *mutex object* in any way: the duration of the *mutex object* shall extend at least until the destruction of the unique_lock that manages it.

Documentation constructeur std::unique_lock::unique_lock

```
explicit unique_lock( mutex_type& m );
```

Constructs a unique_lock with m as the associated mutex. Additionally, locks the associated mutex by calling `m.lock()`. The behavior is undefined if the current thread already owns the mutex except when the mutex is recursive.

Parameters:

`m` mutex to associate with the lock and optionally acquire ownership of.

Documentation méthode std::unique_lock::unlock

```
void unlock();
```

Unlocks the associated mutex and releases ownership.

Parameters: none

Return value: none

Tournez la page S.V.P.

Document technique DT4. « spurious wakeup »

Tiré en partie de l'article de blog http://blog.vladimirprus.com/2005/07/spurious-wakeups.html (avec modifications).

One of the two basic synchronisation primitives in multithreaded programming is called "condition variables". Here's a small example :

```
bool something_happened;
mutex m;
condition_variable c;
void thread1() {
  unique_lock<mutex> lock(m);
  while(!something_happened) {
    c.wait(m);
  }
}
```

```
void thread2() {
  // do lots of work
  unique_lock<mutex> lock(m);
  something_happened = true;
  c.notify_one()
}
```

Here, the call to "c.wait()" unlocks the mutex (allowing the other thread to eventually lock it), and suspends the calling thread. When another thread calls 'notify_one', the first thread wakes up, locks the mutex again (implicitly, inside 'wait'), sees that variable is set to 'true' and goes on. But why do we need the while loop, can't we write:

```
  if (!something_happened)
    c.wait(m);
```

We can't. And the killer reason is that 'wait' can return without any 'notify_one' call. That's called *spurious wakeup* and is explicitly allowed by POSIX. Essentially, return from 'wait' only indicates that the shared data *might* have changed, so that data must be evaluated again.

Okay, so why this is not fixed yet? The first reason is that nobody wants to fix it. Wrapping call to 'wait' in a loop is very desired for several other reasons. But those reasons require explanation, while spurious wakeup is a hammer that can be applied to any first year student without fail.

The second reason is that fixing this is supposed to be hard. Most sources I've seen say that fixing that would require very large overhead on certain architectures. Strangely, no details were ever given, which made me wonder if avoiding spurious wakeups is simple, but all the threading experts secretly decided to tell everybody it's hard.

After asking on comp.programming.thread, I at least know the reason for Linux (thanks to Ben Hutchings). Internally, 'wait' is implemented as a call to the 'futex' system call. Each blocking system call on Linux returns abruptly when the process receives a signal -- because calling signal handler from kernel call is tricky. What if the signal handler calls some other system function? And a new signal arrives? It's easy to run out of kernel stack for a process. Exactly because each system call can be interrupted, when glibc calls any blocking function, like 'read', it does it in a loop, and if 'read' returns EINTR, calls 'read' again.

Can the same trick be used to conditions? No, because the moment we return from 'futex' call, another thread can send us notification. And since we're not waiting inside 'futex', we'll miss the notification. So, we need to return to the caller, and have it reevaluate the predicate. If another thread indeed set it to true, we'll break out of the loop.

Document technique DT5. Configuration réseau avant l'établissement du VPN

Extraits des résultats de la commande "`ifconfig`" :

```
eth0 Link encap:Ethernet HWaddr b8:ae:ed:73:b8:c9
inet addr:192.168.1.13 Bcast:192.168.1.255 Mask:255.255.255.0
inet6 addr: fe80::baae:edff:fe73:b8c9/64 Scope:Link

eth0.2 Link encap:Ethernet HWaddr b8:ae:ed:73:b8:c9
inet addr:10.254.99.99 Bcast:10.254.99.103 Mask:255.255.255.248
inet6 addr: fe80::baae:edff:fe73:b8c9/64 Scope:Link

lo Link encap:Local Loopback
inet addr:127.0.0.1 Mask:255.0.0.0
inet6 addr: ::1/128 Scope:Host
```

Extraits des résultats de la commande "`route -n`":

```
Kernel IP routing table
Destination Gateway Genmask Iface
0.0.0.0 192.168.1.1 0.0.0.0 eth0
10.254.99.96 0.0.0.0 255.255.255.248 eth0.2
192.168.1.0 0.0.0.0 255.255.255.0 eth0
```


Tournez la page S.V.P.

Document technique DT6. Configuration réseau après l'établissement du VPN

Extraits des résultats de la commande "`ifconfig`" :

```
eth0 Link encap:Ethernet HWaddr b8:ae:ed:73:b8:c9
inet addr:192.168.1.13 Bcast:192.168.1.255 Mask:255.255.255.0
inet6 addr: fe80::baae:edff:fe73:b8c9/64 Scope:Link

eth0.2 Link encap:Ethernet HWaddr b8:ae:ed:73:b8:c9
inet addr:10.254.99.99 Bcast:10.254.99.103 Mask:255.255.255.248
inet6 addr: fe80::baae:edff:fe73:b8c9/64 Scope:Link

lo Link encap:Local Loopback
inet addr:127.0.0.1 Mask:255.0.0.0
inet6 addr: ::1/128 Scope:Host

tun0 Link encap:UNSPEC
inet addr:10.8.0.218 P-t-P:10.8.0.217 Mask:255.255.255.255
```

Extraits des résultats de la commande "`route -n`":

```
Kernel IP routing table
Destination Gateway Genmask Iface
0.0.0.0 192.168.1.1 0.0.0.0 eth0
10.8.0.0 10.8.0.217 255.255.0.0 tun0
10.8.0.217 0.0.0.0 255.255.255.255 tun0
10.254.99.96 0.0.0.0 255.255.255.248 eth0.2
192.168.1.0 0.0.0.0 255.255.255.0 eth0
212.211.132.250 10.8.0.217 255.255.255.255 tun0
217.196.149.233 10.8.0.217 255.255.255.255 tun0
```

Document technique DT7. Code du client VPN

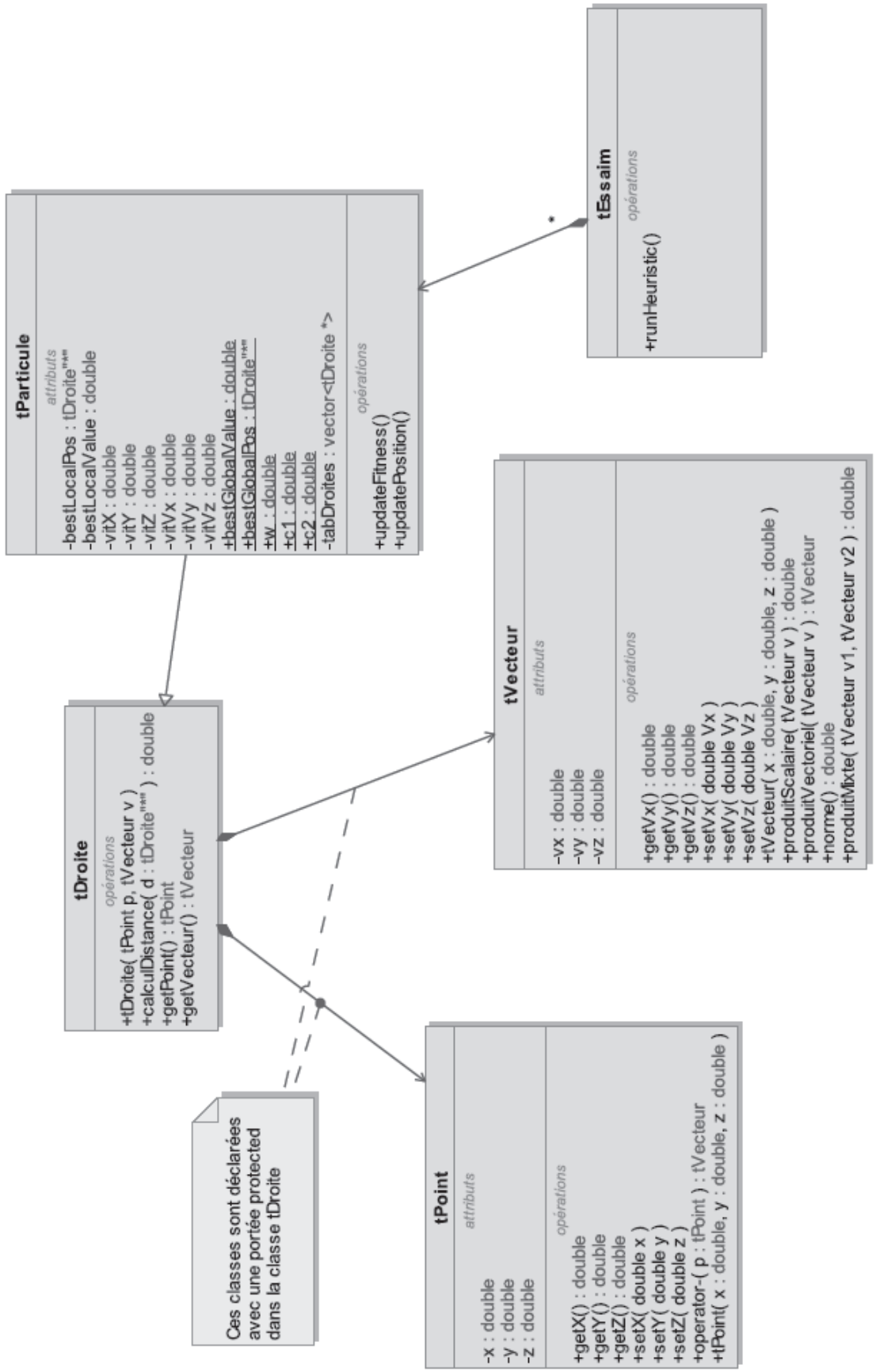
Ce document technique comporte deux pages. Les numéros de ligne sont indiqués dans la colonne de gauche. Le code suivant est sous licence GPL.

```
3199 int main(int argc, char **argv)
3200 {
3201   int do_load_balance;
3202   const uint8_t hex_test[] = { 0, 1, 2, 3 };
3203   struct sa_block oursa[1];
3204   struct sa_block *s = oursa;
3205
3206   test_pack_unpack();
3207 #if defined(__CYGWIN__)
3208   gcry_control(GCRYCTL_SET_THREAD_CBS, &gcry_threads_pthread);
3209 #endif
3210   gcry_check_version("1.1.90");
3211   gcry_control(GCRYCTL_INIT_SECMEM, 16384, 0);
3212   group_init();
3213
3214   memset(s, 0, sizeof(*s));
3215   s->ipsec.encap_mode = IPSEC_ENCAP_TUNNEL;
3216   s->ike.timeout = 1000; /* 1 second */
3217
3218   do_config(argc, argv);
3219
3220   DEBUG(1, printf("\nvpnc version " VERSION "\n"));
3221   hex_dump("hex_test", hex_test, sizeof(hex_test), NULL);
3222
3223   DEBUGTOP(2, printf("S1 init_sockaddr\n"));
3224   init_sockaddr(&s->dst, config[CONFIG_IPSEC_GATEWAY]);
3225   init_sockaddr(&s->opt_src_ip, config[CONFIG_LOCAL_ADDR]);
3226   DEBUGTOP(2, printf("S2 make_socket\n"));
3227   s->ike.src_port = atoi(config[CONFIG_LOCAL_PORT]);
3228   s->ike.dst_port = ISAKMP_PORT;
3229   s->ike_fd = make_socket(s, s->ike.src_port, s->ike.dst_port);
3230   DEBUGTOP(2, printf("S3 setup_tunnel\n"));
3231   setup_tunnel(s);
3232
3233   do_load_balance = 0;
3234   do {
3235     DEBUGTOP(2, printf("S4 do_phase1_am\n"));
3236     do_phase1_am(config[CONFIG_IPSEC_ID],
                    config[CONFIG_IPSEC_SECRET], s);
3237     DEBUGTOP(2, printf("S5 do_phase2_xauth\n"));
3238     /* FIXME: Create and use a generic function in supp.[hc] */
3239     if (s->ike.auth_algo >= IKE_AUTH_HybridInitRSA)
3240       do_load_balance = do_phase2_xauth(s);
3241     DEBUGTOP(2, printf("S6 do_phase2_config\n"));
3242     if ((opt_vendor == VENDOR_CISCO) && (do_load_balance == 0))
3243       do_load_balance = do_phase2_config(s);
```


Tournez la page S.V.P.

```
3244    } while (do_load_balance);
3245    DEBUGTOP(2, printf("S7 setup_link (phase 2 + main_loop)\n"));
3246    DEBUGTOP(2, printf("S7.0 run interface setup script\n"));
3247    config_tunnel(s);
3248    do_phase2_qm(s);
3249    DEBUGTOP(2, printf("S7.9 main loop (receive and transmit ipsec
packets)\n"));
3250    vpnc_doit(s);
3251
3252    /* Tear down phase 2 and 1 tunnels */
3253    send_delete_ipsec(s);
3254    send_delete_isakmp(s);
3255
3256    /* Cleanup routing */
3257    DEBUGTOP(2, printf("S8 close_tunnel\n"));
3258    close_tunnel(s);
3259    s_atexit_sa = NULL;
3260
3261    /* Free resources */
3262    DEBUGTOP(2, printf("S9 cleanup\n"));
3263    cleanup(s);
3264
3265    return 0;
3266 }
```

Document technique DT8. Diagramme de classes



Tournez la page S.V.P.

classe tPoint

- attributs
    - *x, y et z* : coordonnées du point selon l'axe (O,$\vec{x}$), (O,$\vec{y}$) et (O,$\vec{z}$).
- méthodes
    - toutes les méthodes préfixées par *get* permettent de récupérer la coordonnée correspondante de l'axe correspondant ;
    - toutes les méthodes préfixées par s*et* permettent de fixer la coordonnée correspondante de l'axe correspondant.

classe tVecteur

- attributs
    - *vx, vy et vz* : coordonnées du vecteur selon l'axe (O,$\vec{x}$), (O,$\vec{y}$) et (O,$\vec{z}$).
- méthodes
    - toutes les méthodes préfixées par *get* permettent de récupérer la coordonnée de l'axe correspondant ;
    - toutes les méthodes préfixées par *set* permettent de fixer la coordonnée de l'axe correspondant ;
    - *double produitScalaire(tVecteur v)* : calcul le produit scalaire entre le vecteur courant (*this*) le vecteur passé en paramètre ;
    - *double produitVectoriel(tVecteur v)* : calcule le produit vectoriel entre le vecteur courant (*this*) et le vecteur passé en paramètre ;
    - *double norme()* : renvoie la norme euclidienne du vecteur ;
    - *double produitMixte(tVecteur v1, tVecteur v2)* : renvoie la valeur du produit mixte entre le vecteur courant (this – représenté ici v3) et les vecteurs v1 et v2 passés en paramètres tel que le produit mixte est égale à $\left(\overrightarrow{v3} \wedge \overrightarrow{v1}\right) \cdot \overrightarrow{v2}$.

classe tDroite

- attributs
    - voir question **Q.3.9** sur la justification des liens.
- méthodes
    - *tPoint getPoint()* : récupère une copie du point caractéristique de la droite ;
    - tVecteur getVecteur() : récupère une copie du vecteur caractéristique de la droite ;
    - double calculDistance(tDroite* d) : calcule la distance entre la droite courante et la droite *d* en paramètre (code à fournir depuis la question **Q.3.13**).

classe tParticule

- attributs
    - *vitX, vitY,…, vitVy, vitVz* : permettent de stocker les vitesses des attributs x,y,…,vy,vz ;
    - *bestLocalValue* : meilleure valeur calculée par la particule ;
    - *bestLocalPos* : meilleure droite trouvée par la particule ;
    - *tabDroites* : objet de type vector (de la standard library) permettant de stocker les adresses de toutes les lignes de vue (donc des droites) des caméras ;
    - bestGlobalPos : meilleure droite trouvée par l'ensemble des particules ;
    - bestGlobalValue : meilleure valeur calculée par l'ensemble des particules ;
    - w, c1, c2 : paramètres fixés de l'algorithme.
- méthodes
    - updateFitness() : méthode de calcul de la nouvelle valeur de *fitness* (coût).
    - updatePosition() : méthode de calcul de la nouvelle position.

Tournez la page S.V.P.

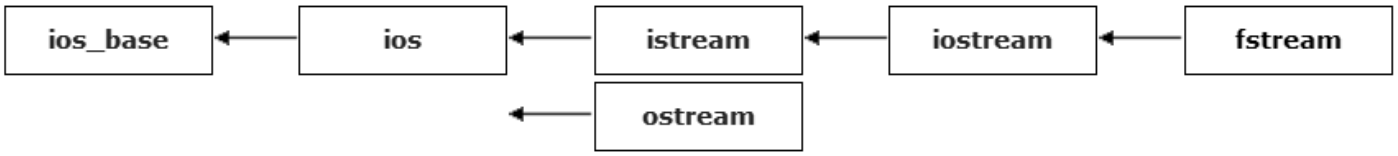
Document technique DT9. Documentation de la classe std:fstream et des classes hérités

Cette annexe contient la documentation partielle de la classe et de ses membres en pages suivantes. Issue de http://www.cplusplus.com/reference/fstream/fstream/ (avec modif.).

Uniquement la lecture d'un fichier est requise, les membres des classes « *iostream* » et « *ostream* » gérant les écritures ont été omis.

std::fstream Input/output file stream class

Class diagram:



Input/output stream class to operate on files.
File streams are associated with files either on construction, or by calling member `open`.

Public member functions

`(constructor)`	Construct object and optionally open file (public member function)
`open`	Open file (public member function)
`is_open`	Check if a file is open (public member function)
`close`	Close file (public member function)

Public member functions inherited from `istream`

`operator>>`	Extract formatted input (public member function)
`ignore`	Extract and discard characters (public member function)

Public member functions inherited from `ios`

`eof`	Check whether eofbit is set (public member function)

public member function std::fstream::fstream (constructor)

(1)
```
fstream();
```

(2)
```
explicit fstream (const char* filename,
                  ios_base::openmode mode=ios_base::in|ios_base::out);
explicit fstream (const string& filename,
                  ios_base::openmode mode=ios_base::in|ios_base::out);
```

Construct object and optionally open file

Constructs an `fstream` object:

(1) default constructor

Constructs an `fstream` object that is not associated with any file.

(2) initialization constructor

Constructs a `fstream` object, initially associated with the file identified by its first argument (*filename*), open with the mode specified by *mode*.

Arguments :

`filename` A string representing the name of the file to be opened. Specifics about its format and validity depend on the library implementation and running environment.

`mode` Flags describing the requested input/output mode for the file. This is an object of the bitmask member type `openmode` that consists of a combination of the following member constants:

member constant stands for access :

`in`	**in**put	File open for reading: the *internal stream buffer* supports input operations.
`out`	**out**put	File open for writing: the *internal stream buffer* supports output operations.
`binary`	**binary**	Operations are performed in binary mode rather than text.
`ate`	**at e**nd	The *output position* starts at the end of the file.
`app`	**app**end	All output operations happen at the end of the file, appending to its existing contents.
`trunc`	**trunc**ate	Any contents that existed in the file before it is open are discarded.

These flags can be combined with the bitwise OR operator (`|`).

Tournez la page S.V.P.

public member function std::istream::operator>>

arithmetic types

```
istream& operator>> (bool& val);
istream& operator>> (short& val);
istream& operator>> (unsigned short& val);
istream& operator>> (int& val);
istream& operator>> (unsigned int& val);
istream& operator>> (long& val);
istream& operator>> (unsigned long& val);
istream& operator>> (long long& val);
istream& operator>> (unsigned long long& val);
istream& operator>> (float& val);
istream& operator>> (double& val);
istream& operator>> (long double& val);
istream& operator>> (void*& val);
```

This operator (`>>`) applied to an input stream is known as *extraction operator*.

Extracts and parses characters sequentially from the stream to interpret them as the representation of a value of the proper type, which is stored as the value of *val*. Leading whitespace characters are automatically ignored.

Parameters :

`val` Object where the value that the extracted characters represent is stored.

Return Value :

The `istream` object (`*this`).

The extracted value or sequence is not returned, but directly stored in the variable passed as argument.

public member function std::istream::ignore

```
istream& ignore (streamsize n = 1, int delim = EOF);
```

Extracts characters from the input sequence and discards them, until either *n* characters have been extracted, or one compares equal to *delim*.

The function also stops extracting characters if the *end-of-file* is reached.

Parameters :

`n` Maximum number of characters to ignore.

`Delim` Explicit *delimiting character*: The operation of extracting successive characters stops as soon as the next character to extract compares equal to this.

Return Value :

The `istream` object (`*this`).

public member function std::ios::eof

```
bool eof() const;
```

Returns `true` if the `eofbit` *error state flag* is set for the stream.

This flag is set by all standard input operations when the End-of-File is reached in the sequence associated with the stream.

Note that the value returned by this function depends on the last operation performed on the stream (and not on the next).

Operations that attempt to read at the *End-of-File* fail. This function can be used to check whether the failure is due to reaching the *End-of-File* or to some other reason.

Parameters : none

Return Value :

`true` if the stream's `eofbit` *error state flag* is set (which signals that the End-of-File has been reached by the last input operation).

`false` otherwise.

constant EOF

It is a macro definition of type `int` that expands into a negative integral constant expression (generally, `-1`).

It is used as the value returned by several functions in header `<cstdio>` to indicate that the End-of-File has been reached or to signal some other failure conditions.

It is also used as the value to represent an invalid character.

Tournez la page S.V.P.

Document technique DT10. Documentation de la classe std:vector et des classes héritées

Cette annexe contient la documentation partielle de la classe et de ses membres en pages suivantes. Issue de http://www.cplusplus.com/reference/vector/vector/ (avec modifications).

class template std::vector

```
template < class T, class Alloc = allocator<T> > class vector;
```

Vectors are sequence containers representing arrays that can change in size.

Internally, vectors use a dynamically allocated array to store their elements. This array may need to be reallocated in order to grow in size when new elements are inserted, which implies allocating a new array and moving all elements to it. This is a relatively expensive task in terms of processing time, and thus, vectors do not reallocate each time an element is added to the container.

Instead, vector containers may allocate some extra storage to accommodate for possible growth, and thus the container may have an actual capacity greater than the storage strictly needed to contain its elements (i.e., its size).

Template parameters :

`T`	Type of the elements. Only if `T` is guaranteed to not throw while moving, implementations can optimize to move elements instead of copying them during reallocations. Aliased as member type `vector::value_type`.
`Alloc`	Type of the allocator object used to define the storage allocation model. By default, the allocator class template is used, which defines the simplest memory allocation model and is value-independent. Aliased as member type `vector::allocator_type`.

Member functions :

`(constructor)`	Construct vector (public member function)
`(destructor)`	Vector destructor (public member function)
`size`	Return size (public member function)
`operator[]`	Access element (public member function)
`push_back`	Add element at the end (public member function)

public member function std::vector::vector

default (1)
```
explicit vector ();
explicit   vector   (const   allocator_type&   alloc   =
                              allocator_type());
```

fill (2)
```
explicit vector (size_type n);
         vector (size_type n, const value_type& val,
                 const     allocator_type&     alloc     =
                           allocator_type());
```

Constructs a vector, initializing its contents depending on the constructor version used:

(1) empty container constructor (default constructor)

Constructs an empty container, with no elements.

(2) fill constructor

Constructs a container with *n* elements. Each element is a copy of `val` (if provided).

public member function std::vector::~vector

```
~vector();
```

Destroys the container object.

This calls allocator_traits::destroy on each of the contained elements, and deallocates all the storage capacity allocated by the vector using its allocator.

public member function std::vector::size

```
size_type size() const noexcept;
```

Returns the number of elements in the vector.

This is the number of actual objects held in the vector, which is not necessarily equal to its storage capacity.

Parameters : none

Return Value :

The number of elements in the container.

Member type `size_type` is an unsigned integral type.

Tournez la page S.V.P.

public member function std::vector::operator[]

```
      reference operator[] (size_type n);
const_reference operator[] (size_type n) const;
```

Returns a reference to the element at position `n` in the vector container.

Portable programs should never call this function with an argument *n* that is *out of range*, since this causes *undefined behavior*.

Parameters :

`n` Position of an element in the container. First element has a position of `0` (not `1`). Member type `size_type` is an unsigned integral type.

Return value :

The element at the specified position in the vector.

If the vector object is const-qualified, the function returns a `const_reference`. Otherwise, it returns a `reference`.

Member types `reference` and `const_reference` are the reference types to the elements of the container.

public member function std::vector::push_back

```
void push_back (const value_type& val);
void push_back (value_type&& val);
```

Adds a new element at the end of the vector, after its current last element. The content of `val` is copied (or moved) to the new element.

This effectively increases the container size by one, which causes an automatic reallocation of the allocated storage space if -and only if- the new vector size surpasses the current vector capacity.

Parameters :

`val` Value to be copied (or moved) to the new element. Member type `value_type` is the type of the elements in the container, defined in vector as an alias of its first template parameter (`T`).

Return value : none

Document technique DT11. Pseudo-code de l'optimisation par essaim particulaire

//initialisation

Pour chaque particule *i*

Pour chaque dimension *d* de la particule

Initialisation de la position x_{id} aléatoirement dans l'espace de recherche

Initialisation de la vitesse v_{id} aléatoirement

Fin Pour

Fin Pour

Itération k=1

//partie principale de l'algorithme

Faire

Pour chaque particule *i*

Calculer la valeur de fitness (mesure)

Si la valeur de fitness est meilleure que la meilleure valeur enregistrée

Mémoriser cette valeur de fitness comme la meilleure calculée

Mémoriser la position associée

Fin Si

Fin Pour

Rechercher et mémoriser la meilleure valeur de fitness ainsi que la position associée parmi toutes les particules

Pour chaque particule *i*

Pour chaque dimension *d*

Calculer la nouvelle valeur de vitesse selon la relation :

$v_{id}(k+1)=w.v_{id}(k)+c_1.rand_1.(\widehat{x}_{id}-x_{id})+c_2.rand_2.(g_{id}-x_{id})$ *(les différents éléments de la relation sont décrits dans le sujet.)*

Mettre à jour la nouvelle position de la particule selon la relation :

$x_{id}(k+1)=x_{id}(k)+v_{id}(k+1)$

Fin Pour

Fin Pour

k=k+1

Tant que le maximum d'itérations ou un certain critère n'est pas atteint

Modèle CMEN-DOC v2 ©NEOPTEC

Nom de famille :
(Suivi, s'il y a lieu, du nom d'usage)

Prénom(s) :

Numéro Inscription :

(Le numéro est celui qui figure sur la convocation ou la feuille d'émargement)

Né(e) le : ☐☐ / ☐☐ / ☐☐☐☐

(Remplir cette partie à l'aide de la notice)

Concours / Examen : …….. **Section/Spécialité/Série :** ……

Epreuve : …….. **Matière :** …….. **Session :** ……..

CONSIGNES

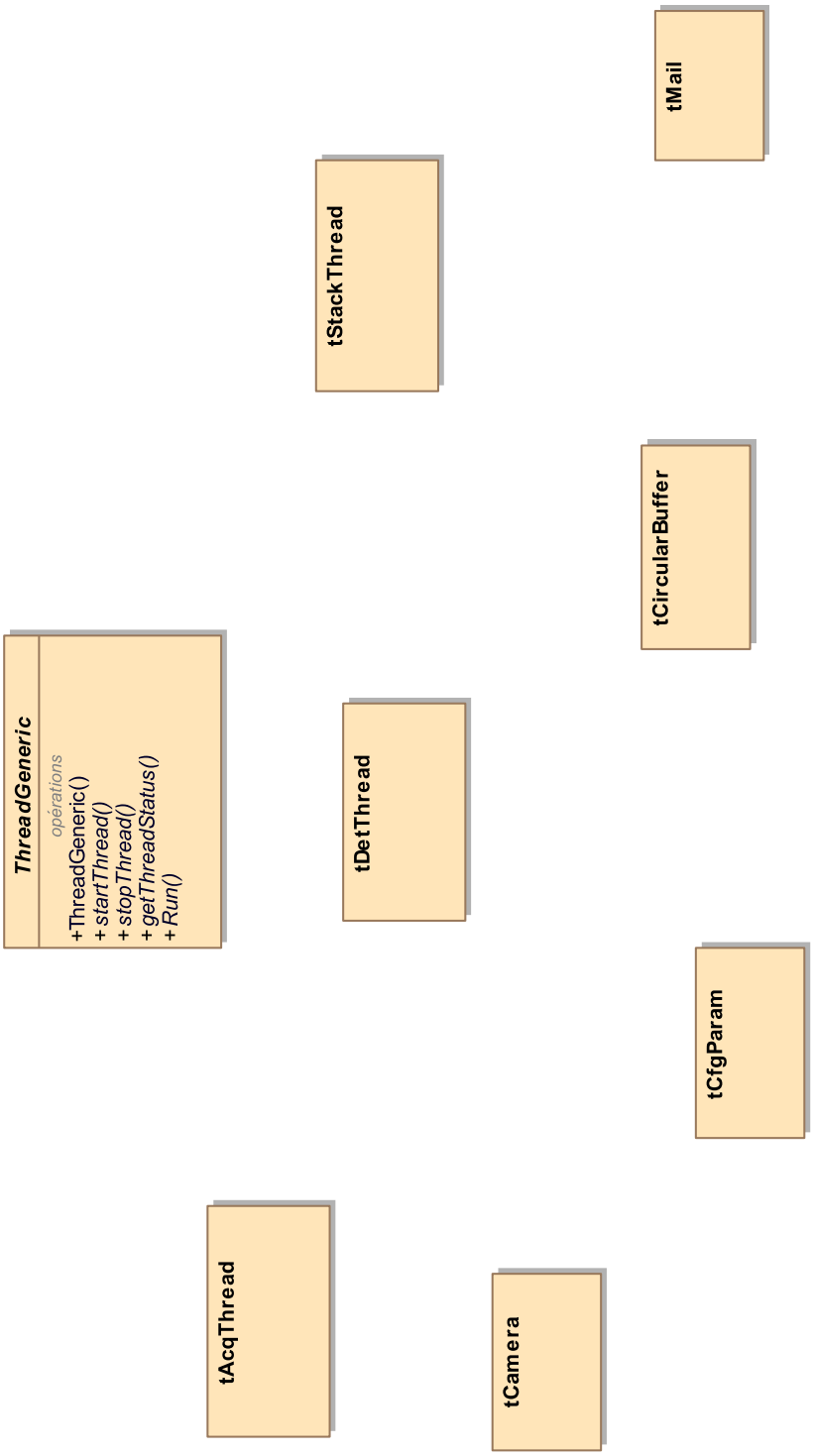
- *Remplir soigneusement, sur CHAQUE feuille officielle, la zone d'identification en MAJUSCULES.*
- *Ne pas signer la composition et ne pas y apporter de signe distinctif pouvant indiquer sa provenance.*
- *Numéroter chaque PAGE (cadre en bas à droite de la page) et placer les feuilles dans le bon sens et dans l'ordre.*
- *Rédiger avec un stylo à encre foncée (bleue ou noire) et ne pas utiliser de stylo plume à encre claire.*
- *N'effectuer aucun collage ou découpage de sujets ou de feuille officielle. Ne joindre aucun brouillon.*

DR1 à DR4

Tous les documents réponses sont à rendre, même non complétés.

NE RIEN ECRIRE DANS CE CADRE

Document réponse DR1. (question Q1.5)



Document réponse DR2. (question Q1.6)

variable partagée PLEIN de type sémaphore initialisée à 0.

variable partagée VIDE de type sémaphore initialisée à N.

variable partagée ACCES de type mutex initialement déverrouillée.

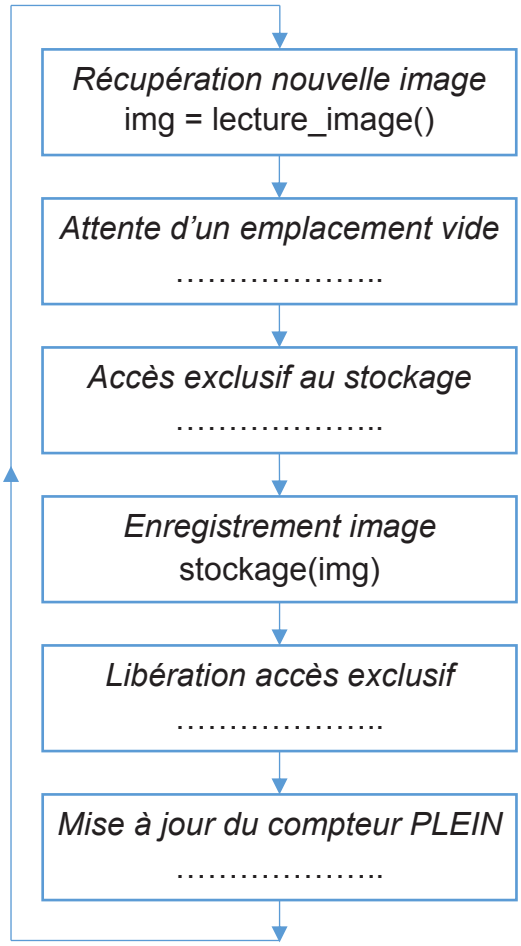
pour chaque thread variable locale img de type image, initialement vide.

Les primitives P() et V() sont décrites dans le questionnement. Par exemple P(ACCES) verrouille le mutex ACCES. V(VIDE) incrémente le sémaphore VIDE.

Compléter les pointillés.

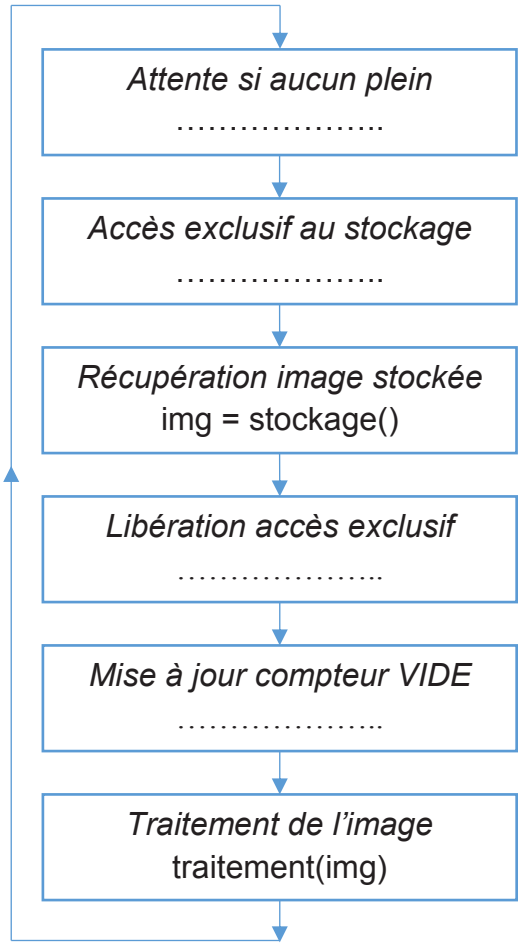
Boucle du thread d'acquisition

(producteur)



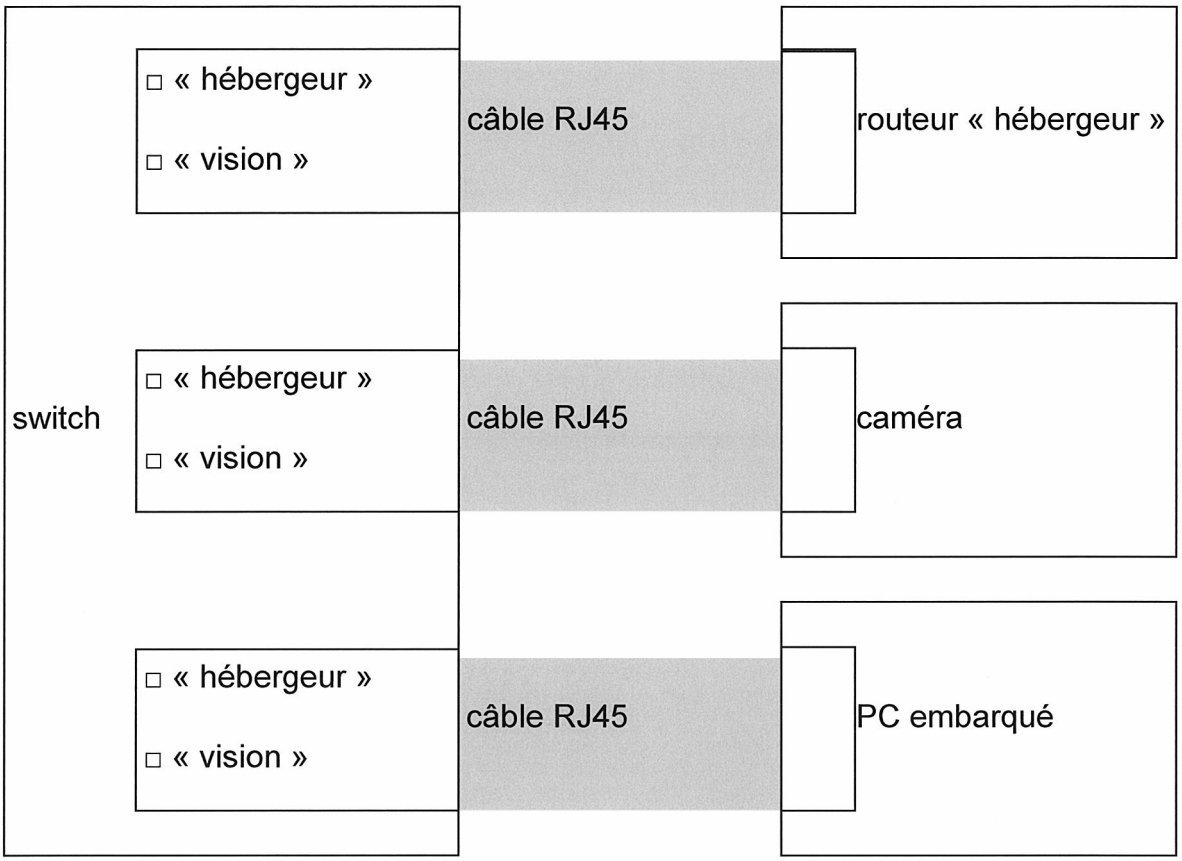
Boucle du thread de détection

(consommateur)

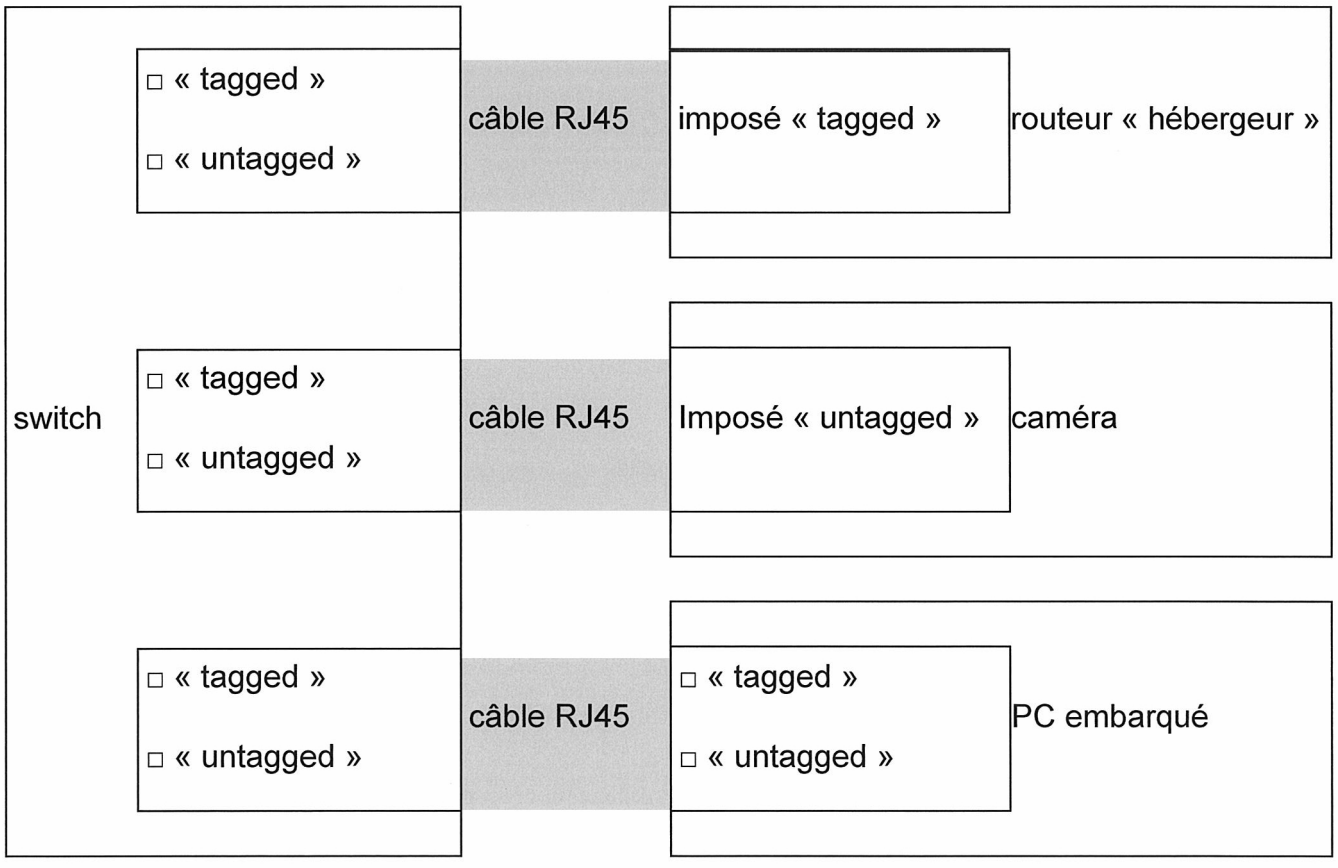


Tournez la page S.V.P.

Document réponse DR3. (question Q2.1)



Document réponse DR4. (question Q2.2)



Modèle CMEN-DOC v2 ©NEOPTEC

Nom de famille :
(Suivi, s'il y a lieu, du nom d'usage)

Prénom(s) :

Numéro Inscription :

(Le numéro est celui qui figure sur la convocation ou la feuille d'émargement)

Né(e) le : / /

(Remplir cette partie à l'aide de la notice)

Concours / Examen : …….. **Section/Spécialité/Série :** ……

Epreuve : …….. **Matière :** …….. **Session :** ……..

CONSIGNES

- *Remplir soigneusement, sur CHAQUE feuille officielle, la zone d'identification en MAJUSCULES.*
- *Ne pas signer la composition et ne pas y apporter de signe distinctif pouvant indiquer sa provenance.*
- *Numéroter chaque PAGE (cadre en bas à droite de la page) et placer les feuilles dans le bon sens et dans l'ordre.*
- *Rédiger avec un stylo à encre foncée (bleue ou noire) et ne pas utiliser de stylo plume à encre claire.*
- *N'effectuer aucun collage ou découpage de sujets ou de feuille officielle. Ne joindre aucun brouillon.*

DR5 à DR7

Tous les documents réponses sont à rendre, même non complétés.

NE RIEN ECRIRE DANS CE CADRE

Document réponse DR5. (question Q3.4)

L'utilisation des fonctions mathématiques de la libc est encouragée.

Les paramètres de la fonction sont :

- `longitude` : longitude géographique d'un point (en degré) ;
- `latitude` : latitude géographique d'un point (en degré) ;
- `h` : altitude géographique d'un point (en mètre) ;
- `coordCart` : pointeur de type double vers le premier élément d'un tableau de trois éléments représentant les coordonnées cartésiennes du point (en mètre).

```
void calculCoordonnees(double longitude, double latitude,
double h, double* coordCart)
{
  // en utilisant le système géodésique WGS84
  double a=6378137; //en mètre
  double f=1/298.257222;
  // autres déclarations à suivre
  // la constante M_PI est disponible
```

Document réponse DR6. (question Q3.6)

Apporter des commentaires au code ci-dessous.

La fonction « *pix_to_earth_vector* » convertit les coordonnées d'un pixel d'une image de la caméra en un vecteur d'une ligne de vue dans le repère cartésien de la Terre. Ses paramètres sont :

- `x, y` : coordonnées de la météorite dans l'image ;
- `para_astro` : tableau de constantes de calibrations (paramètres astronomiques) ;
- `output` : coordonnées du vecteur directeur de la ligne de vue.

```
void pix_to_earth_vector(double x, double y,
                         double * para_astro, double * output)
{
  pix_to_cam_vector(output, x, y, para_astro[0], para_astro[1],
                    para_astro[2], para_astro[3], para_astro[4],
                    para_astro[5], para_astro[6], para_astro[10],
                    para_astro[11]);
  rot_x(output, para_astro[7]);
  rot_y(output, para_astro[8]);
  rot_z(output, para_astro[9]);
}
```

La fonction « *pix_to_cam_vector* » convertit les coordonnées d'un pixel d'une image de la caméra en direction de vue dans le repère cartésien de la caméra. Ses paramètres sont :

- `coord` : coordonnées du vecteur directeur de la ligne de vue ;
- `x, y` : coordonnées de la météorite dans l'image ;
- `x0, y0` : centre de l'image ;
- `V, S, D, P et Q` : coefficient du polynôme d'ordre 9 ;
- `ani, phi` : coefficients de correction liés à l'objectif.

```
void pix_to_cam_vector(double * coord, double x, double y,
                       double x0, double y0,
                       double V, double S, double D, double P,
                       double Q, double ani, double phi )
{
  x = x-x0;
  y = y-y0;
  double R = sqrt(x*x+y*y);
  double A; //angle azimutal (angle entre le vecteur et l'axe x)
  double e; //angle zénithal (angle entre le vecteur et l'axe z)
  if( R==0. ) { A = 0.; } else { A = atan2(y,x); }
  R = R*(1.+ ani*sin(A+phi))/1000.;
  e = R*(V + R*R*(S + R*R*(D + R*R*(P + Q*R*R))));
  coord[0] = -cos(A)*sin(e);
  coord[1] = sin(A)*sin(e);
  coord[2] = cos(e);
}
```


Tournez la page S.V.P.

Document réponse DR7. (questions 3.9 et 3.10)

La fonction *vector<tDroite*> calculerLignesDeVue()* prend en paramètres trois tableaux ayant le même nombre d'éléments (chaque élément représente les données d'une station) :

- la variable *chemins* ;
- une variable de type *vector<double*> coordsCartStations* : vecteur contenant des pointeurs de type *double* représentant des tableaux de 3 coordonnées. Chaque élément correspond aux coordonnées cartésiennes d'une station ;
- une variable de type *vector<double*> paramsStations :* chaque élément du vecteur est un tableau de type *double** donnant les douze paramètres astronomiques d'une station.

```
vector<tDroite*> calculerLignesDeVue(
              const vector<string>& chemins,
              const vector<double*>& coordsCartStations),
              const vector<double*>& paramsStations)
{
  vector<tDroite*> lignesDeVue;
  // pour toutes les lignes de tous les fichiers
```

```
    fstream fichier


      // lecture des valeurs sur la ligne
      int index;
      float brillance, posX, posY;
```

```
      // vecteur directeur pour cet enregistrement
      double vecteurDirecteur[3];
      pix_to_earth_vector(posX, posY, paramStations[i],
                          vecteurDirecteur);
      // création des objets tPoint, tVecteur et tDroite
      tPoint* point = new tPoint(coordsCartStations[i][0],
                                 coordsCartStations[i][1],
                                 coordsCartStations[i][2]);
      tVecteur* vec


      tDroite* droite
      // ajout de la droite à la liste des résultats
```

```
  return lignesDeVue; } //Fin de la méthode
```