MINISTÈRE
DE L'ÉDUCATION
NATIONALE
Liberté
Égalité
Fraternité

EAE SIN 2

SESSION 2025

AGREGATION
CONCOURS EXTERNE

Section : SCIENCES INDUSTRIELLES DE L'INGÉNIEUR

Option : SCIENCES INDUSTRIELLES DE L'INGÉNIEUR ET INGÉNIERIE INFORMATIQUE

MODÉLISATION D'UN SYSTÈME, D'UN PROCÉDÉ OU D'UNE ORGANISATION

Durée : 6 heures

Calculatrice autorisée selon les modalités de la circulaire du 17 juin 2021 publiée au BOEN du 29 juillet 2021.

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

Il appartient au candidat de vérifier qu'il a reçu un sujet complet et correspondant à l'épreuve à laquelle il se présente.

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.
Le fait de rendre une copie blanche est éliminatoire

Tournez la page S.V.P.

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	102	2680

IMPRIMERIE NATIONALE – 25 0105 – D'après documents fournis

Étude dysfonctionnelle d'un système de trains d'atterrissage

FIGURE 1 – Airbus A380 avec ses 5 trains d'atterrissage comportant 22 roues (un à 2 roues, deux à 4 roues, deux à 6 roues), et Airbus A400M avec ses 3 trains d'atterrissage comportant 14 roues (un à 2 roues, deux à 6 roues) : deux solutions techniques pour deux ensembles d'exigences différents
Photos copyright Adrien Daste / Safran

I Contexte et système étudié

I.1 Contexte de l'industrie aéronautique et conception de nouveaux avions

La progression du trafic aérien international est importante et, conjuguée au vieillissement des avions actuels (un avion ayant une durée de vie de 20 à 30 ans), permet de chiffrer le besoin en nouveaux avions commerciaux à plus de 40 000 exemplaires d'ici à l'année 2042.
L'augmentation des normes environnementales, économiques et sociétales rendent une grande partie des avions actuels obsolètes. Les constructeurs aéronautiques doivent renouveler leurs catalogues en mettant à niveau leurs modèles actuels, ou en développant de nouveaux avions, de sorte qu'ils soient plus économes en carburant. Ces améliorations sont rendues possibles par l'utilisation de motorisations optimisées, mais aussi de nouveaux matériaux (composites…) et moyens de production (impression 3D…) rendant les structures et les équipements plus légers, et réduisant donc la consommation en carburant tout en améliorant leurs performances et leurs durabilités.

Tournez la page S.V.P.

I.1.a Choix structurels pour un nouveau modèle d'avion : le cas des trains d'atterrissage

Ces changements entraînent des redimensionnements des structures et des systèmes critiques, par exemple concernant le système de trains d'atterrissage. Ce système doit non seulement répondre aux critères de performance générale, mais aussi s'adapter aux spécificités de l'avion comme sa capacité de charge, sa vitesse d'atterrissage et le type de terrains sur lesquels il opérera (figure 1). Ainsi, le choix du nombre de trains d'atterrissage et du nombre de roues par train est fortement dépendant des performances attendues de l'avion.

FIGURE 2 – Gamme de trains d'atterrissage Safran Landing System
Photos copyright Safran

La conception de nouveaux avions ne repose pas entièrement sur des composants originaux. En effet, pour des raisons de coût, de fiabilité et de délais de développement, un nouvel avion utilise principalement des composants déjà existants, proposés par les fournisseurs du constructeur. Ceux-ci présentent l'avantage d'être déjà développés et éprouvés. Cette approche permet de réduire les risques et de réaliser des économies sur des technologies déjà validées, en adaptant si nécessaire certaines caractéristiques aux exigences spécifiques du nouvel avion. Par exemple, Safran Landing Systems propose une gamme de trains d'atterrissage présentant des performances variées (figure 2).
Toutefois, le dimensionnement d'un système de train d'atterrissage ne se limite pas au choix du nombre de trains d'atterrissage et de roues. Plusieurs autres sous-systèmes et composants doivent être dimensionnés :

- des équipements hydrauliques (pompes, servovalves), permettant d'alimenter et de distribuer la haute pression aux trains d'atterrissage, pour actionner leurs sorties du fuselage avant l'atterrissage ainsi que leurs rentrées après le décollage (figure 3) ;
- des capteurs et des détecteurs divers (position, pression, etc.), un circuit hydraulique, etc. ;

- un ou plusieurs calculateurs de commande, envoyant les consignes aux préactionneurs en fonction de l'état du système (obtenu via les capteurs) et des commandes reçues du cockpit.

I.1.b Étude de sûreté de fonctionnement et point de vue dysfonctionnel

Une étude dysfonctionnelle d'un système est une approche analytique centrée sur les pannes potentielles et les limites des systèmes dans des conditions de fonctionnement non idéales. Elle permet d'anticiper les scénarios de défaillance et de concevoir des systèmes qui peuvent y faire face efficacement, ainsi que de prévoir la probabilité que des situations redoutées surviennent.

FIGURE 3 – Équipement hydraulique et servovalve hydraulique
Photos copyright Safran Aerosystems

Tout au long de la conception d'un système critique tel que celui des trains d'atterrissage, le bureau d'étude, en charge de la conception fonctionnelle du système, fait valider ses choix par une autre équipe d'ingénierie, qui est elle en charge de l'étude dysfonctionnelle du système. Les résultats dysfonctionnels permettent de sélectionner les solutions techniques qui présentent la plus faible probabilité de situation redoutée, et d'écarter celles qui ne vont pas permettre un fonctionnement suffisamment sûr et fiable. Ainsi, à chaque étape de la conception du système, seules les solutions techniques proposées par le bureau d'étude fonctionnel, qui sont suffisamment performantes et validées par les études dysfonctionnelles, sont gardées pour être étudiées lors des étapes suivantes de conception.

I.2 Système étudié

Durant la phase de pré-conception d'un nouvel avion, une architecture de système de train d'atterrissage est proposée par le bureau d'étude fonctionnel. On souhaite en déterminer la fiabilité afin de décider de la retenir, ou de l'écarter pour une autre architecture.
L'architecture proposée ici est largement inspirée du système décrit dans [2].

I.2.a Architecture

L'architecture du système de train d'atterrissage le décompose en trois parties :

- une partie mécanique, composée de trois ensembles de trains d'atterrissage identiques (avant, gauche et droite) ;
- un circuit hydraulique ;
- une commande numérique.

Tournez la page S.V.P.

I.2.b Trains d'atterrissage et séquences de fonctionnement

FIGURE 4 – Train d'atterrissage avant Safran d'un Airbus A319, avec sa porte ouverte
Photo copyright Lionel Flusin / CAPA Pictures / Safran

Chacun des trois ensembles de train d'atterrissage est identique. Chaque ensemble comporte un essieu amovible, ainsi qu'une porte permettant de refermer l'enveloppe de l'avion lorsque la roue est rentrée pour limiter les pertes aérodynamiques (figure 4), ainsi que les vérins permettant de réaliser les divers mouvements et un crochet actionnable permettant de verrouiller la porte lorsqu'elle est fermée pour éviter une ouverture accidentelle en vol (figure 5).
Les vérins utilisés sont double effet : ils peuvent être commandés en ouverture et en fermeture de la porte ou en sortie et en entrée des roues, suivant l'entrée sur laquelle la haute pression hydraulique est transmise (figure 6).

FIGURE 5 – Verrouillage de train et actionneur de trappe du train d'atterrissage
Photos copyright Safran

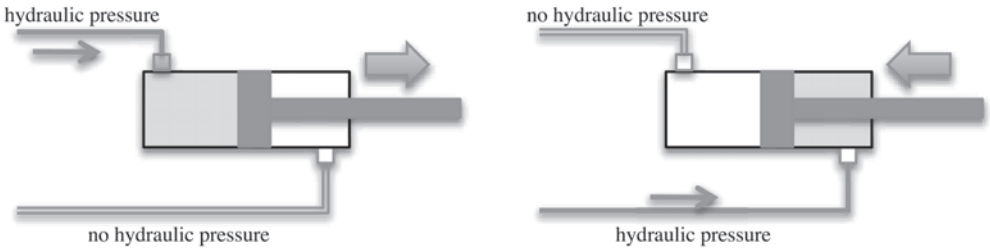


FIGURE 6 – Vérins hydrauliques

Des détecteurs de fin de course sont présents sur chaque vérin, aux deux extrémités. Ceux-ci sont doublés pour détecter une éventuelle défaillance de l'un d'eux.
Pour pouvoir sortir les roues, une séquence d'actions doit être réalisée :

1. déverrouiller la porte ;
2. ouvrir la porte ;
3. une fois la porte ouverte, tout en la gardant ouverte, sortir les roues jusqu'à la position maximale ;
4. une fois les roues sorties, fermer la porte ;
5. verrouiller la porte.

La séquence d'actions pour rentrer les roues est similaire :

1. déverrouiller la porte ;
2. ouvrir la porte ;
3. une fois la porte ouverte, tout en la gardant ouverte, rentrer les roues jusqu'à la position maximale ;
4. une fois les roues rentrées, fermer la porte ;
5. verrouiller la porte.

Chacune de ces actions est temporisée (par exemple pour permettre la stabilisation de la pression dans le circuit hydraulique, ou aux vérins de réaliser leur mouvement) (figure 7).

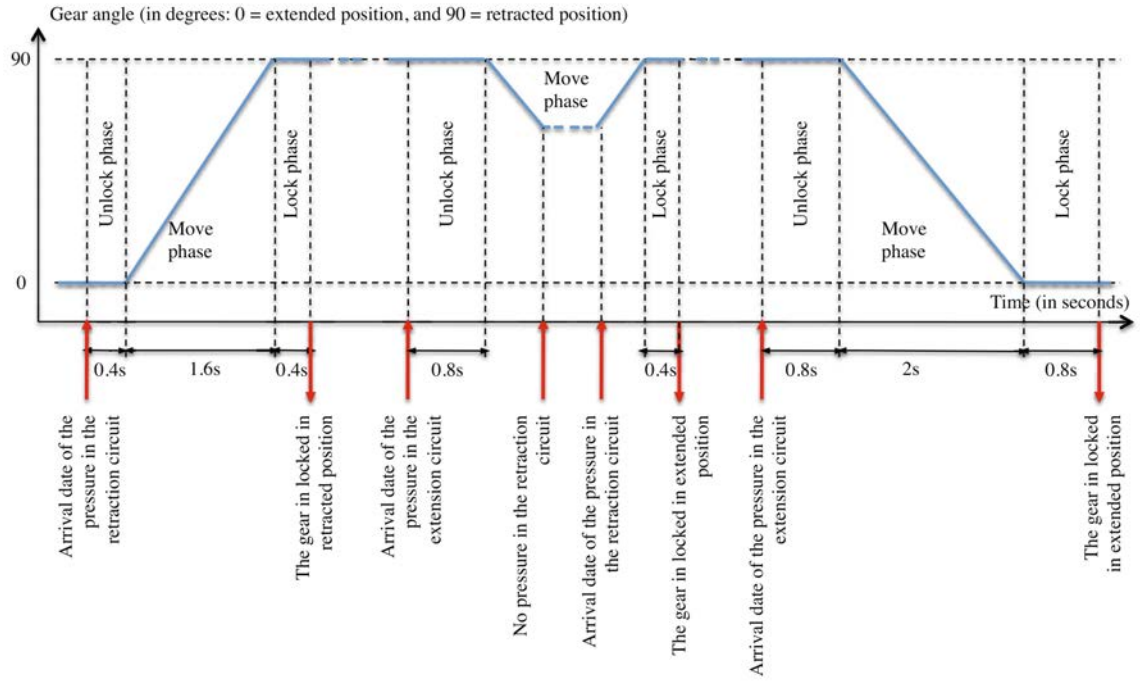


FIGURE 7 – Séquence de sortie et d'entrée des roues temporisée

Tournez la page S.V.P.

I.2.c Circuit hydraulique

Le circuit hydraulique est composé d'une pompe fournissant le fluide sous haute pression. Une électro-valve générale permet l'alimentation de l'ensemble du système en fluide haute pression, et 4 électro-valves permettent d'alimenter les 6 vérins (un vérin par porte et un vérin par ensemble de roues) en ouverture/sortie et en fermeture/rentrée.

I.2.d Calculateur de commande

Le calculateur de commande est composé de deux modules de calculs en redondance : leurs rôles sont identiques. Ils doivent :

- contrôler la séquence d'entrée ou de sortie des trains d'atterrissage lorsque l'ordre est donné ;
- détecter des éventuelles incohérences de l'état du système, pour en inférer son état de bon fonctionnement ou une éventuelle panne, à partir des informations acquises auprès de tous les capteurs et détecteurs présents dans le système ;
- informer le pilote de l'état du système, de son bon fonctionnement, mais aussi (et surtout) de la position des roues du train d'atterrissage, pour permettre à l'avion d'atterrir en sécurité.

Le calculateur fonctionne suivant deux modes :

- le mode normal, lorsque tout fonctionne correctement ;
- le mode dégradé, quand une panne a été détectée.

La détection des pannes faisant partie des rôles du calculateur, il est possible que celui-ci fonctionne en mode normal alors qu'il existe une panne (non détectée) dans le système.

I.2.e Interface Homme-Machine

L'IHM intéressante à l'étude se limite à :

- une manette à deux positions (haut/bas), que le pilote actionne pour indiquer que les roues doivent être rentrées ou sorties ;
- trois voyants, indiquant que :
  - les roues sont en mouvement ;
  - les roues sont sorties ;
  - le système est en état de bon fonctionnement.

I.2.f Exigences

L'objectif de l'étude est de déterminer des performances dysfonctionnelles de ce modèle, telles que sa fiabilité, sa disponibilité ou sa sûreté.
De plus, un ensemble d'exigences que ce système critique doit respecter a été définie. Parmi celles-ci, on peut citer l'exigence (R_{11}) : Lorsque le système est dans l'état de bon fonctionnement, si la commande de sortie du train d'atterrissage a été donnée, alors les roues sont sorties et verrouillées et les portes sont fermées moins de 15 secondes après que la commande a été donnée.

I.3 Étude proposée

La partie II propose une modélisation dysfonctionnelle de composants physiques via des chaînes de Markov, tandis que la partie III s'intéresse aux modes de défaillance du réseau de communication du système.
Dans la partie IV, un outil de modélisation plus haut niveau, sous forme d'automates à transitions gardées, permet de modéliser d'un point de vue dysfonctionnel l'ensemble du système. La partie V aborde la modélisation du comportement temporel du système.
Enfin, la partie VI étudie la simulation du modèle obtenu, en cherchant à le faire de façon efficace pour permettre une simulation stochastique du modèle (partie VII) afin d'en obtenir des performances dysfonctionnelles.

Bien que les différentes parties portent sur l'étude et la modélisation de différentes parties d'un même système, elles peuvent être abordées indépendamment les unes des autres.

Des annexes sont proposées en fin de sujet, et pourront être utilisées tout au long de l'étude.

Tournez la page S.V.P.

II Modélisation dysfonctionnelle

Une étude dysfonctionnelle consiste à modéliser ce qui peut mener à une défaillance d'un système, c'est-à-dire à ce qu'il ne remplisse pas sa fonction (tandis qu'une étude fonctionnelle recherche ce qui va permettre au système de remplir sa fonction).
Dans ce cadre, on va en premier s'intéresser aux défaillances de ses composants. Une défaillance est un évènement qui n'est, dans la plupart des cas, pas possible de prédire avec précision. Toutefois, il est possible, lorsque l'on étudie un grand nombre d'exemplaires d'un même modèle de composant, d'en déduire une loi probabiliste permettant de prédire un taux horaire de défaillances.
Mais la difficulté consiste ensuite à déterminer, à partir des défaillances de ses composants, quel est le taux horaire de défaillances du système entier : il suffit parfois qu'un seul de ses composants soit défaillant, mais il peut aussi en comporter plusieurs en redondance, ce qui complexifie le calcul.
Une méthode de calcul historiquement utilisée pour déterminer des performances dysfonctionnelles d'un système sont les chaînes de Markov.

II.1 Détecteurs

Un détecteur est un capteur qui délivre une information booléenne. Le système étudié en comporte plusieurs, essentiellement des détecteurs de position, ainsi que des capteurs de pression.
Une modélisation dysfonctionnelle simple d'un détecteur consiste en deux états : soit il fonctionne correctement (état OK), soit il est défaillant (état KO). Il n'est pas utile de prendre en compte l'état transmis par le détecteur (on ne s'intéresse pas aux « faux positifs » ou « faux négatifs » pour le moment), puisque l'on souhaite simplement connaitre la probabilité que l'on ne puisse pas avoir connaissance de l'information à mesurer.
Suite à une étude statistique, on choisit de modéliser la probabilité qu'un détecteur passe de l'état de bon fonctionnement (OK) à l'état défaillant (KO) par une probabilité constante, notée λ (de l'ordre de 10^{-4} à 10^{-10} en défaillances par heure).
Dans ces conditions, le graphe des états accessibles, représenté par un graphe orienté annoté par les taux (les probabilités) de transition constants entre états, est appelé une chaîne de Markov. La chaîne de Markov d'un détecteur est représentée figure 8.

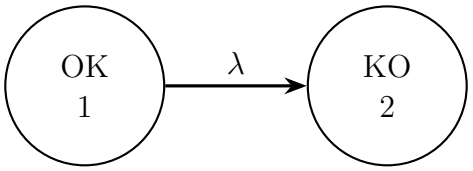


Figure 8 – Chaîne de Markov d'un système composé d'un détecteur

Une chaîne de Markov est un processus stochastique possédant la propriété de Markov.

- Processus stochastique : représente l'évolution d'une variable aléatoire, c'est-à-dire chaque résultat possible suivant des lois de probabilités.
- Propriété de Markov : l'information utile pour prédire l'état futur est entièrement contenue dans l'état présent (le système n'a pas de mémoire, et ne dépend pas de paramètres extérieurs).

Dans notre cas, l'information intéressante est l'évolution de la fiabilité[1] en fonction du temps.
On numérote les états : dans le cas du système composé d'un détecteur, l'état 1 est celui de bon fonctionnement (OK), et l'état 2 est celui de panne (KO). On note $x_i(t)$ la probabilité que le système soit dans l'état i à la date t.
Le système étant à chaque instant dans un seul et unique état, on a forcément :

$$\forall t \in \mathbb{R}, x_1(t) + x_2(t) = 1$$

Enfin, en se plaçant dans les conditions de Heaviside et en considérant que le système est neuf à l'instant $t = 0$, on a $x_1(0) = 1$ et $x_2(0) = 0$.
Ainsi, la probabilité que le système soit dans un état à une date donnée dépend :

- de la probabilité que le système soit dans cet état à un instant précédent, et n'en sorte pas ;
- de la probabilité que le système soit dans un autre état, et qu'il arrive dans cet état.

Cela permet d'obtenir, en utilisant la relation d'Euler explicite, le système d'équations suivant :

$$\begin{cases} x_1(t+dt) &= x_1(t) - \underbrace{\lambda dt x_1(t)}_{\text{vers l'état 2}} \\ x_2(t+dt) &= x_2(t) + \underbrace{\lambda dt x_1(t)}_{\text{depuis l'état 1}} \end{cases}$$

Afin d'améliorer la fiabilité du système, les détecteurs sont dédoublés : on étudie donc un ensemble de deux détecteurs, mesurant la même information. Ainsi, l'état redouté est celui dans lequel tous les détecteurs sont défaillants. Dans le cas où un détecteur est dédoublé, on obtient la chaîne de Markov représentée figure 9. La première partie du nom d'un état correspond au premier détecteur (OK ou KO), et la fin au deuxième détecteur. Pour la suite de l'étude, les états ont été numérotés (l'état OK-OK étant l'état 1).

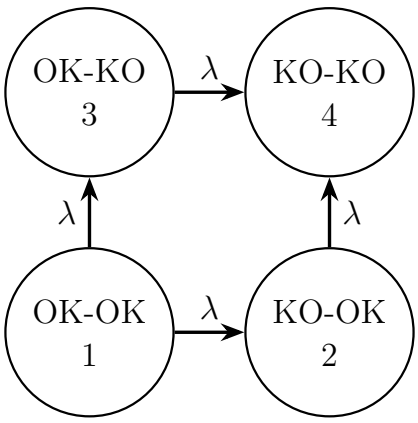


FIGURE 9 – Chaîne de Markov d'un système composé de deux détecteurs

1. Fiabilité : probabilité de bon fonctionnement à une date donnée.

Tournez la page S.V.P.

On note $x_1(t)$ la probabilité que le système à deux détecteurs soit dans l'état de bon fonctionnement (état OK-OK, numéroté 1) à la date t. On note de même $x_2(t)$, $x_3(t)$, $x_4(t)$. On note $X(t) = (x_1(t), x_2(t), x_3(t), x_4(t))$.

Question 1 : En utilisant la relation d'Euler explicite, montrer que l'on obtient la relation suivante :

$$X(t + dt) = X(t) \times P$$

avec P une matrice dont on précisera les coefficients.

Afin de calculer l'évolution probabilités d'état, on discrétise le temps par pas de 1 h. On note t_n la date du n-ième pas de temps, avec $t_0 = 0$ h. De même, on note $X_n = X(t_n)$ On obtient alors la relation discrétisée :

$$X_{n+1} = X_n \times P$$

Question 2 : Exprimer X_n en fonction de P, n et X_0.

Les propriétés de P la rendent diagonalisable. On note :

$$P = R \times D \times R^{-1}$$

avec :

$$R = \begin{pmatrix} 1 & 1 & 0 & -1 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & -1 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} \qquad R^{-1} = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & -2 \\ 0 & 1 & 0 & -1 \\ -1 & 1 & 1 & -1 \end{pmatrix} \qquad D = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1-\lambda & 0 & 0 \\ 0 & 0 & 1-\lambda & 0 \\ 0 & 0 & 0 & 1-2\lambda \end{pmatrix}$$

Question 3 : Déterminer $x_4(t_n)$ $\forall n \in \mathbb{N}$.

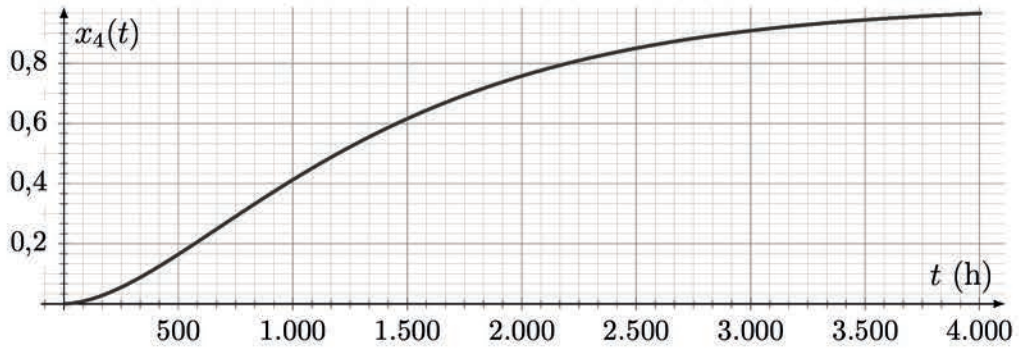


FIGURE 10 – Evolution de $x_4(t_n)$

L'évolution de x_4 en fonction de t, pour une valeur de λ donnée, a été tracée figure 10. Le MTTF [2] est un indicateur statistique de sûreté de fonctionnement, représentant la durée de vie moyenne de l'ensemble des systèmes considérés. Il est déterminé par la moyenne des durées de bon fonctionnement, c'est-à-dire de l'instant 0 à la défaillance.

Question 4 : Sans réaliser de calculs, expliquer comment calculer la valeur du MTTF à partir de $x_4(t)$.

2. MTTF : Mean Time To Failure

II.2 Circuit d'alimentation de fluide

On s'intéresse maintenant à un circuit d'alimentation de fluide sous pression d'un sous-système, via un ensemble d'électrovannes qui relient une alimentation de fluide sous pression à un ensemble d'actionneurs. On souhaite pouvoir contrôler l'alimentation en fluide, c'est-à-dire décider de fournir du fluide, ou de ne pas en fournir.
Une électrovanne peut être modélisée, du point de vue dysfonctionnel, par un automate à état. Deux variables permettent de représenter l'état d'une électrovanne.

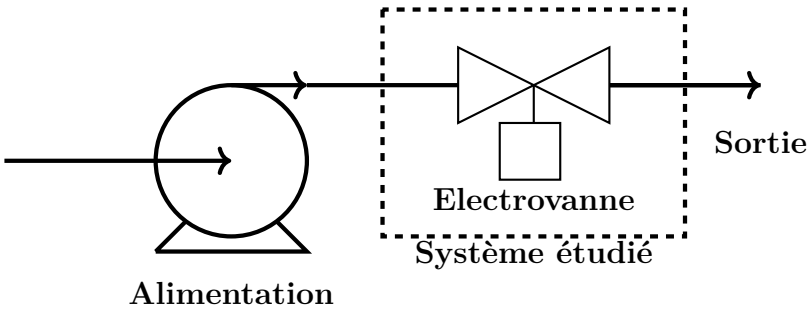
- Sa position : ouverte (laissant passer le fluide, noté O), ou fermée (noté F) ;
- Son état de fonctionnement : bon fonctionnement (noté OK), ou en panne (noté KO).

Ainsi, l'état d'une électrovanne sera noté de manière abrégée : par exemple, l'état de bon fonctionnement en position fermée sera noté FOK.
En bon fonctionnement, l'électrovanne suit une consigne d'ouverture ou de fermeture. Lorsque l'électrovanne est en panne, sa position ne peut plus évoluer.

Question 5 : Justifier qu'il n'est pas possible de modéliser ce comportement par une chaîne de Markov.

II.2.a Modèle à une électrovanne

On s'intéresse au système composé d'une unique électrovanne.



Question 6 : Représenter graphiquement, sous forme d'un graphe orienté (sans préciser les probabilités de transition), le graphe des états accessibles du modèle d'une électrovanne.

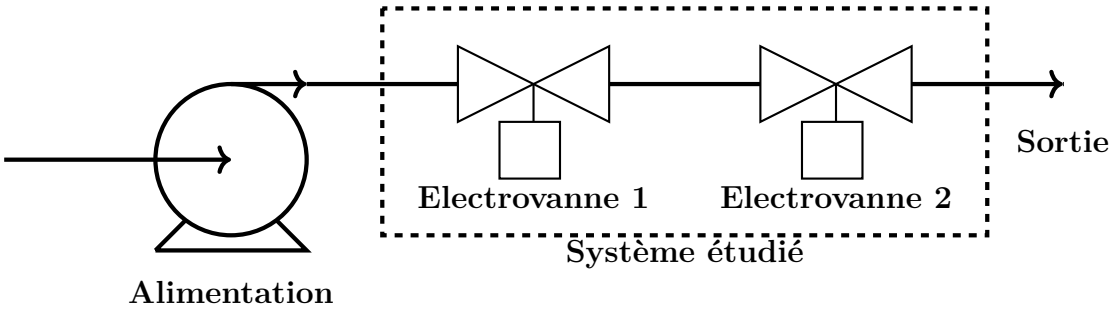
Tournez la page S.V.P.

II.2.b Modèle à deux électrovannes en série en redondance chaude

Afin de diminuer le risque de dysfonctionnement d'un système (c'est-à-dire le risque que le système ne puisse plus remplir sa fonction), une solution technologique est d'utiliser plusieurs composants en redondance.
Il existe plusieurs méthodes de redondances, dont :

- la redondance dîte «chaude», pour laquelle deux composants (ou plus) réalisent les mêmes actions en parallèle. Si l'un des composants dysfonctionne, les autres continuent. Ainsi, sous certaines conditions, le système continue à assurer sa fonction.
- la redondance dîte «froide», pour laquelle l'un des composants (dit activé) réalise la fonction, tandis qu'un autre composant (ou plusieurs autres composants) sont éteints (et ne se dégradent donc pas, ou plus faiblement et lentement que s'ils étaient en fonctionnement). Lorsque le composant activé dysfonctionne, le composant en redondance (ou l'un d'entre eux) est activé, et prend le relai du composant dysfonctionnel pour réaliser la fonction du système.

On s'intéresse dans un premier temps à l'utilisation de deux électrovannes en série en redondance chaude. Les deux électrovannes vont, dans la mesure du possible, suivre le même cycle d'ouverture et de fermeture. Toutefois, comme elles ne sont pas parfaitement synchronisées, le système ne pourra pas passer directement de l'état « les deux vannes sont ouvertes » à l'état « les deux vannes sont fermées » (et inversement), mais passera par un état intermédiaire.



Question 7 : Représenter graphiquement, sous forme d'un graphe orienté, le graphe des états accessibles du modèle du circuit composé de deux électrovannes en série. Indiquer en vert les sommets qui représentent une défaillance, c'est-à-dire ceux pour lesquels on ne peut pas contrôler l'alimentation en fluide.

II.2.c Modèle à deux électrovannes en série en redondance froide

On souhaite maintenant appliquer une solution de redondance froide au système composé de deux électrovannes en série.

Question 8 : Représenter graphiquement, sous forme d'un graphe orienté, le graphe des états accessibles du modèle du circuit composé de deux électrovannes en série en redondance froide.

II.3 Modélisation d'un avion

Un avion commercial moderne est un système comportant un nombre de composants participant à des fonctions critiques de l'ordre du millier.

Question 9 : En considérant une modélisation simple de chaque composant, déterminer l'ordre de grandeur du nombre de sommets du graphe des états accessibles d'un modèle regroupant 1000 composants. Conclure sur l'utilisation de cette méthode pour l'étude d'un avion commercial moderne.

L'utilisation des chaînes de Markov pour obtenir des indicateurs de sûreté de fonctionnement se heurte à des limites contraignantes sur les modèles étudiables. L'utilisation de modèles plus expressifs est nécessaire pour permettre d'étudier des systèmes complexes.

L'approche MBSA [3] consiste à développer des outils d'étude à partir des modèles, à l'inverse de l'approche historique de développer des modèles à partir des outils d'étude disponibles. Ainsi, la partie IV propose l'utilisation d'un outil de modélisation plus expressif (notamment pour la modélisation des taux de transitions, étudiés en partie V). La simulation stochastique est un outil d'étude permettant des calculs de sûreté de fonctionnement à partir de modèles complexes, ce qui est l'objet des parties VI et VII.

3. MBSA : Model-Based Safety Assessment

Tournez la page S.V.P.

III Transmission des informations

Un module de tableau de bord typique, tel qu'un tableau de commande de système d'alimentation électrique, peut comporter 10 à 15 interrupteurs et voyants locaux liés au système. Chaque interrupteur ayant au moins six fils, soit un total d'au moins 90 fils allant du poste de pilotage au compartiment avionique à partir d'un seul tableau de commande.
Airbus a repensé ces panneaux de commande en connectant tous les interrupteurs et voyants d'un panneau à un contrôleur de bus CAN. Celui-ci fait partie intégrante du panneau, et les données sont transmises à l'aide de deux fils seulement. C'est ce qu'on appelle les panneaux de contrôle intégrés (ICP : Integrated Control Panel). Les ICP communiquent avec les modules d'entrée/sortie (CPIOM : Core Processing Inputs/outputs Modules ; et CRDC : Common Remote Data Concentrator).
Pour réduire le nombre de fils d'interconnexion entre les différents éléments du système étudié, notamment entre les calculateurs (CPIOM et CRDC) et les commandes du cockpit (ICP), le constructeur a mis en place un réseau de communication des données avioniques (ADCN) au moyen d'un bus CAN (dont une description est proposée en annexe).
Une Analyse des Modes de Défaillance et de leurs Effets (AMDE) conduite sur l'ADCN a permis de mettre en évidence trois modes de défaillance prépondérants :

- saturation du bus ;
- erreur de transmission des trames ;
- collision de trames.

L'objectif de la suite de l'étude est d'étudier des solutions technologiques pour détecter ces modes de défaillance et/ou mitiger leurs effets.

III.1 Erreur de transmission des trames

Les trames CAN sont transmises via un câble comportant deux fils (CANH et CANL) conducteurs torsadés (enroulés en hélice l'un autour de l'autre).

Question 10 : Expliquer comment le signal électrique est lu, pourquoi deux fils sont utilisés, et l'intérêt des torsades.

Question 11 : Indiquer quel autre mécanisme est mis en place pour limiter les conséquences d'une erreur de transmission d'une trame CAN.

III.2 Saturation du bus

Un ICP comprend 15 nœuds qui transmettent toutes les 40 ms des données comportant en moyenne 2 octets d'informations utiles. Le bus est configuré à un débit de 1 Mbits/s.

Question 12 : Déterminer la charge réelle du réseau CAN, et la comparer à sa capacité de charge maximale. Conclure sur le risque de saturation du bus.

III.3 Collision de trames

Le temps de propagation d'un signal électrique n'étant pas nul, un nœud peut commencer à émettre une trame sans avoir la possibilité de savoir qu'un autre nœud a fait de même. Pour gérer ce problème, le bus implémente l'algorithme CSMA/AMP (Carrier Sense Multiple Access with Arbitration on Message Priority), qui fonctionne comme suit :

1. un nœud écoute le canal pour vérifier s'il est libre (Carrier Sense) ;
2. si le canal est libre, le nœud commence à transmettre des données ;
3. si le canal est occupé, le nœud attend et réessaye après un certain délai ;
4. pendant la transmission, le nœud continue de surveiller le canal, en comparant ce qu'il a envoyé, et ce qui est effectivement présent sur le canal. Un bit à l'état logique « 0 » étant dominant, et un bit à l'état logique « 1 » étant récessif, lorsque deux nœuds transmettent simultanément :
   - s'ils envoient les mêmes bits, alors aucune collision n'est détectée ;
   - s'ils envoient deux bits différents, alors celui qui a transmis un bit « 1 » lira sur le canal un bit « 0 », détectera qu'il y a eu une collision, et s'arrêtera de transmettre (avant de recommencer plus tard). L'autre nœud ne détectera aucune collision, tout comme les autres nœuds non-émetteurs.

On appelle **temps de propagation (τ)** le temps nécessaire pour transmettre un signal entre deux nœuds (on prendra en général en compte le temps de propagation entre les deux stations les plus éloignées).
Dans le protocole CSMA/AMP, le nœud émetteur d'une trame reste à l'écoute du support pour détecter d'éventuelles collisions avec un autre nœud lors de l'envoi de chaque bit d'information. On appelle **fenêtre de collision (slot-time)** le délai maximum qui s'écoule avant qu'une collision ne soit détectée par un nœud émetteur, mesuré à partir du début de l'envoi d'un bit.

Question 13 : Donner la relation entre la taille (durée) de la fenêtre de collision notée t_{fc} et le temps de propagation τ.

Question 14 : Donner la relation entre la fenêtre de collision et la fréquence d'émission f pour que le nœud émetteur soit certain que sa trame a été transmise sans collision.

Le bus fonctionne à $f = 1\,\text{Mbits/s}$, et la vitesse de propagation d'un signal électrique sur les câbles utilisés est d'environ $\frac{1}{v} = 5\,\text{ns/m}$.

Question 15 : Déterminer la distance maximale entre deux nœuds sur le bus CAN afin d'éviter une collision qui ne serait pas détectée par l'émetteur. La distance entre deux nœuds étant au maximum de $40\,\text{m}$, conclure sur la viabilité du bus CAN.

III.4 Conclusion

Les modes de défaillance étudiés ont des effets et des probabilités différentes.

Question 16 : Conclure sur l'importance de la prise en compte de chaque mode de défaillance du réseau de communication dans le cadre d'une étude dysfonctionnelle du système de trains d'atterrissage.

Tournez la page S.V.P.

IV Modélisation par un Automate à Transitions Gardées (GTS)

Afin de pouvoir modéliser des comportements plus complexes, notamment des interactions entre composants (comme des redondances froides), une solution est d'utiliser des automates à états temporisés.
Dans [3], des automates à états et transitions gardées, synchronisées et stochastiques, nommés « Guarded Transition Systems » (GTS) sont définis. Ce formalisme a été retenu car il permet de modéliser des systèmes d'un point de vue dysfonctionnel, et surtout de simuler efficacement ces modèles pour calculer des indicateurs statistiques de sûreté de fonctionnement.
Ces GTS présentent plusieurs propriétés utiles, dont :

- la description dysfonctionnelle composant par composant, assemblés d'une façon proche de l'architecture fonctionnelle du système étudié ;
- la possibilité de décrire les composants via un langage de modélisation haut-niveau orienté prototype ou orienté objet, ce qui permet de rapidement modéliser de nouveaux composants en dupliquant, héritant, composant, redéfinissant des modèles déjà existants. On peut ainsi construire des bibliothèques de modèles de composants, et étudier différentes configurations possibles d'un système (par exemple en testant la fiabilité d'une solution technique avec deux composants en redondance, ou avec trois) sans repartir d'un modèle vierge ;
- la possibilité de modéliser des transitions temporisées suivant des délais stochastiques, permettant de représenter des pannes « aléatoires » par des lois probabilistes de n'importe quel type.

IV.1 Guarded Transition Systems

Les modèles GTS sont des automates à états et transitions gardées. Un tel modèle est un ensemble composé de :

- variables d'état typées, dont les valeurs représentent l'état actuel du modèle ;
- transitions temporisées et gardées. Une transition est un ensemble composé de :
  - une garde : expression booléenne portant sur les variables du GTS ;
  - une action : un ensemble d'affectations, éventuellement conditionnelles, portant sur les variables du GTS ;
  - un évènement temporisé : une fonction qui ne prend pas d'argument et renvoie un temps positif, pouvant être une constante (éventuellement nulle), ou un nombre positif aléatoire suivant une loi probabiliste propre à chaque temporisation.
- variables de flux typées et d'assertions permettant de calculer leurs valeurs, dont l'objet est de modéliser les interactions entre les composants d'un système ;

Les types des variables d'état et de flux peuvent théoriquement être de n'importe quel type, du moment que des calculs peuvent être réalisés avec : `float`, `int`, `str`, etc. Toutefois, dans le cadre de ce sujet, on se limitera aux variables de type `bool` (booléennes).

IV.1.a Description d'un composant

Par exemple, le GTS présenté figure 11 modélise un interrupteur pouvant défaillir. Son état est contenu dans deux variables d'état booléennes :

- `working`, qui vaut initialement `True`, et qui indique si l'interrupteur est en bon état de fonctionnement ;
- `position`, qui vaut initialement `False`, et qui indique si l'interrupteur est en position ouverte (`False`) ou fermée (`True`).

Interrupteur			
nom de la variable d'état	: *type*	*valeur initiale*	
`working`	: Boolean	`True`	
`position`	: Boolean	`False`	
nom de la transition	*temporisation*	: *garde*	-> *action*
`eOn`	constante(5)	: `working` and not `position`	-> `position` = `True`
`eOff`	constante(10)	: `working` and `position`	-> `position` = `False`
`eFail`	exponentielle(10^{-7})	: `working`	-> `working` = `False`

FIGURE 11 – Modèle GTS d'un interrupteur

Ses comportements fonctionnel et dysfonctionnel sont modélisés par trois transitions temporisées et gardées :

- la transition `eOn` modélise la fermeture de l'interrupteur (passage de `position` de la valeur `False` à `True`) :
  - cette transition n'est tirable que si l'interrupteur est en position ouverte et est en bon état de fonctionnement, ce qui est imposé par la garde `working and not position` ;
  - lorsque cette transition est tirée, la position passe à la valeur fermée, par l'exécution de l'action (`position := True`) ;
  - le tir de cette transition a lieu après un délai constant de 5 s, mesuré à partir de l'instant auquel la transition devient tirable, c'est-à-dire que sa garde devient vraie. Si la garde devient fausse, la mesure est remise à zéro.
- la transition `eOff`, qui modélise l'ouverture de l'interrupteur et qui est, dans sa description, similaire à `eOn`, mais avec un délai constant de 10 s :
  - ainsi, tant qu'il est dans un état de bon fonctionnement, l'interrupteur va alterner entre la position fermée durant 5 s et l'état ouvert durant 10 s.
- la transition `eFail`, qui modélise la défaillance de l'interrupteur :
  - cette transition n'est tirable que si l'interrupteur est en état de bon fonctionnement, comme l'indique la garde `working` (`working` doit avoir la valeur `True`) ;
  - lorsque cette transition est tirée, `working` prend la valeur `False` (comme l'indique l'action `working := False`) ;
  - le tir de cette transition a lieu après un délai aléatoire, lui aussi mesuré à partir de l'instant auquel la transition devient tirable, choisi suivant une loi probabiliste exponentielle de paramètre $1 \times 10^{-7}\,\mathrm{s}^{-1}$.

Ce GTS ne comporte pas de variable de flux pour le moment, mais les gardes et les expressions des actions (membres de droite des actions) peuvent porter aussi bien sur des

Tournez la page S.V.P.

variables d'état que sur des variables de flux, ou des constantes. En revanche, les variables affectées par les actions (membres de gauche des actions) sont forcément des variables d'état.

IV.1.b Description d'un système à partir de composants

À partir de ce GTS d'un interrupteur, il est possible de créer le modèle d'un panneau de contrôle, constitué, par composition, de deux interrupteurs, nommés `I1` et `I2` (figure 12).

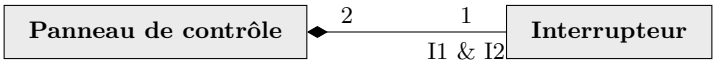


FIGURE 12 – Modèle GTS d'un panneau de contrôle constitué de deux interrupteurs

Les propriétés des GTS font qu'il est très facile de «mettre à plat» les relations de composition, pour obtenir un unique GTS. Ainsi, le GTS figure 13 est équivalent à celui de la figure 12, mais les deux interrupteurs ont été mis à plat dans le GTS du panneau de contrôle.

Panneau de contrôle			
`I1.working`	: Boolean	`True`	
`I1.position`	: Boolean	`False`	
`I2.working`	: Boolean	`True`	
`I2.position`	: Boolean	`False`	
`I1.eOn`	constante(5)	: `I1.working` and not `I1.position`	-> `I1.position` = `True`
`I1.eOff`	constante(10)	: `I1.working` and `I1.position`	-> `I1.position` = `False`
`I1.eFail`	exponentielle(10^{-7})	: `I1.working`	-> `I1.working` = `False`
`I2.eOn`	constante(5)	: `I2.working` and not `I2.position`	-> `I2.position` = `True`
`I2.eOff`	constante(10)	: `I2.working` and `I2.position`	-> `I2.position` = `False`
`I2.eFail`	exponentielle(10^{-7})	: `I2.working`	-> `I2.working` = `False`

FIGURE 13 – Modèle GTS mis à plat d'un panneau de contrôle

IV.1.c Interactions entre composants

Les variables de flux et les assertions permettent de facilement modéliser les interactions entre composants. On peut compléter le modèle de l'interrupteur pour lui ajouter deux connecteurs, un pour l'entrée et l'autre pour la sortie (figure 14). Chacun est modélisé par une variable de flux booléenne : `True` indiquant que le potentiel électrique est au niveau haut, tandis que `False` qu'il est nul. Si le connecteur n'est pas relié à une source de tension, alors son potentiel sera considéré au niveau bas : la valeur par défaut sera donc `True`.

Le comportement de l'interrupteur peut être modélisé par l'affectation conditionnelle suivante : si l'interrupteur est à la position fermée, alors la sortie est au même potentiel que l'entrée, peu importe qu'il soit en état de bon fonctionnement (cela n'a de conséquence

que sur sa capacité à modifier sa position). Cette affectation conditionnelle se traduit par l'assertion :

- de condition `position` ;
- d'affectation `out = in`.

Les conditions peuvent porter sur les variables d'état comme de flux, tout comme la valeur calculée de l'affectation (membre de droite de l'affectation). En revanche, la variable affectée (membre gauche de l'affectation) est forcément une variable de flux.
Comme rien ne force le sens du courant électrique, on pourrait aussi modéliser la relation inverse en ajoutant l'assertion correspondante. Pour la simplicité de l'exemple, ce comportement ne fait pas partie de ce modèle.

Interrupteur

nom de la variable d'état	: *type*	*valeur initiale*
`working`	: Boolean	`True`
`position`	: Boolean	`False`

nom de la transition	*temporisation*	: *garde*	-> *action*
`eOn`	constante(5)	: `working` and not `position`	-> `position` = `True`
`eOff`	constante(10)	: `working` and `position`	-> `position` = `False`
`eFail`	exponentielle(10^{-7})	: `working`	-> `working` = `False`

nom de la variable de flux	: *type*	*valeur par défaut*
`in`	: Boolean	False
`out`	: Boolean	False

condition de l'assertion	: *variable affectée*	= *valeur de l'affectation*
`position`	: `out`	= `in`

FIGURE 14 – Modèle GTS d'un interrupteur avec une entrée et une sortie

Les variables de flux et les assertions se mettent à plat de la même façon que les variables d'état et les transitions. On peut alors représenter les relations entre les composants en reliant les variables de flux. Par exemple, pour indiquer que les deux interrupteurs sont en série, il suffit d'ajouter l'assertion :

- de condition `True` (toujours vraie) ;
- d'affectation `I2.in` = `I1.out` ;

On obtient alors le GTS mis à plat figure 15.

IV.2 Simulation d'un GTS

La simulation d'un GTS est réalisée pas à pas, chaque pas correspondant au tir d'une transition. Ainsi, entre deux tirs de transition, l'état d'un modèle GTS n'évolue pas, et il n'y a aucun calcul à réaliser : c'est l'un des avantages importants de ce type de modèle, puisqu'il permet de simuler à moindre coût des temps longs, au contraire par exemple d'une simulation multiphysique qui nécessite de nombreux pas de calcul.
La simulation d'un GTS peut donc être réalisée de façon programmatique. Un ensemble de classes Python à cet effet a été défini (voir en annexe VIII).
L'objet de cette partie est de définir un ensemble d'étapes nécessaires à la simulation.

Tournez la page S.V.P.

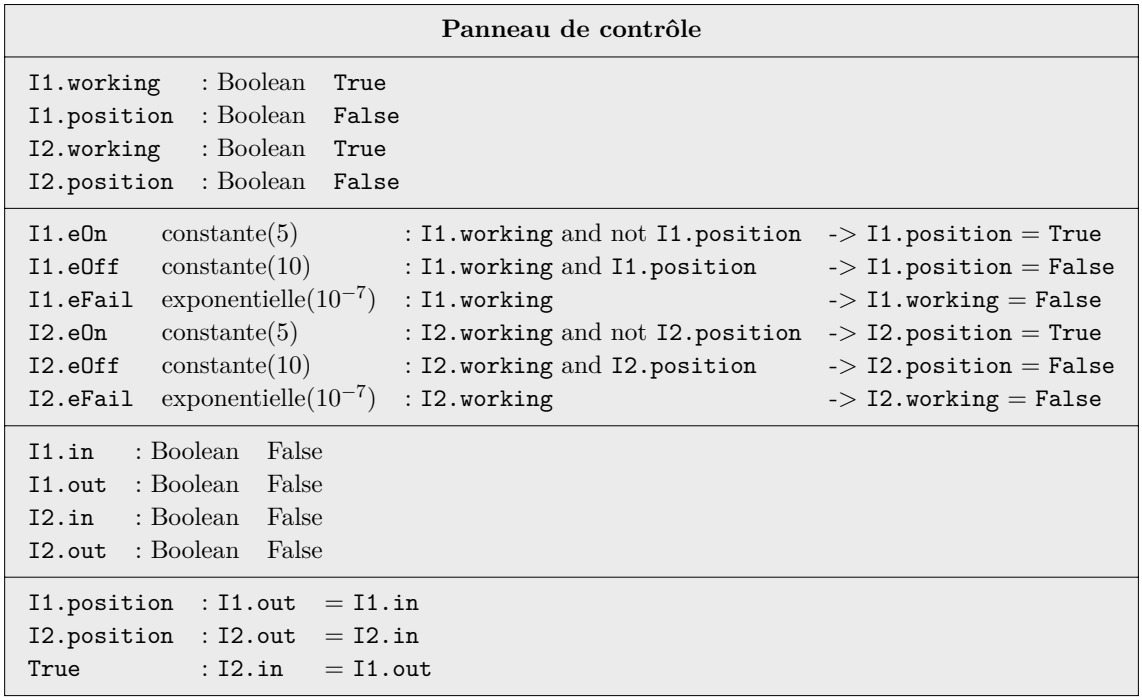


FIGURE 15 – Modèle GTS mis à plat d'un panneau de contrôle

La simulation d'un GTS suit les étapes suivantes :

- affectation des valeurs initiales aux variables ;
- tant que nécessaire :
  - mise à jour de l'échéancier des transitions tirables : (étudié en partie VI.2)
    - si une garde devient fausse, retrait de la transition de l'échéancier ;
    - si une garde devient vraie, insertion de la transition de l'échéancier avec une date de tir choisie en fonction du délai associé (éventuellement aléatoire) (étudié en partie V) ;
  - avancement du temps jusqu'à la date de tir de la prochaine transition ;
  - tir de la transition : application de son action ;
  - propagation (calcul) des variables de flux (étudié en partie VI.1).

L'intérêt de pouvoir simuler un GTS est de pouvoir en obtenir des indicateurs probabilistes de sûreté de fonctionnement. Pour cela, une simulation stochastique (aussi appelée simulation de Monte-Carlo) peut être mise en place (étudié en partie VII). Mais, pour qu'elle soit réalisable en un temps raisonnable, il est nécessaire que les différents algorithmes de simulations soient efficaces (étudié en partie VI.3).

IV.3 Modélisation d'un détecteur

Le système étudié comporte plusieurs détecteurs de fin de course, aux extrémités de chacun des vérins hydrauliques. On souhaite obtenir un modèle GTS de ce composant, avec son comportement dysfonctionnel.
Un détecteur de fin de course émet une information booléenne à partir d'une mesure d'une grandeur physique. La grandeur physique mesurée étant une position d'un vérin, elle sera modélisée par une grandeur booléenne. On aura donc deux variables de flux

booléennes, `fIn` et `fOut`, pour modéliser l'entrée (le mesurande) et la sortie (la mesure) du détecteur de fin de course.
Une analyse des modes de défaillance d'un détecteur de fin de course a mis en évidence plusieurs modes de défaillance :

- un mode d'extinction complète, de loi de probabilité exponentielle de taux 10^{-6}, dans lequel la sortie est toujours à l'état bas (`False`) ;
- un mode court-circuit, de loi de probabilité exponentielle de taux 10^{-7}, dans lequel la sortie est toujours à l'état haut (`True`) ;

Lorsqu'il fonctionne correctement, un détecteur de fin de course peut être modélisé par un « fil », dont la sortie est identique à l'entrée.

Question 17 : Proposer un modèle GTS de ce composant.

Un autre mode de défaillance existe : le mode collé, de loi de probabilité exponentielle de taux 10^{-5}, dans lequel la sortie reste à l'état qu'elle avait lors de la défaillance.

Question 18 : Indiquer les modifications à apporter au modèle GTS précédent pour prendre en compte le mode de défaillance collé.

IV.4 Modélisation des fonctions de fusion de données et de surveillance du calculateur

Parmi les différentes fonctions que doivent réaliser chacun des calculateurs, la détection de problèmes mécaniques repose sur un ensemble de règles qui, si elles ne sont pas respectées, doivent conduire à signaler un problème au pilote et à passer en mode dégradé.
L'une de ces règles porte sur les ensembles de détecteurs de fin de course. Chaque fin de course est mesurée par trois détecteurs, qui peuvent chacun défaillir suivant différents modes.
Pour chaque fin de course, chaque calculateur reçoit trois informations booléennes, via trois variables de flux différentes, nommées `fIn1`, `fIn2` et `fIn3`. La valeur de l'information considérée (fusion des trois entrées) est obtenue sur la variable de flux `fOut`. Deux fonctions doivent être réalisées par le calculateur : fusionner les trois valeurs obtenues en une seule, et détecter les défaillances des détecteurs.
Le protocole concernant les détecteurs est le suivant :

- si à la date t les trois informations sont considérées valides et sont égales, alors la valeur considérée par le calculateur est la valeur commune ;
- si à la date t une information est différente des deux autres pour la première fois (i.e. les trois informations étaient considérées comme valides jusqu'à t), alors cette information est considérée comme invalide et est définitivement éliminée. Seules les deux informations restantes sont considérées comme valides dans le futur. La valeur considérée par le calculateur est la valeur commune aux deux informations restantes ;
- si une information a été éliminée précédemment, et qu'à la date t les deux informations restantes ne sont pas de valeurs égales, alors une anomalie est signalée.

Une anomalie est signalée en passant à `True` la variable de flux `fAnomalie`.

Question 19 : Proposer un modèle GTS de la fonction de fusion d'information et de détection des défaillances des détecteurs.

V Dates des évènements

V.1 Tirage aléatoire des dates

Une transition d'un GTS est un passage d'un état à un autre, qui survient :

- soit de façon voulue, programmée : par exemple, lorsqu'un composant est commandé pour assurer une fonction (ouverture d'une vanne pour sortir le train d'atterrissage…), ou en réaction à un changement d'état d'un autre composant. La date à laquelle cette transition est franchie peut alors être calculée de façon déterministe en ajoutant un délai fixe à partir de la date à laquelle une condition devient vérifiée.
- soit de façon incertaine ou inattendue : par exemple, lorsqu'un composant tombe en panne, ou lorsque le délai d'exécution d'une tâche dépend de paramètres extérieurs ou inconnus. Dans ce cas là, la date à laquelle cette transition est franchie est stochastique, et peut-être modélisée par un délai, dépendant d'une densité de probabilité, ajouté à la date à laquelle une condition devient vérifiée.

Afin de simuler le modèle d'un système, il faut savoir calculer les dates de franchissement des transitions, notamment dans le cas stochastique.
Un indicateur classique de sûreté de fonctionnement d'un composant est la fiabilité. La fiabilité d'un composant est une fonction temporelle, notée $R(t)$, qui donne la probabilité que le composant ne tombe pas en panne avant la date t (ou, avec le point de vue opposé, que le composant tombe en panne après la date t, car tous les composants finissent par tomber en panne). $R(t)$ est donc une fonction à valeurs dans $[0, 1]$ décroissante, avec $R(0) = 1$ (à $t = 0$, le composant est neuf et est en bon état).
Cet indicateur permet de calculer la densité de défaillance (ou densité de probabilité de défaillance), qui est une fonction temporelle représentant la probabilité que le composant défaille entre les dates t et $t + dt$. En faisant tendre dt vers 0, la densité de défaillance $f(t)$ se calcule par :

$$f(t) = -\frac{dR}{dt}(t)$$

On peut aussi calculer le taux de défaillances $\lambda(t)$, qui est une fonction temporelle représentant la probabilité que le composant défaille entre les dates t et $d + dt$ à la condition qu'il n'ai pas encore défailli :

$$\lambda(t) = -\frac{1}{R(t)}\frac{dR}{dt}(t)$$

Parmi les densité de probabilité classiquement utilisées pour modéliser la survenance d'une panne d'un système, on peut citer la densité de probabilité exponentielle qui, comme son nom ne l'indique pas, est constante. Elle permet de modéliser une probabilité de défaillance « à taux constant » : à chaque instant, il y a une probabilité constante qu'un système, s'il est en bon état de marche, tombe en panne, ce qui se traduit par :

$$\lambda(t) = \text{constante} = \lambda_{\text{expo}}, \forall t \in \mathbb{R}^+$$

Plus le temps augmente, moins il est probable que le système soit en bon état de marche, et donc moins la densité de défaillance est grande. La fiabilité correspondante est alors une exponentielle décroissante (figure 16), tout comme la densité de défaillance (figure 17), d'où l'appellation d'exponentielle :

$$R_{\text{expo}}(t) = e^{-\lambda_{\text{expo}} \times t} \qquad f_{\text{expo}}(t) = \lambda_{\text{expo}} \times e^{-\lambda_{\text{expo}} \times t}$$

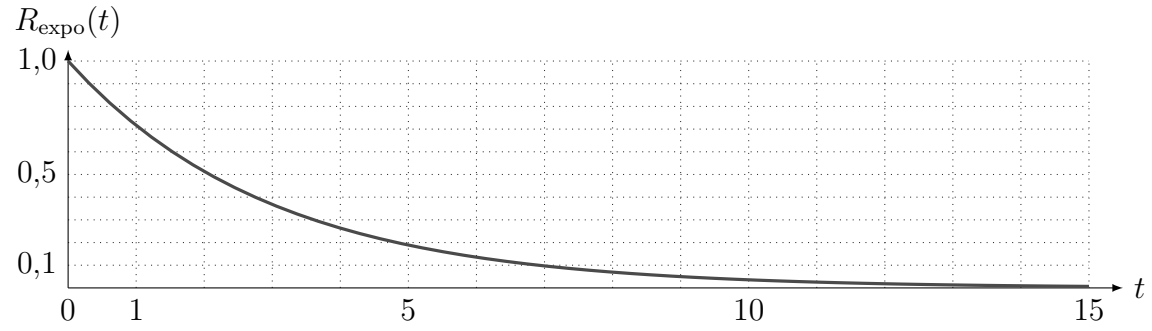


FIGURE 16 – Fiabilité d'un modèle de probabilité exponentielle

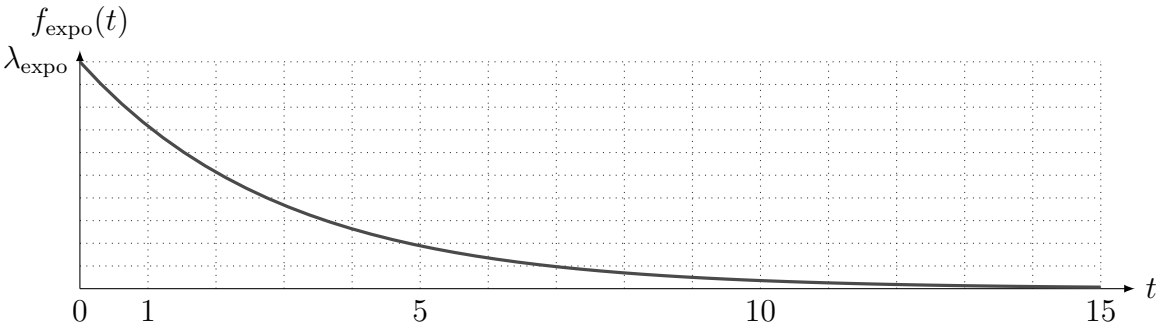


FIGURE 17 – Densité de défaillance d'un modèle de probabilité exponentielle

Question 20 : Proposer une méthode permettant de choisir aléatoirement une date de défaillance d'un composant suivant la loi probabiliste modélisant son comportement dysfonctionnel.

V.2 Durée de mouvement des vérins hydrauliques

Outre les pannes, les processus physiques peuvent aussi être modélisés stochastiquement. La séquence de rentrée-sortie des trains d'atterrissage est temporisée (figure 7), et celle-ci est surveillée par l'exigence (R_{11}). Il est donc nécessaire de modéliser le temps mis pour que chaque vérin réalise son mouvement.
La durée nécessaire pour qu'un vérin hydraulique rentre ou sorte dépend de l'état du circuit hydraulique (pression, utilisation) et de contraintes extérieures (efforts appliqués sur le vérin, frottements…).
Une étude a permis de déterminer la durée nominale des différentes étapes de rentrée-sortie des vérins des trains d'atterrissage, que ce soit pour actionner la porte ou la roue (dans les deux sens), ou pour verrouiller la porte (uniquement en position rentrée) ou la roue (dans les deux positions), pour les trois trains d'atterrissage (avant, droite et gauche). Ces différentes durées nominales sont résumées dans la table 1.

Tournez la page S.V.P.

Durée	avant		droite		gauche	
(en s)	roue	porte	roue	porte	roue	porte
Déverrouillage dans la position sortie	0,8	-	0,8	-	0,8	-
Rentrée	1,6	1,2	2,0	1,6	2,0	1,6
Verrouillage dans la position rentrée	0,4	0,3	0,4	0,3	0,4	0,3
Déverrouillage dans la position rentrée	0,8	0,4	0,8	0,4	0,8	0,4
Sortie	1,2	1,2	1,6	1,5	1,6	1,5
Verrouillage dans la position sortie	0,4	-	0,4	-	0,4	-

TABLE 1 – Durées nominales des différentes opérations

Toutefois, en raison des aléas physiques, les durées réelles sont comprises autour des durées nominales avec une incertitude de ±20% : la densité de probabilité (similaire mathématiquement à une densité de probabilité de défaillance, hormis qu'elle représente un phénomène autre qu'une défaillance), pour le mouvement d'un vérin de durée nominale d, est de la forme décrite figure 18.

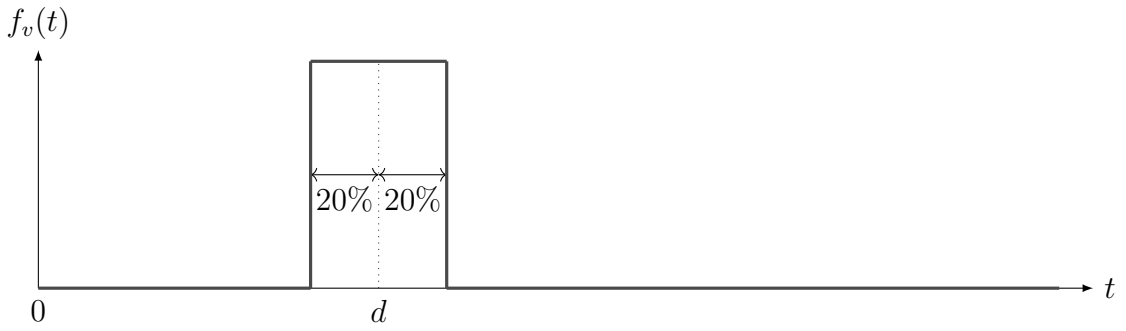


FIGURE 18 – Forme de la densité de probabilité d'un délai de mouvement d'un vérin, de durée nominale d, de durée nominale $d \pm 20\%$

Question 21 : Donner la forme de la fonction de répartition (similaire à la fiabilité) correspondante à la forme de la densité de probabilité figure 18, en précisant les coordonnées des points particuliers en fonction de la durée nominale d.

Question 22 : Proposer une fonction `delaiverin(d:float) -> float`, qui prend en argument un délai nominal d, et qui renvoie un délai aléatoirement choisi en respectant la densité de probabilité de durée de mouvement d'un vérin de durée nominale d.

V.3 Modèle de défaillance basé sur un historique de pannes

La détermination d'un modèle probabiliste associé à un composant est un problème difficile, puisqu'il s'agit de prédire la probabilité d'une défaillance dans le futur. Dans le cas d'un composant qui a déjà été utilisé dans un autre système, il est possible d'utiliser les historiques de pannes pour déterminer une loi probabiliste qui est représentative de la réalité.
Dans le domaine aéronautique, les avions subissent des inspections et des maintenances fréquentes et surtout documentées. Les constructeurs des différents systèmes et sous-systèmes peuvent ainsi suivre le cycle de vie de leurs produits.
L'un de ces constructeurs enregistre les résultats des inspections et maintenances dans une base de données, dont font partie les deux tables suivantes (figure 19) :

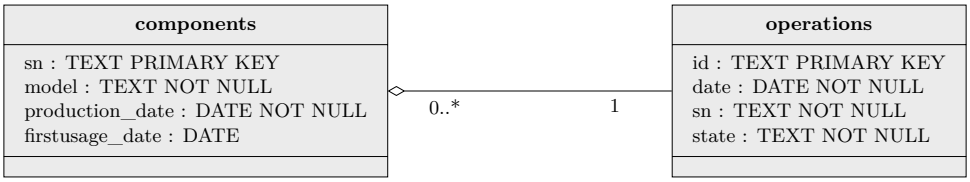


FIGURE 19 – Tables extraites de la base de données

Un extrait de chacune de ces tables est proposé figure 20.
La table `components` contient la liste des composants fabriqués par le constructeurs. Elle a pour attributs :

- `sn`, clé primaire de type texte ;
- `model`, champ de type texte, représentant le modèle (dans le catalogue du constructeur) du composant ;
- `production_date`, champ de type date, indiquant la date de production du composant ;
- `firstusage_date`, champ de type date, indiquant la date de première utilisation du composant (si elle existe).

La table `operations` contient les informations concernant toutes les inspections et maintenances sur les composants. Elle a pour attributs :

- `id`, clé primaire de type texte, générée aléatoirement ;
- `date`, champ de type date, contenant la date de l'inspection ou de la maintenance ;
- `sn`, champ de type texte, contenant le numéro de série du composant ;
- `state`, champ de type texte, contenant le type d'opération réalisée parmi « `inspection` » (le composant a été inspecté, sans maintenance nécessaire) ou « `replacement` » (le composant a dû être remplacé).

Extrait de la documentation SQL concernant la manipulation des champs de type date :

- `CURRENT_DATE()` récupérer la date courante ;
- `DATE()` extraire une date à partir d'une chaîne contenant une valeur au format DATE ;
- `DATE_ADD()` ajouter une valeur au format TIME à une date ;
- `DATE_SUB()` soustraire une valeur au format TIME à une date ;
- `DATEDIFF()` déterminer le nombre de jours entre 2 dates.

Tournez la page S.V.P.

components			
sn	model	production_date	firstusage_date
B6V78KKC7WFM2MBJ1	P500	2002-10-25	2007-05-19
5J749XC29UCG134S3	R541	2008-04-04	2017-04-09
SMRWDY7X2E6GS7N7B	R541	2009-06-02	2016-02-12
UVB7B9TT08EP6TT9S	P500	2003-07-20	2010-07-29
V8B21A34X9TTMSV2Y	P500	2015-05-28	2017-12-09
...	...	...	...

operations			
id	date	sn	state
fbbd7bbf-8a45-4aa5-9525-458d4bf5384b	2020-08-06	V8B21A34X9TTMSV2Y	inspection
f3a10a3a-17f1-41d4-a02d-2ced0e14c3d6	2021-09-03	5J749XC29UCG134S3	inspection
79a1e240-cc32-46a0-b5e7-0a8fd8ff8342	2021-09-04	V8B21A34X9TTMSV2Y	inspection
d9b2737e-77f0-4df2-a0fc-00a6e7594c37	2022-04-12	V8B21A34X9TTMSV2Y	replacement
1943148a-0ece-402c-a4ce-a6de415c9cd3	2022-04-15	B6DJPHUC2E3HLKZ70	replacement
...	...	...	...

FIGURE 20 – Extraits des tables `components` et `operations`

Afin d'identifier une loi probabiliste correspondant à la fiabilité d'un composant, il est nécessaire d'obtenir une liste des durées de bon fonctionnement des exemplaires du même modèle de ce composant, depuis leur première utilisation jusqu'à leur remplacement.

Question 23 : Écrire une requête SQL permettant d'obtenir l'ensemble des durées de bon fonctionnement des composants de modèle `P500`.

En traçant l'histogramme des durées de bon fonctionnement des exemplaires de ce modèle de composant, on obtient le graphique figure 21. Les données obtenues permettent aussi de tracer l'évolution du taux de défaillance (figure 22).

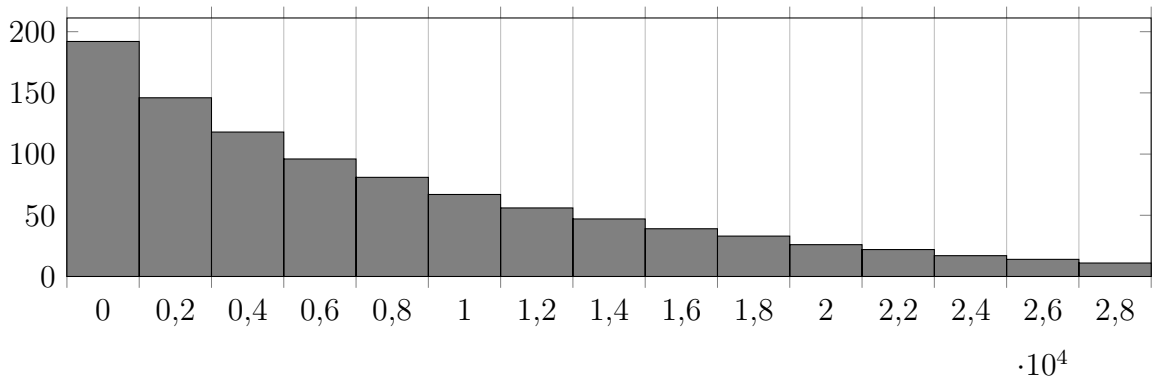


FIGURE 21 – Histogramme des durées de bon fonctionnement du composant `P500`

Question 24 : Écrire la fonction Python `LambdaExperimental(durees, t)`, prenant en argument une liste de durées de bon fonctionnement `durees` et une date `t`, et renvoyant la valeur du taux de défaillance à la date `t` avec un pas de temps `dt`.

Un memento Python est disponible en annexe.

Question 25 : Écrire la fonction Python `TracerLambdaExperimental(durees)`, prenant en argument une liste de durées de bon fonctionnement `durees` et affichant le graphe du taux de défaillance expérimental en fonction du temps.

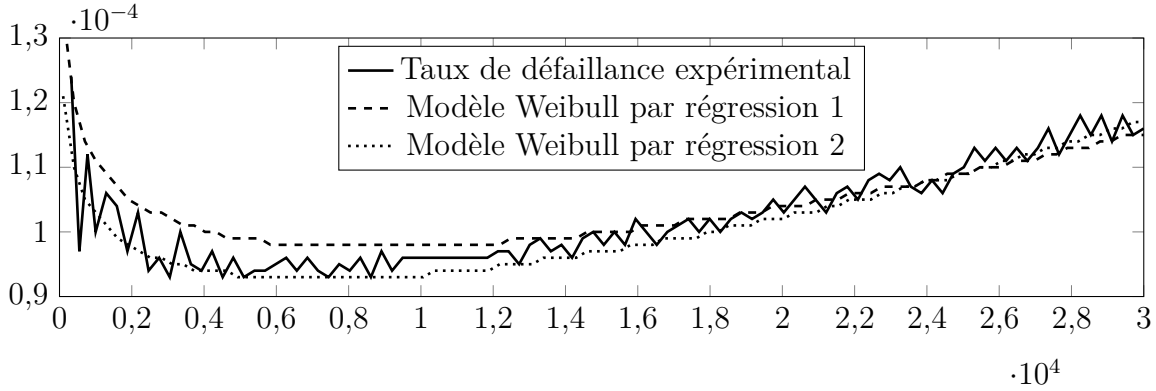


FIGURE 22 – Taux de défaillance expérimental, et modélisation du taux de défaillance par deux loi de Weibull par régression, du composant `P500`

On constate alors que le modèle exponentiel, qui utilise comme hypothèse que le taux de défaillance est constant, n'est pas représentatif du comportement de ce composant. Cette évolution du taux de défaillance, en « baignoire », est constitué de trois phases [5].

- Une première phase à fort taux de défaillance, qualifiée de « période de jeunesse » : cela correspond à la période de rodage des systèmes mécaniques, et aux défaillances dûes aux défauts de fabrication ;
- Une deuxième phase à faible taux de défaillance, à taux constant ;
- Enfin, une troisième phase avec un taux de défaillance croissant, correspondant à la « période de vieillissement » ou « d'usure ».

Un tel comportement peut-être modélisé par une loi de Weibull, soit, pour la fiabilité :

$$R_{\text{Weibull}}(t) = e^{-\frac{t^{\alpha+\mu t}}{\eta}}$$

et pour la densité de défaillance :

$$\lambda_{\text{Weibull}}(t) = \frac{(\alpha+\mu t)}{\eta} \times \left(\frac{t}{\eta}\right)^{\alpha+\mu t-1}$$

avec :

- α le paramètre de jeunesse (sans dimension) ;
- μ le paramètre de vieillesse (de dimension inverse d'un temps) ;
- η le paramètre d'échelle (de dimension temporelle).

Afin d'identifier les paramètres de la loi de Weibull correspondant à l'historique des durées de bon fonctionnement, une régression par minimisation du carré des erreurs sur la densité de défaillance va être mise en place.

La quantité à minimiser est la différence entre la densité de défaillance expérimentale $\lambda_{\text{expérimental}}$ et la densité de défaillance de la loi de Weibull λ_{Weibull}, définie par la moyenne continue des écarts au carré :

$$\text{erreur} = \frac{1}{t_{\max}} \int_{t=0}^{t_{\max}} \left(\lambda_{\text{Weibull}}(t) - \lambda_{\text{expérimental}}(t)\right)^2 \times dt$$

Tournez la page S.V.P.

Toutefois, la fonction $\lambda_{\text{expérimental}}$ étant définie à partir de données discrètes, on peut approximer l'erreur par une somme discrète :

$$\text{erreur} \approx \frac{1}{i_{\max}} \sum_{i=1}^{i_{\max}} \left(\lambda_{\text{Weibull}}(t_i) - \lambda_{\text{expérimental}}(t_i)\right)^2$$

avec t une liste discrétisée de dates.

Question 26 : Définir la fonction Python `erreur(durees, alpha, mu, eta)` renvoyant la valeur approximée de l'erreur pour une loi de Weibull de paramètres $\alpha =$`alpha`, $\mu =$`mu`, $\eta =$`eta`, par rapport à une liste de durées de bon fonctionnement `durees`, en discrétisant sur 1000 **points entre** $t = 0$ **et** $t = t_{\max}$ **($t_{\max}$ à définir).**

Une bibliothèque de calcul numérique propose plusieurs algorithmes pour minimiser une fonction : deux d'entre elles sont appliquées pour obtenir des valeurs correspondant à deux triplets de paramètres (α, μ, η) de la loi de Weibull. Les deux fonctions représentant le taux de défaillance correspondantes à ces deux triplets de paramètres sont représentées figure 22.

Question 27 : Conclure quant à la représentativité des modèles obtenus, et sur leur utilisation respective pour estimer les performances dysfonctionnelles d'un système aéronautique.

VI Simulation d'un GTS

Afin de pouvoir obtenir des mesures à partir d'un modèle GTS, il est nécessaire de pouvoir le simuler, c'est-à-dire de pouvoir en calculer une suite d'états temporisés en fonction de ses transitions. Pour cela, l'algorithme de simulation décrit en IV.2 peut être décomposé en plusieurs étapes, dont deux sont relativement complexes : la propagation des assertions (étudié en VI.1), et la gestion de l'échéancier (étudié en VI.2).
Afin de mettre en place une simulation stochastique efficace, la complexité temporelle des algorithmes utilisés sera améliorée (étudié en VI.3).

VI.1 Assertions : modélisation acausale des flux entre composants

Les GTS permettant une modélisation composant par composant d'un système, et le comportement d'un composant dépendant de ses interactions avec d'autres composants, il est nécessaire de pouvoir modéliser les échanges de matière, énergie ou information entre les différents composants lorsque ceux-ci ont une importance pour la modélisation dysfonctionnelle du système.
Le moyen de modélisation retenu dans les GTS [1] sont des variables, dites de flux, et un ensemble d'affectations conditionnelles, appelées assertions.
À la différence des variables d'état classiques, les valeurs des variables de flux sont recalculées à chaque étape par l'algorithme dit de propagation :

1. toutes les variables de flux sont définies comme indéterminées ;
2. tant que nécessaire :
   - pour chaque assertion, si sa condition est calculable (ne repose pas sur des variables de flux indéterminées) et est vraie :
     - alors, si son expression est calculable, on la calcule et on l'affecte à la variable de flux.
3. pour chaque variable de flux encore indéterminée, on lui affecte sa valeur par défaut définie dans le modèle.

Cet algorithme, appelé algorithme du point fixe, permet d'avoir différentes expressions permettant de calculer la valeur d'une variable suivant l'état du modèle. On peut ainsi modéliser un comportement de façon acausale, le sens de causalité étant déterminé (et pouvant être différent) à chaque instant.

Pour illustrer cette modélisation, on s'intéresse à un modèle d'électrovanne du circuit hydraulique. Ce modèle utilise deux variables d'état booléennes :

- `working` modélise l'état de bon fonctionnement (`True`) ou non (`False`) de l'électrovanne ;
- `position` modélise si la vanne laisse passer le fluide (`True`) ou non (`False`).

Pour modéliser les échanges avec d'autres composants, le modèle a trois variables de flux booléennes :

- `fLeft` et `fRight`, qui indique si le fluide est à haute pression (`True`) ou basse pression (`False`) sur le connecteur gauche et droite de la vanne ;
- `fConsigne`, qui permet à l'électrovanne de recevoir une consigne d'ouverture (`True`) ou de fermeture (`False`).

Tournez la page S.V.P.

Trois transitions permettent de modéliser son comportement fonctionnel et dysfonctionnel :

- `eOpen` et `eClose` permettent d'ouvrir et fermer la vanne si elle en reçoit la consigne et qu'elle est en état de bon fonctionnement (et qu'elle n'est pas déjà ouverte ou fermée). Ces opérations ont une durée constante, de valeur 1 ;
- l'électrovanne peut défaillir, suivant une loi probabiliste exponentielle, ce qui est modélisé par la transition `eFail`.

Enfin, le passage ou non du fluide est modélisé par une double assertion, dépendant de `position` :

- la première indique que si la vanne est ouverte, alors la haute pression du connecteur droit va au connecteur gauche ;
- la seconde est en sens inverse (de gauche à droite).

Ainsi, ce modèle d'électrovanne (figure 23) permet de représenter le passage du fluide de gauche à droite et de droite à gauche, et permet une modélisation acausale d'un circuit hydraulique.

Electrovanne			
`working` : Boolean `True` `position` : Boolean `False`			
`eOpen`	constante(1)	: `consigne` and `working` and not `position`	-> `position` = `True`
`eClose`	constante(1)	: not `consigne` and `working` and `position`	-> `position` = `False`
`eFail`	exponentielle(10^{-8})	: `working`	-> `working` = `False`
`fConsigne` : Boolean False `fLeft` : Boolean False `fRight` : Boolean False			
`position` : `fLeft` = `fRight` `position` : `fRight` = `fLeft`			

FIGURE 23 – Modèle GTS d'un interrupteur avec une entrée et une sortie

On s'intéresse à un circuit hydraulique composé de trois électrovannes (figure 24).

Pour réaliser le modèle GTS de ce circuit hydraulique, il suffit d'instancier plusieurs électrovannes (nommées `EV1`, `EV2a`, `EV2b`) en reliant leurs variables de flux entre elles, à l'aide d'autres assertions. On ajoute deux variables de flux booléennes, `Input` et `Output`, pour modéliser l'entrée et la sortie du circuit. Afin de modéliser l'alimentation (non étudiée ici), on ajoute une assertion `Input = True` sans condition.

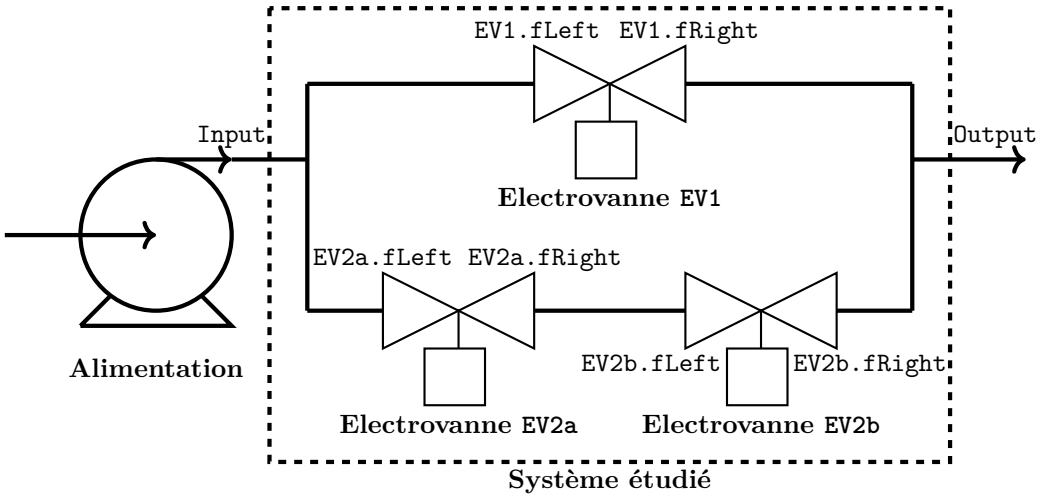


FIGURE 24 – Schéma hydraulique d'un ensemble d'électrovannes

On obtient alors l'ensemble d'assertions suivantes (table 2).

Numéro	Condition	Affectation
0	`True`	`Input = True`
1	`True`	`EV1.fLeft = Input`
2	`True`	`Input = EV1.fLeft`
3	`True`	`EV2a.fLeft = Input`
4	`True`	`Input = EV2a.fLeft`
5	`True`	`EV2a.fRight = EV2b.fLeft`
6	`True`	`EV2b.fLeft = EV2a.fRight`
7	`True`	`Output = EV1.fRight`
8	`True`	`EV1.fRight = Output`
9	`True`	`Output = EV2b.fRight`
10	`True`	`EV2b.fRight = Output`
11	`EV1.position`	`EV1.fLeft = EV1.fRight`
12	`EV1.position`	`EV1.fRight = EV1.fLeft`
13	`EV2a.position`	`EV2a.fLeft = EV2a.fRight`
14	`EV2a.position`	`EV2a.fRight = EV2a.fLeft`
15	`EV2b.position`	`EV2b.fLeft = EV2b.fRight`
16	`EV2b.position`	`EV2b.fRight = EV2b.fLeft`

TABLE 2 – Ensemble d'assertions

On suppose que les électrovannes sont dans les états :

- `EV1` : `working = True`, `position = True` ;
- `EV2a` : `working = False`, `position = False` ;
- `EV2b` : `working = True`, `position = True`.

Tournez la page S.V.P.

La détermination de la valeur de chaque variable de flux se fait en suivant l'algorithme de propagation.

Question 28 : Indiquer, dans un tableau de même forme que celui de la figure 25, la valeur de chaque variable de flux à la fin de chaque étape de l'algorithme de propagation, chaque étape ou itération d'étape correspondant à une colonne du tableau. Les variables indéfinies n'ayant pas de valeur, la case correspondante sera laissée vide.

Étape	1	2.1	2.2	...		3
`Input`						
`Output`						
`EV1.fLeft`						
`EV1.fRight`						
`EV2a.fLeft`						
`EV2a.fRight`						
`EV2b.fLeft`						
`EV2b.fRight`						

FIGURE 25 – Format de tableau pour indiquer l'évolution de la valeur des variables de flux durant l'exécution de l'algorithme de propagation.

Question 29 : Déterminer un critère d'arrêt pour la boucle de l'étape 2 de l'algorithme de propagation.

Question 30 : Définir, en Python, la méthode `GTS.Propagation()`, qui implémente l'algorithme de propagation.

Une amélioration possible de l'algorithme du point fixe consiste à ne pas reboucler sur les assertions qui ont déjà été utilisées. En conséquence, un critère d'arrêt peut être l'absence d'évolution du nombre d'assertions utilisées.

Question 31 : Indiquer les modifications à apporter à la méthode `GTS.Propagation()` pour ne pas réutiliser une assertion déjà utilisée.

VI.2 Gestions des transitions : échéancier

L'échéancier est un composant important de la simulation stochastique d'un modèle GTS. Il a pour rôles de :

- mémoriser les transitions tirables, et à quelles dates elles sont prévues ;
- retirer de l'échéancier les transitions qui ne sont plus tirables ;
- indiquer la prochaine transition à tirer, et à quelle date.

L'état courant du modèle regroupe l'ensemble des informations permettant de définir complètement l'état du modèle :

- la date de simulation actuelle ;
- l'ensemble des valeurs des variables d'état ;

- l'ensemble des valeurs des variables de flux (qui sont déterminées à partir des variables d'état, et ne sont donc pas indispensables pour définir l'état, mais sont tout de même accessibles) ;
- l'échéancier, contenant les dates de tir prévues de toutes les transitions tirables.

Toutes les données font partie de l'objet `GTS` du module `GTS`, dont la documentation est en annexe VIII.

VI.2.a Échéancier sous forme de liste

Un format de données naïf d'échéancier est un stockage sous forme de liste de couples `(date, transition)` des transitions qui sont tirables, avec leur date de tir prévue.
Après chaque tir de transition, il est nécessaire de mettre à jour l'échéancier, pour qu'il soit cohérent avec l'état courant du modèle, en contenant toutes les transitions tirables et uniquement celles-ci.

Question 32 : Définir, en Python, la méthode `GTS.MiseAJourEcheancier()`, qui met à jour en place l'échéancier (attribut `_echeancier` de la classe `GTS`).

Question 33 : Définir, en Python, la méthode `GTS.ProchaineTransition()`, qui renvoie l'indice dans l'échéancier de la prochaine transition à tirer.

On note :
- n_s le nombre de variables d'état du GTS ;
- n_t le nombre de transitions du GTS ;
- n_f le nombre de variables de flux du GTS ;
- n_a le nombre d'assertions du GTS.

Question 34 : Déterminer la complexité temporelle de `GTS.MiseAJourEcheancier()` et de `GTS.ProchaineTransition()` dans le pire cas.

Question 35 : Définir, en Python, la méthode `GTS.ProchainTir()`, qui, à partir de l'état actuel, tire la prochaine transition et met à jour l'état actuel en conséquence.

VI.2.b Échéancier stocké dans un tableau

Afin d'améliorer la complexité temporelle, il est proposé de stocker l'information sur la date de tir prévue dans un tableau contenant n_t couples de la forme [`date`, `transition`], avec `date` :
- si la transition est tirable, la date de tir prévue ;
- si la transition n'est pas tirable, `None`.

Toutes les transitions sont donc présentes dans l'échéancier, qu'elles soient tirables ou non.

Question 36 : Parmi les fonctions étudiées précédemment, redéfinir celles qui doivent l'être pour prendre en compte ce format de données.

Question 37 : Conclure quant à cette modification du format de données de l'échéancier.

Tournez la page S.V.P.

VI.3 Amélioration de la complexité temporelle

L'implémentation d'un simulateur stochastique utilisant les algorithmes vus précédemment permet, via la simulation d'un ensemble divers de modèles, de déterminer la proportion de temps passée par chaque étape de la simulation à l'aide d'un outil de profilage logiciel. Cette étude expérimentale indique que la mise à jour de l'échéancier occupe entre 35 % et 80 % du temps d'exécution, et que l'évaluation des gardes en occupe 28 % à 64 %. Afin d'améliorer les performances temporelles de la simulation de modèles, il peut donc être pertinent d'améliorer la performance temporelle de la mise à jour de l'échéancier.
Pour cela, il est possible d'utiliser une propriété liée aux systèmes modélisés. Le tir d'une transition va probablement modifier l'état du composant du système modélisé, de façon moins certaine modifier l'état du sous-système auquel il appartient, ou du moins des composants voisins, et beaucoup moins probablement modifier l'état de composants « éloignés » : il n'est donc pas forcément nécessaire de recalculer les gardes des transitions des composants éloignés, car elles n'auront pas changé de valeur. Il est ainsi possible de ne pas vérifier tout l'échéancier à chaque tir de transition, mais uniquement les transitions qui sont éventuellement concernées par les modifications engendrées par la transition qui vient d'être tirée.

Pour cela, un calcul préalable aux simulations va permettre de déterminer, pour chaque transition, quelles gardes sont possiblement affectées par leur tir, c'est-à-dire par l'application de leur action. Plusieurs conséquences de l'application d'une action sont à prendre en compte :

- L'action va modifier des variables d'état ;
- Les variables d'état vont pouvoir modifier des valeurs de variables de flux ;
- Les gardes dépendent de variables d'état et de variables de flux et peuvent donc voir leur valeur modifiée si au moins l'une d'entre elles peut être modifiée.

Il est possible de modéliser ces conséquences par un graphe orienté, tel que celui présenté figure 26.

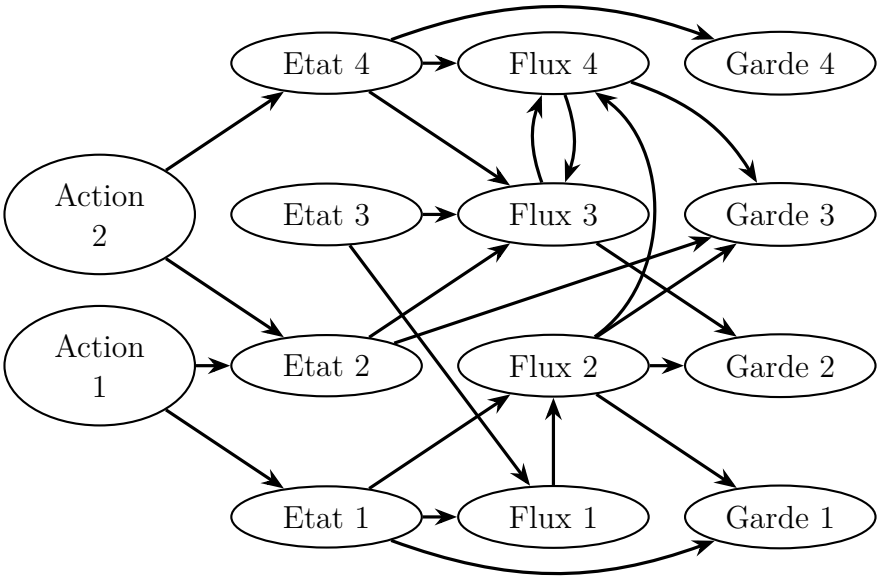


Figure 26 – Exemple de conséquences d'actions sur des gardes, via les variables d'état et les variables de flux (et des assertions)

À partir de ce graphe, via une recherche d'accessibilité en partant d'une action, on peut obtenir les gardes qui sont à recalculer après le tir de la transition correspondant à l'action de départ (figure 27).

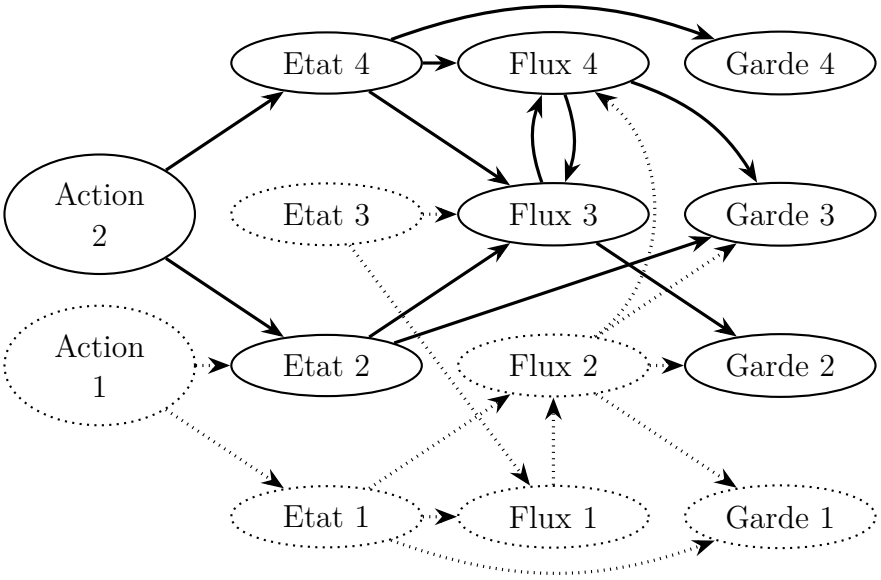


FIGURE 27 – Exemple de conséquences de l'action 2 sur des gardes

À l'aide des capacités d'inspections des objets Python, on dispose de la fonction `inspect(fonction)`, qui renvoie un couple `(write, read)` :

- `write` est une liste de variables auxquelles une valeur peut être affectée par le code de cette fonction ;
- `read` est une liste de variables dont la valeur peut être obtenue par le code de cette fonction.

On souhaite définir la fonction Python `GardesAffectees(GTS: GTS.GTS, transition: GTS.transition) -> list[transition]`, qui renvoie la liste transitions dont les gardes affectées par le tir de la transition, via une exploration du graphe reliant les actions aux gardes du modèle, à partir de la liste des transitions et de la liste des assertions du GTS. L'exploration du graphe va se faire en plusieurs étapes :

1. calcul de la liste des variables d'état affectées par le tir de la transition ;
2. calcul de la liste des variables de flux affectées, directement ou indirectement, par une modification des variables d'état, via la propagation des assertions ;
3. calcul de la liste des transitions dont la valeur de la garde peut être modifiée suite au tir de la transition.

Tournez la page S.V.P.

Ces étapes correspondent à la suite d'instructions de la fonction `GardesAffectees(GTS: GTS.GTS, transition: GTS.transition) -> list[transition]` :

```
def GardesAffectees(GTS, transition):
    # calcul de la liste Lvaretats des variables d'état affectées par le tir de la transition
    ...

    # calcul de la liste Lvarflux des variables de flux affectées, directement ou indirectement, par une modification des variables d'état, via la propagation des assertions
    ...


    # calcul de la liste Lgardes des transitions dont la valeur de la garde peut être modifiée suite au tir de la transition
    ...

    return Lgardes
```

Question 38 : Écrire les instructions Python, correspondant à la première partie de la fonction `GardesAffectees(GTS, transition)`, permettant d'obtenir la liste `Lvaretats` des variables d'état affectées par le tir de la transition.

Question 39 : Écrire les instructions Python, correspondant à la deuxième partie de la fonction `GardesAffectees(GTS, transition)`, permettant d'obtenir la liste `Lvarflux` des variables de flux affectées, directement ou indirectement, par une modification des variables d'état, via la propagation des assertions.

Question 40 : Écrire les instructions Python, correspondant à la troisième partie de la fonction `GardesAffectees(GTS, transition)`, permettant d'obtenir la liste `Lgardes` des transitions dont la valeur de la garde peut être modifiée suite au tir de la transition.

Avant la simulation, pour chacune des transitions du modèle, la fonction `GardesAffectees(transition)` est appelée et son résultat est stocké dans l'attribut `gardesaffectees` de l'objet `transition`.

Question 41 : Parmi les fonctions étudiées précédemment, indiquer, sans les implémenter, les modifications à apporter pour prendre en compte cette optimisation.

Question 42 : Indiquer s'il est possible d'utiliser les calculs déjà réalisés pour améliorer la performance d'autres étapes de la simulation.

VII Simulation stochastique

La simulation d'un modèle GTS étudiée à la partie précédente permet la mise en place d'une simulation stochastique, ou simulation de Monte-Carlo, afin d'obtenir des grandeurs statistiques de sûreté de fonctionnement.
Pour cela, un grand nombre de simulations du modèle vont être calculées, et chacune d'entre elles va permettre d'obtenir une mesure (qui dépend de la grandeur de sûreté de fonctionnement que l'on souhaite obtenir). Une étude statistique sur ces mesures est ensuite réalisée, afin de permettre d'obtenir le résultat souhaité.

Le modèle GTS du système de train d'atterrissage complet a été réalisé et implémenté. On s'intéresse à plusieurs exigences :

- Sa MTTF (Mean Time To Failure), correspondant à la moyenne des durées de bon fonctionnement du système.
  - Pour connaitre l'état de bon fonctionnement, une variable de flux booléenne nommée `BonFonctionnement` a été définie, avec une assertion correspondante : la valeur `True` indiquant que le système est en état de bon fonctionnement.
- Le respect de l'exigence (R_{11}).

VII.1 Mesure de la MTTF

On s'intéresse à la mesure de la MTTF.
Afin de mettre en place la simulation stochastique, c'est-à-dire de simuler un grand nombre de fois le modèle, il est nécessaire de déterminer :

- une condition d'arrêt de chaque simulation ;
- la grandeur mesurée à chaque simulation.

Question 43 : Déterminer la condition d'arrêt de chaque simulation, ainsi que la grandeur mesurée à chaque simulation, pour permettre la détermination de la MTTF.

La grandeur mesurée produit une série de valeurs x_i, avec $i \in [\![1\ ;\ n]\!]$ le numéro de la simulation correspondante, parmis les n simulations calculées.
La MTTF est la moyenne des durées de bon fonctionnement du système. Toutefois, comme il n'est pas possible de connaitre toutes les « vies » du système, la simulation stochastique permet d'obtenir un échantillon de ses vies, ce qui permet de calculer une estimation des paramètres statistiques de cette grandeur probabiliste, dont la qualité peut être quantifiée par la variance.
Il est donc nécessaire de calculer :

- la moyenne des valeurs de la grandeur mesurée $\overline{x_n}$;
- l'estimation de la variance des grandeurs mesurées $\mathrm{Var}(x_n)$.

La moyenne et la variance se calculent habituellement par les expressions :

$$\overline{x_n} = \frac{1}{n}\sum_{i=1}^{n} x_i \qquad \mathrm{Var}(x_n) = \overline{x_n}^2 - \overline{x_n^2}$$

Question 44 : Expliquer pourquoi ces expressions du calcul de la moyenne et de la variance ne sont pas adaptées au calcul des paramètres statistiques de la MTTF.

Tournez la page S.V.P.

L'algorithme de Welford [4] va permettre de calculer l'estimation de la moyenne et de la variance via une relation de récurrence :

$$\overline{x_j} = \overline{x_{j-1}} + \frac{x_j - \overline{x_{j-1}}}{n} \qquad M_{2,j} = M_{2,j-1} + (x_j - \overline{x_{j-1}})\,(x_j - \overline{x_j}) \qquad \mathrm{Var}(x_j) = \frac{M_{2,j}}{j-1}$$

Les résultats obtenus étant des estimations, ils ne permettent pas de connaitre les valeurs réelles, mais il est possible de déterminer, sous certaines hypothèses, un intervalle de confiance de l'estimation. Ainsi, la moyenne réelle (celle de l'ensemble des « vies » possibles du système) est probablement comprise dans l'intervalle, calculé à partir des estimateurs de moyenne et de variance :

$$\left[\overline{x_n}\left(1 - \sqrt{\mathrm{Var}(x_n)}\frac{t_\alpha}{\sqrt{n}}\right)\ ;\ \overline{x_n}\left(1 + \sqrt{\mathrm{Var}(x_n)}\frac{t_\alpha}{\sqrt{n}}\right)\right]$$

avec t_α une valeur liée au niveau de confiance désiré. Par exemple, pour un taux de confiance de 95%, $t_\alpha = 1,960$, ce qui permet de calculer un intervalle de confiance dans lequel il y a une probabilité de 95% que la moyenne réelle se situe.
On souhaite obtenir la MTTF avec un intervalle de confiance à 95% de largeur inférieure à 1% de la moyenne.

Question 45 : Définir, en Python, la fonction `SimulationMTTF(GTS)` qui, via une simulation stochastique du modèle `GTS`, renvoie une estimation de la MTTF avec un intervalle de confiance à 95% de largeur inférieur à 1% de la valeur.

VII.2 Mesure de l'exigence (R_{11})

Un modèle du contrôleur a été implémenté, permettant d'envoyer des consignes aux actionneurs à partir de l'IHM et des valeurs obtenues par les capteurs. De plus, un modèle de plan de vol (comportant des séquences de décollage et d'atterrissage, représentatifs de l'utilisation normale d'un avion) est inclu dans le modèle.
L'exigence (R_{11}) porte sur la temporisation de la séquence de sortie des roues : «Lorsque le système est dans l'état de bon fonctionnement, si la commande de sortie du train d'atterrissage a été donnée, alors les roues sont sorties et verrouillées et les portes sont fermées moins de 15 secondes après que la commande ai été donnée.»
On souhaite calculer le taux d'occurence de la situation redoutée par l'exigence (R_{11}), en occurences par heure.

Question 46 : Indiquer, sans les implémenter, les modifications à apporter aux fonctions étudiées précédemment pour permettre d'obtenir une mesure statistique du taux d'occurence de la situation redoutée par l'exigence (R_{11}).

VII.3 Résultats des simulations

Les simulations stochastiques du modèle ont permis d'obtenir les résultats suivants :

- MTTF : $1{,}818 \times 10^5$ h ;
- (R_{11}) : $5{,}5 \times 10^{-9}$ occurences par heure.

Question 47 : Interpréter ces résultats quant à la viabilité de ce système de train d'atterrissage, et proposer des solutions pour mitiger les éventuels problèmes.

VIII Annexes

Bibliographie

[1] Michel Batteux, Tatiana Prosvirnova, and Antoine Rauzy. Altarica 3.0 assertions : The whys and wherefores. *Proceedings of the Institution of Mechanical Engineers, Part O : Journal of Risk and Reliability*, 231(6):691–700, September 2017.

[2] Frédéric Boniol and Virginie Wiels. The landing gear system case study. In Frédéric Boniol, Virginie Wiels, Yamine Ait Ameur, and Klaus-Dieter Schewe, editors, *ABZ 2014 : The Landing Gear Case Study*, pages 1–18, Cham, 2014. Springer International Publishing.

[3] Antoine Rauzy. Guarded transition systems : a new states/events formalism for reliability studies. *Proceedings of The Institution of Mechanical Engineers Part O-journal of Risk and Reliability*, 222, December 2008.

[4] B. P. Welford. Note on a method for calculating corrected sums of squares and products. *Technometrics*, 4(3):419–420, 1962.

[5] Gilles ZWINGELSTEIN. Sûreté de fonctionnement - principaux concepts. *Automatique et ingénierie système*, March 2019.

Documentation du module Python GTS

class GTS.**VariableEtat**(*nom : str*, *valeurInitiale : bool*)

Variable d'état d'un GTS

_nom: str

Nom de la variable d'état

_initiale: bool

Valeur initiale de la variable d'état

_valeur: bool

Valeur actuelle de la variable d'état

setValeur(*valeur : bool*) → None

Affecte valeur à la valeur de la variable

setValeurInitiale() → None

Affecte sa valeur initiale à la variable

getValeur() → bool

Renvoie la valeur de la variable

getNom() → str

Renvoie le nom de la variable

class GTS.**VariableFlux**(*nom : str*, *valeurDefaut : bool*)

Variable de flux d'un GTS

_nom: str

Nom de la variable de flux

_defaut: bool

Valeur par défaut de la variable de flux

_valeur: bool

Valeur actuelle de la variable de flux

_definie: bool

Indique que la valeur de la variable de flux est définie

setValeur(*valeur : bool*) → None

Affecte valeur à la valeur de la variable de flux, et la marque comme définie

setValeurDefaut() → None

Affecte sa valeur par défaut à la variable de flux, et la marque comme définie

isDefinie() → bool

Teste si la variable de flux est définie

setNonDefinie() → None

Marque la variable de flux comme non définie

setDefinie() → None

Marque la variable de flux comme définie

getValeur() → bool

Renvoie la valeur de la variable

getNom() → str

Renvoie le nom de la variable

Tournez la page S.V.P.

class GTS.**Assertion**(*condition : Callable[[], bool]*, *affectation : Callable[[], None]*)

Assertion d'un GTS. Une assertion est une affectation conditionnelle et est constituée :

- d'une condition, c'est à dire une fonction booléenne portant sur des variables d'état et de flux du modèle GTS ;
- d'une affectation, c'est à dire une fonction qui affecte des valeurs à une ou plusieurs variables de flux (VariableFlux) calculées à partir de variables d'état et de flux du modèle GTS.

_condition: Callable[[], bool]

Test (fonction renvoyant un booléen), indiquant si les conditions sont réunies pour réaliser l'affectation de l'assertion.

_affectation: Callable[[], None]

Fonction d'affectation de valeurs à une ou plusieurs variables de flux (VariableFlux) calculées à partir de variables d'état et de flux du modèle GTS.

isConditionVerifiee() → bool

Teste si la condition est vérifiée.

Affecter() → None

Exécute l'affectation de l'assertion, que la condition soit vérifiée ou non.

getVariablesFluxNecessaires() → list[*GTS.VariableFlux*]

Renvoie la liste des variables de flux qui sont utilisées soit dans la condition, soit dans l'affectation

class GTS.**Transition**(*nom : str*, *garde : Callable[[], bool]*, *action : Callable[[], None]*, *temporisation : Callable[[], float]*)

Transition d'un GTS. Une transition est composée :

- d'une garde : fonction booléenne, autorisant le tir de la transition ;
- d'une action : fonction qui affecte des valeurs à une ou plusieurs variables d'état (VariableEtat) d'un GTS, à exécuter lors du tir de la transition ;
- d'une temporisation : fonction qui renvoie un float, choisi aléatoirement suivant la loi probabiliste associée à la transition ;
- et d'un nom : chaîne de caractères utilisée pour identifier la transition.

_nom: str

Nom de la transition

_garde: Callable[[], bool]

Fonction booléenne, dont le résultat autorise ou non le tir de la transition

_action: Callable[[], None]

Fonction qui affecte des valeurs à une ou plusieurs variables d'état (VariableEtat) d'un GTS, à exécuter lors du tir

_temporisation: Callable[[], float]

Fonction qui renvoie un float, choisi aléatoirement suivant la loi probabiliste associée à la transition

Garde() → bool

Renvoie le résultat calculé de la garde de la transition

Action() → None

Exécute l'action de la transition

Temporisation() → float

Renvoie une durée choisie aléatoirement suivant la loi probabiliste associée à la transition

getNom() → str

Renvoie le nom de la transition

getGarde() → Callable[[], bool]

Renvoie la fonction de garde de la transition

getAction() → Callable[[], None]

Renvoie la fonction d'action de la transition

getTemporisation() → Callable[[], float]

Renvoie la fonction de temporisation de la transition

class GTS.**GTS**(*variablesetat : list[*GTS.VariableEtat*]*, *variablesflux : list[*GTS.VariableFlux*]*, *transitions : list[*GTS.Transition*]*, *assertions : list[*GTS.Assertion*]*)

Guarded Transition System

Un GTS est composé :

- d'un ensemble de variables d'état (VariableEtat) ;
- d'un ensemble de variables de flux (VariableFlux) ;
- d'un ensemble de transitions (Transition) ;
- d'un ensemble d'assertions (Assertion) ;
- et, pour permettre sa simulation, d'une date de simulation et d'un échéancier.

_variablesetat: list[*GTS.VariableEtat*]

Liste des variables d'état du GTS

_variablesflux: list[*GTS.VariableFlux*]

Liste des variables de flux du GTS

_transitions: list[*GTS.Transition*]

Liste des transitions du GTS

_assertions: list[*GTS.Assertion*]

Liste des assertions du GTS

_date: float

Date actuelle de la simulation du GTS

_echeancier: list[tuple[float, *GTS.Transition*]]

Echéancier (définition dans le sujet)

getVariablesEtat() → list[*GTS.VariableEtat*]

Renvoie la liste des variables d'état du GTS

getVariablesFlux() → list[*GTS.VariableFlux*]

Renvoie la liste des variables de flux du GTS

getTransitions() → list[*GTS.Transition*]

Renvoie la liste des transitions du GTS

getAssertions() → list[*GTS.Assertion*]

Renvoie la liste des assertions du GTS

getDate() → float

Renvoie la date de simulation du GTS

getEcheancier() → list[tuple[float, *GTS.Transition*]]

Renvoie l'échéancier du GTS

setDate(*date : float*) → None

Définie la date de simulation du GTS

Propagation() → None

Propage les assertions du GTS

MiseAJourEcheancier() → None

Met à jour l'échéancier du GTS

ProchaineTransition() → int

Renvoie l'indice de la transition qui est la prochaine à devoir être tirée

ProchainTir() → None

Réalise un pas de simulation

Tournez la page S.V.P.

Structure d'une trame CAN

Une trame CAN à une longueur allant de 44 à 108 bits, et est composée des champs suivants (figure 28) :

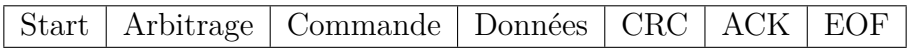


FIGURE 28 – Format d'une trame CAN

- SOF (Start Of Frame) : début de trame, sur 1 bit.
- Champ d'arbitrage : identificateur (11 bits) et bit de RTR (Remote Transmission Request), dominant (= 0) pour une trame de données et récessif (= 1) pour une trame de requête.
- Champ de commande : 6 bits, dont les 4 derniers représentent le DLC (Data Length Code), c'est-à-dire le nombre d'octets présents dans le champ Data.
- Champ de données (Data) : entre 0 à 8 octets. Dans le cas d'une trame de requête, ce champ est vide.
- Champ de CRC (Contrôle de Redondance Cyclique) : 16 bits (15 + 1). La séquence calculée CRC est contenue dans les 15 premiers bits tandis que le dernier bit est un délimiteur de fin de champ de CRC (bit toujours récessif).
- Champ d'acquittement (acknowledge) : 2 bits, laissés libres par la station émettrice. Le premier correspond à l'acquittement par l'ensemble des nœuds ayant reçu le message (0 signifie acquitté et donc, pas d'erreur détectée ; 1 signifie non acquitté, émission d'une trame d'erreur). Le second est toujours récessif et délimite l'« acknowledge ».
- EOF (End of Frame) : fin de trame, 7 bits récessifs.

Entre chaque trame, une pause (IFS) de 3 bits doit être respectée.

Mémento Python 3

Mémento dérivé de celui d'Éric Ducasse & Jean-Luc Charles (Version AM−1.5) : https://savoir.ensam.eu/moodle/course/view.php?id=1428
Forme inspirée initialement du mémento de Laurent Pointal : https://perso.limsi.fr/pointal/python:memento

Aide (F1)

`dir(nom)` *liste des noms des méthodes et attributs de* ***nom***
`help(nom)` *aide sur l'objet* ***nom***
`help("nom_module.nom")` *aide sur l'objet* **nom** *du module* **nom_module**

Types de base

Entier, décimal, complexe, booléen, rien

(objets non mutables)

```
int      783   0   -192   0b010   0o642   0xF3
                   zéro    binaire  octal   hexadécimal
float    9.23  0.0   -1.7e-6   ( -1,7×10⁻⁶ )
complex  1j  0j  2+3j  1.3-3.5e2j
bool     True  False
NoneType None   (une seule valeur : « rien »)
```

Objets itérables

- **Conteneurs numérotés** *(listes, tuples, chaînes de caractères)*

```
list   [1,5,9]    ["abc"]    []    ["x",-1j,["a",False]]
tuple  (1,5,9)    ("abc",)   ()    11,"y",[2-1j,True]
str    "abc"      "z"        ""
```

Conteneurs hétérogènes
Objets non mutables
expression juste avec des virgules → `tuple`
Singleton
Objet vide

Nombre d'éléments			
`len(`*objet*`)` donne :	3	1	0	3

- **Itérateurs** *(objets destinés à être parcourus par* `in`*)*

`range(n)` : pour parcourir les *n* premiers entiers naturels, de 0 à *n*-1 inclus.
`range(n,m)` : pour parcourir les entiers naturels de *n* inclus à *m* exclu par pas de 1.
`range(n,m,p)` : pour parcourir les entiers naturels de *n* inclus à *m* exclu par pas de *p*.
`reversed(`*itérable*`)` : pour parcourir un objet itérable à l'envers.
`enumerate(`*itérable*`)` : pour parcourir un objet itérable en ayant accès à la numérotation.
`zip(`*itérable1*`,`*itérable2*`,...)` : pour parcourir en parallèle plusieurs objets itérables.

Identificateurs

Noms d'objets, de fonctions, de modules, de classes, etc.

`a...zA...Z_` suivi de `a...zA...Z_0...9`
- accents possibles mais à éviter
- mots clés du langage interdits
- distinction casse min/MAJ

☺ `a toto x7 y_max BigOne`
☹ ~~`8y and for`~~

Affectation/nommage

Symbole : `=`

affectation ⇔ **association** d'un *nom* à un objet

```
nom_objet = <expression>
```
1) *évaluation de l'expression de droite pour créer un objet*
2) *nommage de l'objet créé*

```
x = 1.2 + 8 + sin(y)
```

Affectations multiples

```
<n noms> = <itérable de taille n>
u,v,w = 1j,"a",None
a,b = b,a        échange de valeurs
```

Affectations combinée avec une opération ✧

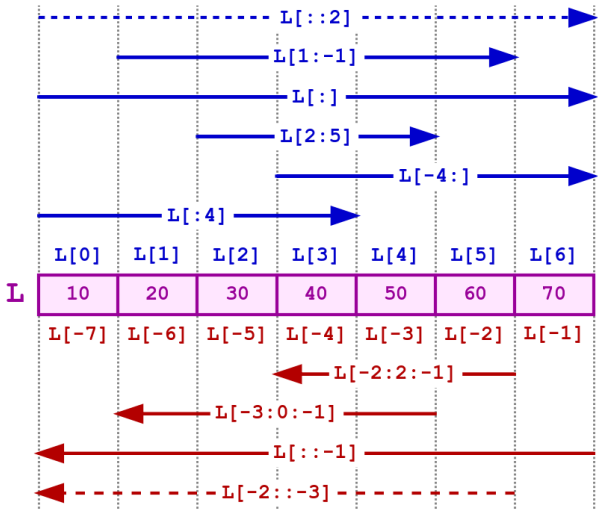
`x ✧= c` *équivaut à :* `x = x✧c`

Suppression d'un nom

`del x` *l'objet associé disparaît seulement s'il n'a plus de nom, par le mécanisme du « ramasse-miettes »*

Parcours de conteneurs numérotés

index à partir de 0



- Accès à chaque **élément** par `L[`*index*`]`
  - `L[0]` → `10` ⇒ *le premier*
  - `L[1]` → `20` ⇒ *le deuxième*
  - `L[-1]` → `70` ⇒ *le dernier*
  - `L[-2]` → `60` ⇒ *l'avant-dernier*
- Accès à une **partie** par `L[`*début inclus*`:`*fin exclue*`:`*pas*`]`
  - `L[2:5]` → `[30,40,50]` ⇒ *indices 2,3 et 4*
  - `L[:4]` → `[10,20,30,40]` ⇒ *les 4 premiers*
  - `L[-4:]` → `[40,50,60,70]` ⇒ *les 4 derniers*
  - `L[::2]` → `[10,30,50,70]` ⇒ *de 2 en 2*
  - `L[:]` *tous : copie superficielle du conteneur*
  - `L[::-1]` *tous, de droite à gauche*
  - `L[-2::-3]` → `[60,30]` ⇒ *de -3 en -3 en partant de l'avant-dernier*

Sur les listes *(conteneurs mutables),* ***suppression*** *d'un élément ou d'une partie par* `del`*, et* ***remplacement*** *par* `=`

`del L[4]` effet sur la liste `L` similaire à `L.pop(4)` → `L` *devient* `[10,20,30,40,60,70]`
`del L[1::2]` suppression des éléments d'indices impairs → `L` *devient* `[10,30,50,70]`
`L[4] = 99` → `L` *devient* `[10,20,30,40,99,60,70]`
`L[1::2] = "abc"` itérable ayant le même nombre d'éléments que la partie à remplacer, sauf si le pas vaut 1 → `L` *devient* `[10,"a",30,"b",50,"c",70]`
`L[1:-1] = range(2)` → `L` *devient* `[10,0,1,70]`

Conteneurs : opérations génériques

`len(c)` `min(c)` `max(c)` `sum(c)`
`nom in c` → booléen, test de présence dans `c` d'un élément identique (comparaison `==`) à ***nom***
`nom not in c` → *booléen, test d'absence*
`c1 + c2` → *concaténation*
`c * 5` → *5 répétitions* (`c+c+c+c+c`)
`c.index(nom)` → *position du premier élément identique à* ***nom***
`c.index(nom,idx)` → *position du premier élément identique à* ***nom*** *à partir de la position* ***idx***
`c.count(nom)` → *nombre d'occurrences*

Opérations sur listes

modification « en place » de la liste `L` *originale ces méthodes ne renvoient rien en général*

`L.append(nom)` *ajout d'un élément à la fin*
`L.extend(itérable)` *ajout d'un itérable converti en liste à la fin*
`L.insert(idx,nom)` *insertion d'un élément à la position* ***idx***
`L.remove(nom)` *suppression du premier élément identique (comparaison* `==`*) à* ***nom***
`L.pop()` *renvoie et supprime le dernier élément*
`L.pop(idx)` *renvoie et supprime l'élément à la position* ***idx***
`L.sort()` *ordonne la liste (ordre croissant)*
`L.sort(reverse=True)` *ordonne la liste par ordre décroissant*
`L.reverse()` *renversement de la liste*
`L.clear()` *vide la liste*

Chaînes de caractères

Caractères spéciaux :
`"\n"` *retour à la ligne*
`"\t"` *tabulation*
`"\\"` *« backslash \ »*
`"\""` *ou* `'"'` *guillemet "*
`"'"` *ou* `'\''` *apostrophe '*

Exemple :

```
ch = "X\tY\tZ\n1\t2\t3"
print(ch) affiche :   X   Y   Z
                      1   2   3
print(repr(ch)) affiche :
'X\tY\tZ\n1\t2\t3'
```

`r"dossier\sd\nom.py"` → `'dossier\\sd\\nom.py'`
Le préfixe `r` *signifie "raw string" (tous les caractères sont considérés comme de vrais caractères)*

Méthodes sur les chaînes

Une chaîne n'est pas modifiable ; ces méthodes renvoient en général une nouvelle chaîne ou un autre objet

```
"nomfic.txt".replace(".txt",".png") → 'nomfic.png'
"b-a-ba".replace("a","eu") → 'b-eu-beu'   remplacement de toutes les occurrences
" \tUne phrase.\n ".strip() → 'Une phrase.'   nettoyage début et fin
"des  mots\tespacés".split() → ['des','mots','espacés']
"1.2,4e-2,-8.2,2.3".split(",") → ['1.2','4e-2','-8.2','2.3']
" ; ".join(["1.2","4e-2","-8.2","2.3"]) → '1.2 ; 4e-2 ; -8.2 ; 2.3'
ch.lower() minuscules, ch.upper() majuscules, ch.title(), ch.swapcase()
```
Recherche de position : `find` *similaire à* `index` *mais renvoie* `-1` *en cas d'absence, au lieu de soulever une erreur*

```
"image.png".endswith(".txt") → False
"essai001.txt".startswith("essai") → True
```

Formatage

La méthode `format` ***sur une chaîne contenant*** `"{<numéro>:<format>}"` **(accolades)**

```
"{} ~ {}".format("pi",3.14) → 'pi ~ 3.14'          ordre et formats par défaut
"{1:} -> {0:}{1:}".format(3,"B") → 'B -> 3B'       ordre, répétition
"essai_{:04d}.txt".format(12) → 'essai_0012.txt'   entier, 4 chiffres, complété par des 0
"L : {:.3f} m".format(0.01) → 'L : 0.010 m'        décimal, 3 chiffres après la virgule
"m : {:.2e} kg".format(0.012) → 'm : 1.20e-02 kg'  scientifique, 2 chiffres après la virgule
```


Tournez la page S.V.P.

Mémento Python 3

Blocs d'instructions

```
instruction parente :
    bloc d'instructions 1…
    ⋮
    instruction parente :
        bloc d'instructions 2…
        ⋮
instruction suivant le bloc 1
⋮
```

indentation

Symbole : puis indentation (4 espaces en général)

Instruction conditionnelle

```
if booléen1 :
    bloc d'instructions 1…
    ⋮
elif booléen2 :
    bloc d'instructions 2…
    ⋮
else :
    dernier bloc…
    ⋮
```

Blocs `else` et `elif` facultatifs.

`if/elif x` : si `x` n'est pas un booléen équivaut en Python à `if/elif bool(x)` : (voir conversions).

Définition de fonction

Une fonction fait des actions et renvoie un ou plusieurs objets, ou ne renvoie rien.

```
def nom_fct(x,y,z=0,a=None) :
    bloc d'instructions…
    ⋮
    if a is None :
        ⋮
    else :
        ⋮
    return r0,r1,…,rk
```

`x` et `y` : arguments positionnels, obligatoires

`z` et `a` : arguments optionnels avec des valeurs par défaut, nommés

Plusieurs `return` possibles (interruptions)

Une absence de `return` signifie qu'à la fin, `return None` (rien n'est renvoyé)

Appel(s) de la fonction

Autant de noms que d'objets renvoyés

```
a0,a1,…,ak = nom_fct(-1,2)
b0,b1,…,bk = nom_fct(3.2,-1.5,a="spline")
```

True/False Logique booléenne

- *Opérations booléennes*
  - `not A` « *non A* »
  - `A and B` « *A et B* »
  - `A or B` « *A ou B* »
  - `(not A) and (B or C)` *exemple*
- *Opérateurs renvoyant un booléen*
  - *nom1* `is` *nom2* *2 noms du même objet ?*
  - *nom1* `==` *nom2* *valeurs identiques ?*
  - **Autres comparateurs :** `<` `>` `<=` `>=` `!=` (≠)
  - *nom_objet* `in` *nom_iterable*
    l'itérable ***nom_iterable*** *contient-il un objet de valeur identique à celle de* ***nom_objet*** *?*

Boucle conditionnelle

Bloc d'instructions répété tant que *condition* est vraie

```
while condition :
    instructions…
    ⋮
    (valeurs impliquées dans condition modifiées)
    ⋮
```

oui / non

Attention aux boucles sans fin !

Contrôle de boucle

`break` *sortie immédiate*

`continue` *itération suivante*

Exemple

```
from random import randint
somme,nombre = 0,0
while somme < 100 :
    nombre += 1
    somme += randint(1,10)
print(nombre,";",somme)
```

Le nombre d'itérations n'est pas connu à l'avance

Boucle par itérations

Bloc d'instructions répété ***pour chaque*** élément de l'*itérable*, désigné par *nom*

```
for nom in itérable :
    instructions…
    ⋮
```

suivant / fini

Variantes avec parcours en parallèle

```
for a,b in itérable :
    bloc d'instructions
```
Itérations sur des couples

```
for numéro,nom in enumerate(itérable) :
    bloc d'instructions
```
Numérotation en parallèle, à partir de 0

```
for numéro,nom in enumerate(itérable,d) :
    bloc d'instructions
```
Numérotation en parallèle, à partir de *d*

```
for e1,e2,… in zip(itérable1,itérable2,…) :
    bloc d'instructions
```
Parcours en parallèle de plusieurs itérables ; s'arrête dès qu'on arrive à la fin de l'un d'entre eux

Conversions

`bool(x)` → `False` *pour* `x` : `None`, `0` (`int`), `0.0` (`float`), `0j` (`complex`), *itérable vide*
→ `True` *pour* `x` : *valeur numérique non nulle, itérable non vide*

`int("15")` → `15`
`int("15",7)` → `12` *(base 7)*
`int(-15.56)` → `-15` *(troncature)*
`round(-15.56)` → `-16` *(arrondi)*
`float(-15)` → `-15.0`
`float("-2e-3")` → `-0.002`
`complex("2-3j")` → `(2-3j)`
`complex(2,-3)` → `(2-3j)`
`list(x)` *Conversion d'un itérable en liste*
exemple : `list(range(12,-1,-1))`
`sorted(x)` *Conversion d'un itérable en liste ordonnée (ordre croissant)*
`sorted(x,reverse=True)` *Conversion d'un itérable en liste ordonnée (ordre décroissant)*
`tuple(x)` *Conversion en tuple*
`"{}".format(x)` *Conversion en chaîne de caractères*
`ord("A")` → `65` ; `chr(65)` → `'A'`

Liste en compréhension

- *Inconditionnelle / conditionnelle*

```
L = [ f(e) for e in itérable ]
L = [ f(e) for e in itérable if b(e)]
```

Fichiers texte

N'est indiquée ici que l'ouverture avec fermeture automatique, au format normalisé UTF-8.

Le « *chemin* » d'un fichier est une chaîne de caractères (voir module `os` ci-dessous)

- *Lecture intégrale d'un seul bloc*

```
with open(chemin,"r",encoding="utf8") as f:
    texte = f.read()
```

- *Lecture ligne par ligne*

```
with open(chemin,"r",encoding="utf8") as f:
    lignes = f.readlines()
```

(Nettoyage éventuel des débuts et fins de lignes)

```
lignes = [c.strip() for c in lignes]
```

- *Écriture dans un fichier*

```
with open(chemin,"w",encoding="utf8") as f:
    f.write(début)…
    ⋮
    f.write(suite)…
    ⋮
    f.write(fin)
```

Gestion basique d'exceptions

```
try :
    bloc à essayer
except :
    bloc exécuté en cas d'erreur
```

Affichage

```
x,y = -1.2,0.3
print("Pt",2,"(",x,",",y+4,")")
```
→ `Pt 2 = ( -1.2 , 4.3 )`

Un espace est inséré à la place de chaque virgule séparant deux objets consécutifs. Pour mieux maîtriser l'affichage, utiliser la méthode de formatage `str.format`

Saisie

```
s = input("Choix ? ")
```

`input` renvoie toujours une ***chaîne de caractères*** ; la convertir si besoin vers le type désiré

Importation de modules

Module `mon_mod` ⇔ Fichier `mon_mod.py`

- *Importation d'objets par leurs noms*

```
from mon_mod import nom1,nom2
```

- *Importation avec renommage*

```
from mon_mod import nom1 as n1
```

- *Importation du module complet*

```
import mon_mod
    ⋮
… mon_mod.nom1 …
```

- *Importation du module complet avec renommage*

```
import mon_mod as mm
    ⋮
… mm.nom1 …
```

Programme utilisé comme module

- ***Bloc-Test*** *(non lu en cas d'utilisation du programme* `mon_mod.py` *en tant que module)*

```
if __name__ == "__main__" :
    Bloc d'instructions
    ⋮
```

Mathématiques

- *Opérations*
  `+` `-` `*` `/`
  `**` *puissance* `2**10` → `1024`
  `//` *quotient de la division euclidienne*
  `%` *reste de la division euclidienne*
- *Fonctions intrinsèques*
  `abs(x)` *valeur absolue / module*
  `round(x,n)` *arrondi du* `float` `x` *à* `n` *chiffres après la virgule*
  `pow(a,b)` *équivalent à* `a**b`
  `pow(a,b,p)` *reste de la division euclidienne de* a^b *par* `p`
  `z.real` → *partie réelle de* `z`
  `z.imag` → *partie imaginaire de* `z`
  `z.conjugate()` → *conjugué de* `z`

Quelques modules internes de Python (The Python Standard Library)

sys

```
import sys
```

`sys.path` → *liste des chemins des dossiers contenant des modules Python*

`sys.path.append(chemin)` *Ajout du* ***chemin absolu*** *d'un dossier contenant des modules*

`sys.platform` → *nom du système d'exploitation*

os

```
import os
```

`os.getcwd()` → ***Chemin absolu*** *du « répertoire de travail »* (working directory), *à partir duquel on peut donner des* ***chemins relatifs***.

Chemin absolu : chaîne commençant par une lettre majuscule suivie de ":" (Windows), ou par "/" (autre)

Chemin relatif par rapport au répertoire de travail `wd` *:*
nom de fichier ⇔ fichier dans `wd`
`"."` ⇔ `wd` ; `".."` ⇔ *père de* `wd`
`"../.."` ⇔ *grand-père de* `wd`
`"sous-dossier/image.png"`

Le séparateur `"/"` fonctionne pour tous les systèmes, au contraire du `"\\"`

`os.listdir(chemin)` → *liste des sous-dossiers et fichiers du dossier désigné par* `chemin`.

`os.path.isfile(chemin)` → Booléen : est-ce un fichier ?
`os.path.isdir(chemin)` → Booléen : est-ce un dossier ?

```
for sdp,Lsd,Lnf in os.walk(chemin) :
```
Parcourt récursivement chaque sous-dossier, de chemin relatif `sdp`, *dont la liste des sous-dossiers est* `Lsd` *et celle des fichiers est* `Lnf`

time

```
from time import time
debut = time()
: (instructions)
duree = time() - debut
```

Évaluation d'une durée d'exécution, en secondes

Mémento Python 3

Aide numpy/scipy

`np.info(nom_de_la_fonction)`

Fonctions mathématiques

`import numpy as np`

En calcul scientifique, il est préférable d'utiliser les fonctions de `numpy`*, au lieu de celles des modules basiques* `math` *et* `cmath`*, puisque les fonctions de* `numpy` *sont vectorisées : elle s'appliquent aussi bien à des scalaires (*`float`*,* `complex`*) qu'à des vecteurs, matrices, tableaux, avec des durées de calculs minimisées.*

`np.pi`, `np.e` → *Constantes* π *et e*

`np.abs`, `np.sqrt`, `np.exp`, `np.log`, `np.log10`, `np.log2` → `abs` *, racine carrée, exponentielle, logarithmes népérien, décimal, en base 2*

`np.cos`, `np.sin`, `np.tan` → *Fonctions trigonométriques (angles en radians)*

`np.degrees`, `np.radians` → *Conversion radian→degré, degré→radian*

`np.arccos`, `np.arcsin` → *Fonctions trigonométriques réciproques*

`np.arctan2(y,x)` → *Angle dans* $]-\pi,\pi]$

`np.cosh`, `np.sinh`, `np.tanh` *(trigonométrie hyperbolique)*

`np.arcsinh`, `np.arccosh`, `np.arctanh`

Tableaux numpy.ndarray : généralités

Un tableau `T` *de type* `numpy.ndarray` *(« n-dimensional array ») est un conteneur homogène dont les valeurs sont stockées en mémoire de façon séquentielle.*

`T.ndim` → *« dimension d » = nombre d'indices (1 pour un vecteur, 2 pour une matrice)*

`T.shape` → *« forme » = plages de variation des indices, regroupées en* `tuple` $(n_0, n_1, \dots, n_{d-1})$ *: le premier indice varie de 0 à* n_0-1*, le deuxième de 0 à* n_1-1*, etc.*

`T.size` → nombre d'éléments, valant $n_0 \times n_1 \times \cdots \times n_{d-1}$

`T.dtype` → *type des données contenues dans le tableau (*`np.bool`, `nb.int32`, `np.uint8`, `np.float`, `np.complex`, `np.unicode`, *etc.)*

générateurs

shp est la forme du tableau créé, data_type le type de données contenues dans le tableau (`np.float` *si l'option* `dtype` *n'est pas utilisée)*

`T = np.empty(shp,dtype=data_type)` → *pas d'initialisation*

`T = np.zeros(shp,dtype=data_type)` → *tout à* `0`*/*`False`

`T = np.ones(shp,dtype=data_type)` → *tout à* `1`*/*`True`

- ***Tableaux de même forme que*** `T` *(même type de données que* `T` *si ce n'est pas spécifié) :*

`S = np.empty_like(T,dtype=data_type)`

`S = np.zeros_like(T,dtype=data_type)`

`S = np.ones_like(T,dtype=data_type)`

Vecteurs

- ***Un vecteur*** `V` ***est un tableau à un seul indice***
- *Comme pour les listes,* `V[i]` *est le* $(i+1)^{ème}$ *coefficient, et l'on peut extraire des sous-vecteurs par :* `V[:2]`, `V[-3:]`, `V[::-1]`, etc.

Si `c` est un nombre, les opérations `c*V`, `V/c`, `V+c`, `V-c`, `V//c`, `V%c`, `V**c` *se font sur chaque coefficient*

Si `U` est un vecteur de même dimension que `V`, les opérations `U+V`, `U-V`, `U*V`, `U/V`, `U//V`, `U%V`, `U**V` *sont des opérations terme à terme*

- *Produit scalaire :* `U.dot(V)` *ou* `np.dot(U,V)` *ou* `U@V`

générateurs

`np.linspace(a,b,n)` → *n valeurs régulièrement espacées de a à b (bornes incluses)*

`np.arange(xmin,xmax,dx)` → *de* x_{min} *inclus à* x_{max} *exclu par pas de dx*

Statistiques

Sans l'option `axis`*, un tableau est considéré comme une simple séquence de valeurs*

`T.max()`, `T.min()`, `T.sum()`

`T.argmax()`, `T.argmin()` *indices séquentiels des extremums*

`T.sum(axis=d)` → *sommes sur le (d-1)-ème indice*

`T.mean()`,`T.std()`,`T.std(ddof=1)` *moyenne, écart-type*

`V = np.unique(T)` *valeurs distinctes, sans ou avec les effectifs*

`V,N = np.unique(T,return_counts=True)`

`np.cov(T)`, `np.corrcoef(T)` *matrices de covariance et de corrélation ; T est un tableau k×n qui représente n répétitions du tirage d'un vecteur de dimension k ; ces matrices sont k×k.*

Tirages pseudo-aléatoires

Modules `random` *et* `numpy.random`

`import random`

`random.random()` → *Valeur flottante dans l'intervalle [0,1[(loi uniforme)*

`random.randint(a,b)` → *Valeur entière entre a inclus et b inclus (équiprobabilité)*

`random.choice(L)` → *Un élément de la liste L (équiprobabilité)*

`random.shuffle(L)` → `None`*, mélange la liste L « en place »*

`import numpy.random as rd`

`rd.rand(n0,...,nd-1)` → *Tableau de forme* $(n_0, \dots, n_{d-1})$*, de flottants dans l'intervalle [0,1[(loi uniforme)*

`rd.randint(a,b,shp)` → *Tableau de forme shp, d'entiers entre a inclus et b exclu (équiprobabilité)*

`rd.randint(n,size=d)` → *Vecteur de dimension d, d'entiers entre 0 et n-1 (équiprobabilité)*

`rd.choice(Omega,n,p=probas)` → *Tirage avec remise d'un échantillon de taille n dans Omega, avec les probabilités probas*

`rd.choice(Omega,n,replace=False)` → *Tirage sans remise d'un échantillon de taille n dans Omega (équiprobabilité)*

`rd.normal(m,s,shp)` → *Tableau de forme shp de flottants tirés selon une loi normale de moyenne m et d'écart-type s*

`rd.uniform(a,b,shp)` → *Tableau de forme shp de flottants tirés selon une loi uniforme sur l'intervalle [a , b [*

Conversions

Le passage maîtrisé `list` ↔ `ndarray` *permet de bénéficier des avantages des 2 types*

`T = np.array(L)` → *Liste en tableau, type de données automatique*

`T = np.array(L,dtype=data_type)` → *Idem, type spécifié*

`L = T.tolist()` → *Tableau en liste*

`new_T = T.astype(data_type)` → *Conversion des données*

`S = T.flatten()` → *Conversion en vecteur (la séquence des données telles qu'elles sont stockées en mémoire)*

`np.unravel_index(ns,T.shape)` *donne la position dans le tableau* `T` *à partir de l'index séquentiel* `ns` *(indice dans* `S`*)*

Matrices

générateurs

`np.eye(n)` → *matrice identité d'ordre n*

`np.eye(n,k=d)` → *matrice carrée d'ordre n avec des 1 décalés de d vers la droite par rapport à la diagonale*

`np.diag(V)` → *matrice diagonale dont la diagonale est le vecteur V*

- ***Une matrice*** `M` ***est un tableau à deux indices***
- `M[i,j]` *est le coefficient de la* $(i+1)$*-ième ligne et* $(j+1)$*-ième colonne*
- `M[i,:]` *est la* $(i+1)$*-ième ligne,* `M[:,j]` *la* $(j+1)$*-ième colonne,* `M[i:i+h,j:j+l]` *une sous-matrice h×l*
- ***Opérations : voir Vecteurs***
- *Produit matriciel :* `M.dot(V)` *ou* `np.dot(M,V)` *ou* `M@V`

`M.transpose()`,`M.trace()` → *transposée, trace*

Matrices carrées uniquement (algèbre linéaire) :

`import numpy.linalg as la` *("Linear algebra")*

`la.det(M)`, `la.inv(M)` → *déterminant, inverse*

`vp = la.eigvals(M)` → `vp` *vecteur des valeurs propres*

`vp,P = la.eig(M)` → `P` *matrice de passage*

`la.matrix_rank(M)`, `la.matrix_power(M,p)`

`X = la.solve(M,V)` → *Vecteur solution de* `M X = V`

Tableaux booléens, comparaison,tri

`B = (T==1.0)`

`B = (abs(T)<=1.0)` → `B` *est un tableau de booléens, de même forme que* `T`

`B = (T>0)*(T<1)` *Par exemple* `B*np.sin(np.pi*T)` *renverra un tableau de* $\sin(\pi x)$ *pour tous les coefficients x dans* $]0,1[$ *et de 0 pour les autres*

`B.any()`, `B.all()` → *booléen « Au moins un* `True` *», « Que des* `True` *»*

`indices = np.where(B)` → *tuple de vecteurs d'indices donnant les positions des* `True`

`T[indices]` → *extraction séquentielle des valeurs*

`T.clip(vmin,vmax)` → *tableau dans lequel les valeurs ont été ramenées entre* v_{min} *et* v_{max}

`np.allclose(T1,T2)` → *booléen indiquant si les tableaux sont numériquement égaux*

Intégration numérique

`import scipy.integrate as spi`

`spi.odeint(F,Y0,Vt)` → *renvoie une solution numérique du problème de Cauchy* $\mathbf{Y}'(t) = \mathbf{F}(\mathbf{Y}(t),t)$*, où* $\mathbf{Y}(t)$ *est un vecteur d'ordre n, avec la condition initiale* $\mathbf{Y}(t_0) =$ `Y0`*, pour les valeurs de t dans le vecteur* `Vt` *commençant par* t_0*, sous forme d'une matrice n×k*

`spi.quad(f,a,b)[0]` → *renvoie une évaluation numérique de l'intégrale :* $\int_a^b f(t)\,dt$

Tournez la page S.V.P.

Mémento Python 3

Graphiques Matplotlib

```
import matplotlib.pyplot as plt
```

plt.figure(*mon_titre*, figsize=(*W*,*H*)) *crée ou sélectionne une figure dont la barre de titre contient mon_titre et dont la taille est W×H (en inches, uniquement lors de la création de la figure)*

plt.plot(X,Y,*dir_abrg*) *trace le nuage de points d'abscisses dans* **X** *et d'ordonnées dans* **Y** *; dir_abrg est une chaîne de caractères qui contient une couleur ("r"-ed, "g"-reen, "b"-lue, "c"-yan, "y"-ellow, "m"-agenta, "k" black), une marque (voir ci-dessous) et un type de ligne ("" pas de ligne, "-" plain, "--" dashed, ":" dotted, ...)*

▪ **Options courantes :**

- **linewidth=**... *épaisseur du trait (0 pour aucun trait)*
- **color=**... *couleur du trait :* (*r*,*g*,*b*) *ou* "m","c",...
- **label=**... *étiquette pour la légende*
- **dashes=**... *style de pointillé (liste de longueurs)*
- **marker=**... *forme de la marque :*

"o" "." "*" ">" "<" "^" "v" "s" "d" "D" "p" "h" "H" "8" "1" "2" "3" "4" "+" "x"

- **markersize=**... *taille de la marque*
- **markeredgewidth=**... *épaisseur du contour*
- **markeredgecolor=**... *couleur du contour*
- **markerfacecolor=**... *couleur de l'intérieur*

plt.axis("equal"), plt.grid() *repère orthonormé, quadrillage*

plt.xlim(a,b), plt.ylim(a,b) *plages d'affichage ; si a > b, inversion de l'axe*

plt.xlabel(*axe_x*,size=*s*,color=(*r*,*g*,*b*)), plt.ylabel(*axe_y*,...) *étiquettes sur les axes, en réglant la taille s et la couleur de la police de caractères (r, g et b dans [0,1])*

plt.legend(loc="best",fontsize=*s*) *affichage des labels des "plot" en légende*

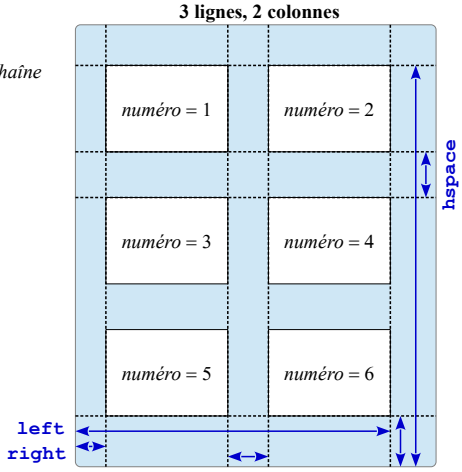
plt.show() *affichage des différentes figures et remise à zéro*

plt.twinx() *bascule sur une deuxième échelle des ordonnées apparaissant à droite du graphique*

plt.xticks(Xt), plt.yticks(Yt) *réglage des graduations des axes*

plt.subplot(nbL,nbC,numero) *début de tracé dans un graphique situé dans un tableau de graphiques à* **nbL** *lignes,* **nbC** *colonnes ;* **numero** *est le numéro séquentiel du graphique dans le tableau (voir ci-contre).*

plt.subplots_adjust(left=*L*,right=*R*,bottom=*B*,top=*T*,wspace=*W*,hspace=*H*) *ajustement des marges (voir ci-contre)*



plt.title(*Titre_du_graphique*) *rajout d'un titre au graphique en cours de tracé*

plt.suptitle(*Titre_général*) *rajout d'un titre à la fenêtre de graphiques*

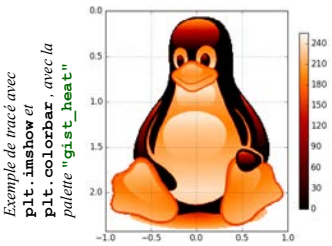
plt.text(*x*,*y*,*texte*,fontdict=*dico*,horizontalalignment=*HA*,verticalalignment=*HV*) *tracé du texte texte à la position (x,y), avec réglage des alignements (HA∈{"center","left","right"}, HV∈{"center","top","bottom"}) ; dico est un dictionnaire (voir ci-contre)*

plt.axis("off") *suppression des axes et du cadre*

plt.imshow(T,interpolation="none",extent=(*gauche*,*droite*,*bas*,*haut*)) *tracé d'une image pixélisée non lissée à partir d'un tableau* **T** *(n_L×n_C×4 format **RGBA**, n_L×n_C×3 format **RGB**) ; l'option* **extent** *permet de régler la plage correspondant à l'image ((–0.5, n_C–0.5, n_L–0.5, –0.5) par défaut)*

plt.imshow(T,interpolation="none",vmin=v_{min},vmax=v_{max},cmap=*palette*,extent=(*gauche*,*droite*,*bas*,*haut*)) *tracé d'une image pixélisée non lissée à partir d'un tableau* **T** *rectangulaire n_L×n_C correspondant à des niveaux de gris sur la plage [v_{min}, v_{max}], avec la palette de couleurs palette : voir* https://matplotlib.org/examples/color/colormaps_reference.html

plt.colorbar(schrink=*c*) *Affichage de l'échelle des couleurs du tracé précédent sur la plage [v_{min}, v_{max}]*



Exemple de tracé avec **plt.imshow** *et* **plt.colorbar** *, avec la palette "gist_heat"*

Dictionnaires

Un dictionnaire **D**, *de type* **dict** *(type itérable), se présente sous la forme :*
{clef_0:valeur_0,clef_1:valeur_1,...}
On peut accéder aux clefs par **D.keys()**, *aux valeurs par* **D.values()**, *et obtenir un itérateur sur les couples par* **D.items()** :

Parcours des clefs et valeurs du dictionnaire :

```
for key,val in D.items() :
    (bloc d'instructions)
```

Extraire une valeur par sa clef : **D[key]**
Compléter le dictionnaire :

```
D[new_key] = new_val
dict1.update(dict2)
```

Supprimer un entrée : **del D[key]**
Dictionnaire vide : **D = dict()** *ou* **D = {}**

***Exemple pour la fonction* plt.text :**

```
{ "family" : "Courier New",
  "weight" : "bold",
  "style" : "normal",
  "size" : 18,
  "color" : (0.0,0.5,0.8) }
```

Calcul formel avec sympy

Il est conseillé d'utiliser un **notebook jupyter** (voir https://jupyter.readthedocs.io/en/latest/) avec en en-tête pour avoir de belles formules mathématiques à l'écran les instructions ci-contre :

```
import sympy as sb
sb.init_printing()
```

Nombres exacts

Rationnels : **sb.Rational(2,7)** ou **sb.S(2)/7**

Irrationnels :

- **sb.sqrt(2)** $\sqrt{2}$
- **sb.pi, sb.E** π et e
- **sb.I** i tel que $i^2=-1$

Fonction indéfinie

f = sb.Function("f")

L'infini

sb.oo → ∞

Symboles

- **sb.symbols("a,a_0,a^*")** → (a, a_0, a^*) (**tuple**)
- **sb.symbols("x", real=True)** → *réel* x
- **sb.symbols("y", nonzero=True)** → *réel* y *non nul*
- **sb.symbols("j,k", integer=True)** → *entiers relatifs* j *et* k
- **sb.symbols("m", integer=True, positive=True)** → $m \in \mathbb{N}^*$
- **sb.symbols("n", integer=True, nonzero=True)** → $n \in \mathbb{Z}^*$
- **sb.symbols("s", zero=False)** → *symbole* s *non nul*
- **sb.symbols(..., nonnegative=True)** → *positif ou nul*
- **sb.symbols(..., negative=True)** → *strictement négatif*
- **sb.symbols(..., nonpositive=True)** → *négatif ou nul*
- **sb.symbols(..., complex=True)** → *complexe*
- **sb.symbols(..., imaginary=True)** → *imaginaire pur*

Fonctions mathématiques

- **sb.sqrt, sb.exp** → *Racine carrée, exponentielle,*
- **sb.log, sb.factorial** → *Logarithme népérien, factorielle*
- **sb.cos, sb.sin, sb.tan** → *Fonctions trigonométriques*
- **sb.acos, sb.asin, sb.atan** → *Fonctions trigonométriques réciproques*
- **sb.atan2(y,x)** → *Angle dans]–π,π]*
- **sb.cosh, sb.sinh, sb.tanh** *(trigonométrie hyperbolique)*
- **sb.acosh, sb.asinh, np.atanh**

Manipulation d'expressions

Expressions symboliques **A** *et* **B**

- **A+B, A-B, A*B, A/B,** *etc.* → *Opérations mathématiques*
- **A.diff(x), A.diff(x,*n*), A.diff(x,3,y,2,z)** → $\frac{\partial A}{\partial x}$, $\frac{\partial^n A}{\partial x^n}$, $\frac{\partial^6 A}{\partial x^3 \partial y^2 \partial z}$
- **A.expand(), A.simplify()** → *Développer, simplifier*
- **A.factor(), A.together()** → *Factoriser, réduire une fraction*
- **A.collect(x)** → *Regrouper les termes par rapport à x*
- **A.apart(x)** → *Décomposition en élément simples par rapport à x*

Égalités et équations

Expressions symboliques **A** *et* **B**

- **A == B** → *Booléen : identité parfaite des expressions*
- **A.equals(B)** → *Booléen : égalité après simplifications*
- **sb.Eq(A,B)** → ***Équation :*** *booléen seulement si l'équation peut être identifiée comme vraie ou fausse*
- **sb.Ne(A,B)** *ou* **Lt, Le, Gt, Ge** → *Inéquations* ≠ < ≤ > ≥