ALASCA — Architecture logicielles avancées pour les systèmes cyber-physiques autonomiques

© Jacques Malenfant

Master informatique, spécialité STL – UFR 919 Ingénierie

Sorbonne Université
Jacques.Malenfant@lip6.fr

Cours 10

De l'entité de contrôle — méthodologie et développement

Objectifs pédagogiques du cours 10

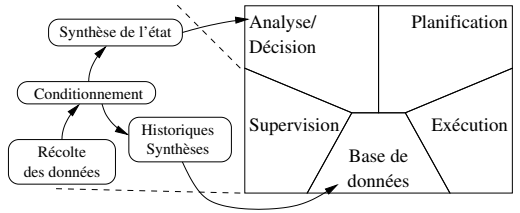
- Au-delà des notions plus conceptuelles vues au cours précedent, acquérir des savoir-faire pratiques pour la mise en œuvre des entités de contrôle.
- Apprendre comment programmer les contrôles et les différentes fonctions de l'entité de contrôle.
- Introduire des approches permettant de rendre la composition entre entités adaptables et entités de contrôle plus robustes.

Construction des entités de contrôle

- Dans ce cours, nous allons nous intéresser aux entités de contrôle *individuelles* implantant un contrôle *unique* ou la *fusion* de plusieurs contrôles en *un seul modèle* de contrôle.
- La construction des contrôleurs *composites* ou *coordonnés* sera essentiellement étudiée au cours 12, mais nous introduirons ici des éléments utiles à la conception des contrôleurs *individuels* en prévision de leur composition et de leur coordination.
- Bien que les entités de contrôle puissent être implantées de manière réactive ou délibérative, nous allons reprendre un examen structuré autour des quatre grandes fonctions des gestionnaires autonomiques d'IBM : supervision, analyse/décision, planification et exécution des adaptations.
- Nous traiterons enfin de problématiques au niveau du processus de développement et du cycle de vie : test, validation, auto-configuration.

Plan

Principales sous-fonctions de la supervision



1. Récolte des données via les interfaces capteurs des entités adaptables et les sondes placées sur l'environnement.
2. Conditionnement des données préparatoire à la suite des traitements (filtrage, cohérence temporelle, etc.).
3. Synthèse de l'état des entités adaptables et de l'environnement tel que vu par le modèle de décision qui va assurer la suite immédiate du traitement dans la boucle de contrôle.
4. Calcul de synthèses et d'historiques à partir des données courantes pour un usage ultérieur.

Besoins de la fonction de supervision

- En contact avec les interfaces capteurs des entités adaptables, l'entité de contrôle peut prendre en charge certaines des fonctions vues en étudiant ces dernières et ce, en utilisant les mêmes techniques pour les compléter et les corriger le cas échéant :
  - obtenir ou fournir des données de bonne qualité (correction des biais, filtrage, ...) ;
  - produire des données d'état les plus fiables possibles (fusion de capteurs, ...).
- Elle peut aussi :
  - assurer la cohérence temporelle des ensembles de données de capteurs utilisées conjointement dans la suite du contrôleur ;
  - construire progressivement des modèles du monde et de l'environnement à partir du flôt des données de capteurs (contrôleurs délibératifs).

Collecte des données

- Modes (rappel) : passif versus actif
  - actif (mode « *pull* ») : plutôt contrôleur périodique ou apériodique
  - passif (mode « *push* ») : plutôt contrôleur par événements
- Difficultés de la collecte active versus passive :
  - En mode actif, attention aux délais d'appel à distance, en particulier si les données à collecter sont nombreuses (passage à l'échelle problématique).
  - Mode passif peut suppléer, mais alors il faut tenir le rythme :
    - adopter des fils d'exécution (*thread*) séparés pour la réception et le traitement,
    - faire un minimum d'opérations à chaque réception.
  - Réglage et configuration : cas du contrôleur périodique
    - En général, la période du contrôleur détermine la période de lecture des données de supervision.
    - Difficultés liés à l'hétérogénéité des sources $\Rightarrow$ découpler la période du contrôle de la période de lecture ou adopter des modes passifs ou mixtes.

Cohérence temporelle des ensembles de mesures I

- Rappel : besoin pour utiliser conjointement des données qu'elles portent sur un même état *i.e.*, temporellement cohérentes.
- Problématique pas uniquement due aux instants de *collecte* des données, mais liée également aux instants de *mesure*.
- Techniques de la fenêtre temporelle $[t_{inf}, t_{sup}]$

$d_{1,1}$ - $t_{1,1}$	$d_{2,1}$ - $t_{2,1}$		$d_{n,1}$ - $t_{n,1}$
$d_{1,2}$ - $t_{1,2}$	$d_{2,2}$ - $t_{2,2}$		$d_{n,2}$ - $t_{n,2}$
$d_{1,3}$ - $t_{1,3}$	$d_{2,3}$ - $t_{2,3}$		$d_{n,3}$ - $t_{n,3}$
$d_{1,4}$ - $t_{1,4}$	$d_{2,4}$ - $t_{2,4}$		$d_{n,4}$ - $t_{n,4}$
...	...		...

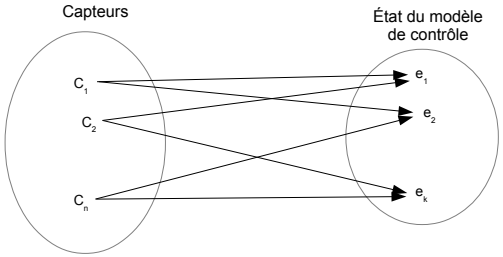
avec
$t_{inf} \leq t_{1,3} \leq t_{sup}$,
$t_{inf} \leq t_{2,2} \leq t_{sup}$,
...,
$t_{inf} \leq t_{n,3} \leq t_{sup}$.

Cohérence temporelle des ensembles de mesures II

- La technique de la fenêtre temporelle *ne garantit pas* d'obtenir des données temporellement cohérentes car pour des bornes t_{inf} et t_{sup} données, il reste *possible* qu'on n'obtienne *jamais* un ensemble de données $d_1, \ldots, d_n$ dans la fenêtre requise.
- Il faut donc aussi configurer correctement les algorithmes de mesure et la fréquence de collecte pour s'assurer d'avoir des ensembles de données cohérentes temporellement.
- La technique de la fenêtre temporelle peut être étendue pour intégrer d'autres contraintes comme celles sur la fraîcheur des données.

Synthèse de l'état du modèle de contrôle

- Conçu pour faciliter la détermination d'une politique, le modèle de contrôle peut utiliser des variables d'état différentes des propriétés adaptables, dont des données d'historiques.
- Synthèse de l'état : faire le pont entre les données capteurs $c_1, \ldots, c_n$ et les états $e_1, \ldots, e_k$ du modèle de contrôle, sachant que ce n'est pas nécessairement du un à un.



Construction d'historiques et de synthèses

- Dans beaucoup de techniques de décision, il est nécessaire de conserver d'une manière ou d'une autre une mémoire des données de capteurs ou d'état.
  - statistiques (moyenne, écart-type, ...), fonctions approximées (tabulations, interpolations, régressions, ...), réseaux bayésiens, classeurs, réseaux de neurones (*deep learning*), etc.
- Domaine du traitement des données « massives », mais en ligne.
- Problématiques à prendre en compte :
  - Taille mémoire : attention, à raison d'un mot de 32 bits par seconde, on accumule près de 350Ko/jour et 125Mo/an !
  - Conservation ou non des données brutes : réduction des besoins en mémoire, mais perte d'information (surtout pour des utilisations futures non prévues au moment de la collecte).
  - Lourdeur des traitements : éviter de bloquer la collecte (fil d'exécution séparé et à moindre priorité) et de retarder la décision du cycle de contrôle.

Plan

Principes généraux

- À chaque instant de décision, le contrôleur synthétise un état $s \in S$ grâce aux capteurs, calcule l'action d'adaptation par la loi de commande $\gamma(s) \in A$ puis l'applique grâce aux actionneurs.
- En pratique, il y a bien deux aspects distincts :
  1. La *détermination* de la fonction γ, employant des techniques d'automatique, de décision ou d'optimisation (cf. cours 13).
  2. L'*expression* (ou la *programmation*) de γ dans un contrôleur.
- Dans les transparents qui viennent, nous allons donc nous concentrer sur la *programmation* des contrôleurs réactifs.
- Deux principales familles de techniques de programmation :
  1. programmation impérative classique et
  2. des langages spécialisés.
- Trois principales familles de techniques de représentation de γ :
  1. des approches de l'analyse numérique et des statistiques,
  2. des approches à base de règles et
  3. des approches qualitatives à base de logique floue.

Utilisation de la programmation impérative classique

- C'est l'approche la plus évidente !
- Par exemple, un contrôleur PID se résume souvent au pseudo-code suivant :

```
integrale:= 0 ; iprecedent:= 0 ;

i:= lireCapteur() ;
integrale:= integrale + delta*i ;
sortie:= kp*i + ki*integrale + kd*((i - iprecedent)/delta) ;
iprecedent:= i ;
ecrireActionneur(sortie) ;
```

- Les paramètres propres à la configuration du contrôleur, `delta`, `kp`, `ki` et `kd`, devraient être représentés par des variables (constantes), à la fois :
  - pour faciliter la maintenance et la réutilisation du code,
  - pour permettre leur mise à jour dynamique, ce qui revient à une forme de contrôle adaptatif sur laquelle nous reviendrons en parlant du méta-contrôle.

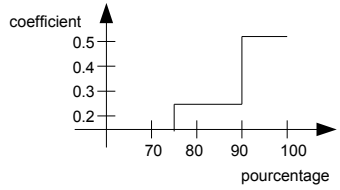
Autre exemple : allocation de mémoire I

- Principe : augmenter ou diminuer la taille de la mémoire m d'une machine virtuelle en fonction du pourcentage p de mémoire utilisée après les passes de ramasse-miettes.
- Comment déterminer la relation entre ce pourcentage et la quantité de mémoire à ajouter ? Idée : discrétisation
  - $p \geq 90 \Rightarrow$ augmenter la mémoire de 50%,
  - $90 > p \geq 75 \Rightarrow$ augmenter la mémoire de 25%.

Politique en code impératif :

```
p:= lireCapteurPourcentage() ;
m:= lireCapteurMemoire() ;
if (p >= 90.0 and disponible(0.5m))
then sortie = 0.5m ;
else if (p >= 75.0 and disponible(0.25m))
then sortie = 0.25m ;
else sortie = 0.0 ;
ecrireActionneur(sortie) ;
```

Politique sous la forme d'une fonction :



Langages spécialisés

- La programmation impérative des contrôles multiples supposent d'organiser les calculs afférents aux différents contrôles dans un même cycle de contrôle.
- Les contrôleurs réactifs sont souvent programmés à partir de blocs de code (en anglais *code elements* ou *codels*) devant être ordonnancés en fonction :
  - de leurs contraintes temporelles (délais de réaction, ...),
  - de leurs relations de précédences (production/consommation de données, ...)
  - et d'autres contraintes d'accès aux ressources.
- Quelques langages spécialisés ont été proposés pour faciliter l'expression des relations utilisées pour l'ordonnancement, que nous allons illustrer par deux exemples :
  - les langages synchrones en programmation temps-réel, et
  - les architectures comportementales de Brooks en robotique.

Les langages synchrones I

- Les langages synchrones ont été proposés pour programmer des systèmes réactifs, c'est-à-dire réagissant en permanence avec un délai borné à des événements se produisant dans leur environnement par des actions sur ce dernier.
- Les langages synchrones sont fondés sur un ensemble commun d'hypothèses dont les principales sont :
  - l'existence d'un temps discret où tous les événements se produisent, les réactions calculées et exécutées ;
  - les calculs et les réactions sont instantanés, c'est-à-dire ont une durée d'exécution nulle.
- Il existe deux familles (« duales ») de langages synchrones :
  1. les langages à flôt de données, et
  2. les langages dirigés par les événements, que nous allons plus particulièrement survoler.

  Les événements peuvent être des données reçues des capteurs.

Les langages synchrones II

- Par leurs hypothèses, les langages dirigées par les événements :
  - supposent des interventions à des instants discrets où l'ensemble des événements qui se sont produits depuis la dernière intervention sont « consommés » ;
  - le calcul des réactions (*i.e.*, actions d'adaptation) s'assimile alors à l'exécution d'un automate où les événements provoquent des transitions et l'exécution de *codels* émettant de nouveaux événements ou appelant des actionneurs.
- Par rapport à une expression directe sous la forme d'automates, les langages synchrones facilitent la production de ces derniers par une expression supposée plus facile à écrire, qui sera ensuite compilée vers l'automate correspondant.
  L'automate est un peu un langage intermédiaire du compilateur de langage synchrone...

Exemple : compteur de vitesse

```
module Speedometer:
input Second, Meter ; output Speed
in loop
     var Distance:= 0: integer
     in do every Meter do
          Distance:= Distance + 1
        end every
        upto Second ;
        emit Speed(Distance)
     end var
   end loop
end module.
```

```
module SpeedSupervisor
input Speed, Meter ; output TooFast
in signal Speed: integer in [
     run Speedometer
     ||
     every Speed do
       if ?Speed > MaxSpeed then
         emit TooFast
       end if
     end every
   ]
   end signal
end module
```

- Parmi les nombreuses constructions non-illustrées par cet exemple, il est possible d'appeler des *codels* externes.
- L'opérateur de parallélisme '| |' sert à exprimer un parallélisme *logique*, permettant de composer plusieurs contrôles sans les ordonner explicitement et sans qu'il y ait exécution parallèle.

Sémantique d'Esterel

- Le compilateur Esterel, à partir des
  - relations de production et de consommation d'événements,
  - des *codels* extractibles du programme Esterel et
  - des *codels* externes appelés,

  produit un automate qui, à chaque instant, consomme tous les événements présents, déclenche les transitions, produit puis exécute les réactions, générant possiblement de nouveaux événements, visibles au prochain instant.
- Le code compilé est un ordonnancement statique des codels produit par un ordonnanceur implicite statique, évitant le recours à un ordonnanceur générique dynamique (à priorité, EDF, ...).
- Plutôt que d'utiliser un automate explicite, l'automate est généralement compilé en code C itératif équivalent.
  C est alors le langage cible du compilateur...
- Il existe (maintenue ?) une implantation d'Esterel en Java (SugarCubes).

Architectures réactives en robotique (rappels)

- Au cours 9, l'*architecture réactive en robotique* a été présentée comme la réalisation directe des fonctions de perception et d'action, rejetant la fonction de planification au profit d'un calcul d'action sans mémoire uniquement à partir des données courantes des capteurs.
- Dans un robot réactif, chaque contrôle est vu comme un *comportement* liant certains capteurs à certains actionneurs pour réaliser l'une des fonctionnalités du robot, comme l'évitement d'obstacle, le déplacement aléatoire ou le suivi d'un corridor.
- La vision de l'architecture réactive consiste à créer des robots complexes en *composant différents comportements*, qui sont en réalité des contrôles au sens de nos entités de contrôle.
- Nous allons maintenant étudier l'*architecture de Brooks*, une des premières et des plus représentatives, en particulier pour tirer des leçons sur leur manière de programmer et composer différents contrôles réactifs.

Architecture à subsomption de Brooks I

- Architecture réactive utilisant trois types de modules de base (assimilables à des *codels*) :
  1. Modules de *lecture des capteurs*, lisant les capteurs physiques pour émettre des données vers les autres modules.
  2. Module de *calcul*, consommant des données et en produisant de nouvelles pour les autres modules.
  3. Modules *actionneurs*, consommant des données des autres modules pour les transmettre aux actionneurs physiques.

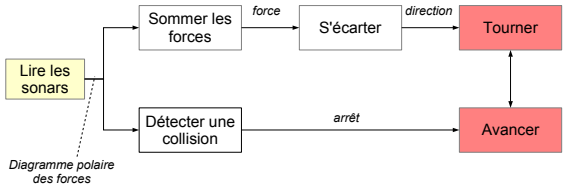
  Les modules sont regroupés en *comportements*, puis empilés en *couches* des plus basiques aux plus évolués, un comportement plus évolué pouvant *intercepter* les réactions produites par les plus basiques, d'où la *subsomption*.
- Ces modules sont connectés par des liens de transmission de données, d'un émetteur vers un récepteur ; les modules de lecture sont lancés en début de cycle, les autres étant déclenchés par la présence d'une donnée sur un lien en entrée.

Architecture à subsomption de Brooks II

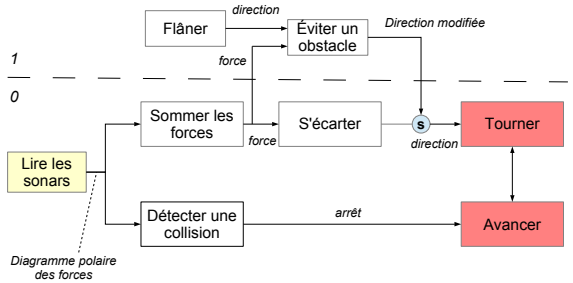
- Des nœuds de combinaison consomment des données d'entrée et produisent des données en sortie sans calculer mais avec deux sémantiques principales :
  1. *Inhibition* : la présence d'une donnée émise par le module subsumant bloque la transmission d'une donnée du module subsumé.
  2. *Suppression* : la présence d'une donnée émise par le module subsumant remplace une donnée du module subsumé.
- La programmation d'un robot réactif selon l'approche de Brooks consiste donc :
  1. à identifier et définir des comportements,
  2. à programmer les modules nécessaires,
  3. à les assembler en comportements avec leurs liens de production/consommation de signaux/données, puis
  4. à composer les comportements hiérarchiquement en utilisant des nœuds d'inhibition et de supression pour lier les comportements de niveaux plus élevés à ceux des niveaux moins élevés.

Exemple : un robot suivant un corridor I

1. Niveau 0 : avancer sans entrer en collision avec les obstacles.

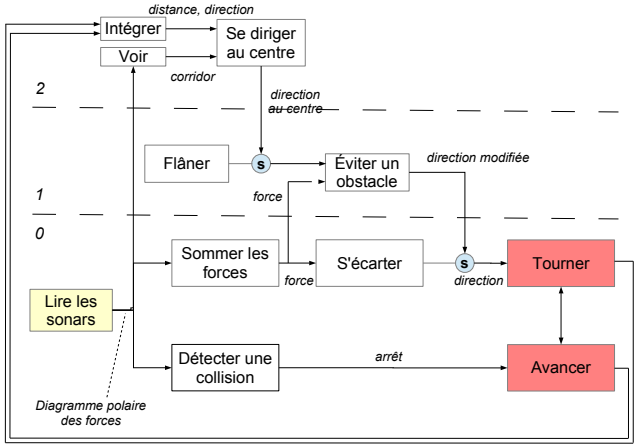


2. Niveau 1 : flâner, tout en évitant les obstacles.



Exemple : un robot suivant un corridor II

1. Niveau 3 : suivre un corridor en évitant les obstacles.



Quelques observations sur les architectures à subsomption

1. Par la synchronisation des lectures de capteurs en début de cycle, l'architecture à subsomption résoud le problème du partage des capteurs entre différents contrôles.
2. Par les relations de précédences, cette architecture permet l'ordonnancement des tâches à l'intérieur du cycle et la vérification de son ordonnançabilité globale.
3. Par son organisation en couches et les mécanismes d'inhibition et de suppression, elle compose les comportements avec des priorités explicites, *mais* cette technique est purement *ad hoc* et donc *très difficile de prouver des propriétés sur le contrôleur*.
4. Par la composition des signaux d'actions, comme les champs de potentiels, cette architecture (comme d'autres) autorise aussi le déclenchement de plusieurs actions se combinant en fin de cycle avant l'envoi des commandes aux actionneurs.

   Ex.: des directions calculées comme des vecteurs sont simplement additionnées pour produire la nouvelle direction.

Représenter γ : approches fonctionnelles et statistiques

- Dans les exemples précédents, la loi de commande a été exprimée par du code or dans beaucoup de cas, la loi est trop complexe pour être facilement programmée de cette manière.
- Une loi de commande $\gamma : \mathbf{S} \rightarrow \mathbf{A}$ peut être connue par un nuage de points, suite à des expérimentations et des mesures à l'exécution, par exemple.
- Si γ est continue, trouver une représentation à partir de points $\{(\mathbf{s}_i, \mathbf{a}_i)\}$ est une problématique étudiée en analyse numérique.
- Types de représentations :
  - paramétriques : fonctions approximées (linéaires, splines, séries de Taylor, séries de Fourier, ...) possiblement décomposées en morceaux (linéaires ou splines par morceaux, ...) ;
  - non-paramétriques : tableaux, structures de données associatives (tables de hachage), diagrammes de décision binaires, règles floues, réseaux de neurones, ...

Approximations paramétriques

- Il s'agit d'utiliser une forme de fonction prédéfinie avec des paramètres $p_0, \ldots, p_k$ puis à trouver *globalement* les valeurs de ces paramètres qui collent (le mieux) avec l'ensemble des points.
- Comme le problème devient vite surcontraint, il faut généralement utiliser des approximations qui ne passent pas nécessairement par tous les points.
- Les approximations *par morceaux* permettent d'améliorer les choses en trouvant les meilleures valeurs *localement à chaque morceau*, n'utilisant qu'un nombre limité de points à chaque fois. Ex.: une fonction linéaire par morceau définie sur les points (x_i, y_i) en ordre croissant de x_i :

$$y = \frac{(y_{i+1} - y_i)}{(x_{i+1} - x_i)} x + \frac{(x_{i+1} y_i - x_i y_{i+1})}{(x_{i+1} - x_i)},\ x_i \leq x \leq x_{i+1}$$

Approximations non paramétriques

- Les représentations non-paramétriques les plus simples sont les tableaux ou les tables de hachage qui fonctionnent plutôt dans le cas des *fonctions discrètes* où *tous les points sont connus*.
- Dans le cas de *fonctions continues*, une *interpolation* deviendra nécessaire, comme dans les approximations paramétriques. Comme dans ce cas, on peut se fonder sur les intervalles et utiliser des diagrammes de décision binaires ou toutes autres structures arborescentes pour accéder plus rapidement à l'intervalle contenant le s pour lequel la valeur de a est requise.
- Ces approches simples deviennent cependant trop coûteuses en mémoire lorsque le nombre de points devient trop grand.
- Pour réduire la taille mémoire nécessaire, on utilise des représentations capables de *minimiser* la quantité d'informations nécessaire, dont par exemple les *réseaux de neurones profonds* qui connaissent une forte popularité.

Approches statistiques

- Les systèmes réels sont rarement déterministes :
  - des erreurs ou des dérives aléatoires peuvent entacher la trajectoire d'évolution de l'état ;
  - les conséquences des actions peuvent être entachées d'aléa ;
  - les meilleures lois de commande ou politiques peuvent consister à prendre des décisions aléatoires.
- Pour les deux premiers cas, il faut prendre en compte de l'aléa dans le modèle d'évolution de l'état du système.
  - Par exemple, en étudiant les contrôleurs PID nous avons supposé que l'évolution du système pouvait être modélisée par une fonction $p(k+1) = a_0 p(k) + a_1 u(k)$.
  - Comment déterminer les paramètres comme a_0, a_1 ?
  - Des expérimentations peuvent fournir des nuages de points.
  - Il faut ensuite estimer les paramètres en fonction de ces points.
- Les réseaux de neurones peuvent représenter des modèles stochastiques, mais nous allons maintenant plutôt voir des méthodes purement statistiques.

Modèles de régression

- Des modèles relativement simples qui s'appuient sur la notion de *corrélation* entre variables aléatoires :
  - une corrélation existe entre variables aléatoires observées sur une même population lorsque les variations de ces variables se produisent dans le même sens ou en sens contraire.
- *Idée* : considérer le contrôle U et l'état du système P comme deux variables aléatoires corrélées.
- Le *coefficient de corrélation linéaire* est une mesure de l'*intensité de la liaison* qui existe entre ces variables aléatoires. Soient les couples (x_i, y_i), on calcule le coefficient r par la formule :

$$r = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2}\sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2}} \qquad \bar{x} = \frac{\sum_{i=1}^{n} x_i}{n} \qquad \bar{y} = \frac{\sum_{i=1}^{n} y_i}{n}$$

- r varie dans l'intervalle $[-1, 1]$, et plus $|r|$ est proche de 1, plus la corrélation est forte.

Droite de régression linéaire

- Lorsqu'une corrélation linéaire existe, ce qui s'apprécie par tests statistiques, la *droite de régression linéaire* est la droite $y = \alpha_0 + \alpha_1 x$ qui s'ajuste le mieux possible aux points $(x_i, y_i), i = 1, ..., n$ observés.
- La droite des moindres carrés est celle dont les paramètres minimise la somme des carrés des écarts des y_i observés à la droite. Elle est calculée à partir des observations par :

$$\alpha_1 = \frac{\sum_{i=1}^{n}(x_i-\bar{x})(y_i-\bar{y})}{\sum_{i=1}^{n}(x_i-\bar{x})^2} \qquad \bar{x} = \frac{\sum_{i=1}^{n} x_i}{n}$$

$$\alpha_0 = \bar{y} - b_1\bar{x} \qquad \bar{y} = \frac{\sum_{i=1}^{n} y_i}{n}$$

  Note : ces formules peuvent être déduites par l'analyse classique en prenant la fonction S donnant la somme des carrés des écarts avec α_0, α_1 comme variables puis d'annuler les dérivées partielles de S par rapport à α_0, α_1.
- r^2 est le *degré d'ajustement* exprime la proportion de la variation des y_i par rapport aux x_i qui est expliquée par la droite sur la variation totale des y_i.

Régression linéaire multiple et autres modèles

- Généralisation de la régression linéaire simple au cas à plusieurs variables explicatives.
- Les coefficients de corrélation des différentes variables explicatives vis-à-vis de la variable expliquée sont calculées, puis les variables sont progressivement intégrées au modèle en ordre décroissant de la valeur des coefficients de corrélation résiduelle.
- D'autres modèles de régression existent, en particulier pour le cas des séries temporelles (variables aléatoires représentant une évolution d'une quantité dans le temps).
  - Les modèles de régression linéaires ont alors des propriétés statistiques différentes des modèles classiques car il y a *auto-corrélation* dans les variables *i.e.*, entre les valeurs successives d'une même variable.
  - Il existe aussi dans ce cas d'autres modèles, dont des modèles à retard (dépendances sur plus ou moins de valeurs précédentes).

Représentation par réseaux de neurones

- Un RN est composé d'une succession de couches de neurones interconnectés entre leurs sorties et leurs entrées.
- Chaque neurone calcule une *fonction de combinaison* de ses entrées (somme pondérée, ...) et une *fonction d'activation* sur le résultat de cette combinaison (seuillage, ...).
- Un RN calcule une fonction de l'ensemble des entrées dans l'ensemble des sorties ; c'est un modèle paramétrique (poids, seuils, etc.) trop complexe pour être explicité comme tel.
- Qualités recherchées dans un réseau de neurones :
  - *fidélité* : si $\bar{a}_i$ est la sortie calculée pour chacun des s_i connus, alors l'erreur $|a_i - \bar{a}_i|$ est la plus faible possible ;
  - *capacité de synthèse* : taille mémoire (paramètres) du RN inférieure à celle nécessaire pour mémoriser tous les points ;
  - *capacité d'interpolation/extrapolation* : $\forall s \mid s \neq s_i, \forall i$, la sortie $\bar{a}$ observée exhibe une faible erreur à la sortie réelle (inconnue) a en étant « compatible » avec les sorties attendues des sorties $\bar{a}_i$ des s_i « proches » de s.

Construction d'un réseau de neurones

- La forme du réseau de neurones est d'abord définie : nombre de neurones, nombre de couches, interconnexions entre neurones.
- L'approche souvent utilisée est l'*apprentissage supervisé* (*deep learning*) à partir de couples connus (s_i, a_i) :
  - chaque s_i est présenté au réseau qui calcule $\bar{a}_i$;
  - $\bar{a}_i$ est comparée à a_i et l'erreur constatée est rétropropagée dans le réseau par un algorithme modifiant les poids et autres paramètres pour minimiser l'erreur.
- Méthodologiquement, on sépare l'échantillon en deux groupes : l'un sert à l'apprentissage et l'autre de témoin pour valider le réseau appris en mesurant les erreurs d'interpolation observées.
- Cette forme d'apprentissage numérique *a priori* se compare à des formes d'apprentissage *symbolique* utilisées pour créer des bases de règles.

Représenter γ : approches à base de règles

- La programmation à base de règles a été popularisée dans le domaine de l'intelligence artificielle en se fondant sur les méthodes de raisonnement classique en logique utilisant des règles et le *modus ponens* : De la règle $A \implies B$, si A est vrai, on déduit B.
- La popularité des règles pour exprimer des calculs tient :
  1. de leur relative simplicité et lisibilité ;
  2. de leur aspect *déclaratif*, représentant assez directement des connaissances humaines, reléguant le calcul opérationnel à une procédure de raisonnement formalisée ;
  3. de leur *modularité* permettant d'exprimer indépendamment plusieurs règles qui, utilisées ensuite conjointement, fournissent une forme de composition des contrôles ; et,
  4. de la facilité de faire évoluer les ensembles de règles.
- Sans surprise, les approches à base de règles ont été très largement utilisées dans de nombreux domaines, dont la programmation réactive.

Un exemple : règles ECA (*Event/Condition/Action*)

- Formalisme apparu dans le domaine des bases de données actives, puis diffusé aux systèmes de communication par publication/souscription et les processus métier.
- Syntaxe générale : **on** *event* **if** *condition* **do** *action*
  - Événement : détermine le *déclenchement* de la règle.
  - Condition : *restreint* les états où la règle peut s'appliquer.
  - Action : *exécutée* dès que la règle est déclenchée et applicable.
- Exemple : retour sur l'allocation de mémoire
  - Rappel : politique par discrétisation
    - $p \geq 90 \Rightarrow$ augmenter la mémoire de 50%,
    - $p \geq 75 \Rightarrow$ augmenter la mémoire de 25%.
  - Politique exprimée en règles ECA :

    on $p \geq 90$ **if** *available*$(0.5m)$ **do** *allocate*$(0.5m)$
    on $p \geq 75$ **and** $p < 90$ **if** *available*$(0.25m)$ **do** *allocate*$(0.25m)$

Avantages et inconvénients des règles

- Avantages de l'approche à base de règles :
  - plus clair (déclaratif) que du code impératif, donc plus facile à mettre au point, modifier et maintenir ;
  - l'exécution est laissée à un moteur générique, donc pas besoin de la coder explicitement (en particulier le déclenchement) ;
  - la composition entre règles est aussi implicite, selon la sémantique du moteur d'exécution.
- La simplicité relative des règles ECA doit être mise en correspondance avec le non dit de leur sémantique superficielle :
  - besoin d'une sémantique et d'un moteur d'exécution temps réel,
  - fenêtre temporelle de validité des événements,
  - moment d'évaluation des conditions sur des signaux temporels,
  - délais entre le déclenchement et l'exécution de l'action, et
  - ordre d'évaluation lorsque plusieurs règles sont applicables.
- Néanmoins, de nombreuses applications ont été développées, avec des moteurs d'exécution variés, dont des versions réparties.

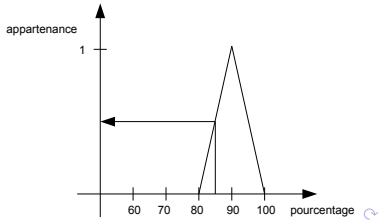
Représenter γ : logique et contrôle flou (qualitatif)

- La logique floue, inventée par Zadeh dans les années '70, vise à un raisonnement *qualitatif* mais fondé sur un calcul, le *calcul à l'aide des mots i.e.*, les valeurs qualitatives.
- Pour comprendre l'intérêt d'une approche qualitative, il est intéressant de comprendre l'effet d'une discrétisation telle que celle qu'on a imposé dans l'exemple précédent.
  - La politique exprimée par les règles ECA est une fonction discontinue en escalier dont les discontinuïtés imposent des *différences arbitrairement grandes* de décision entre des états qui sont en réalité très proches (*comparez p = 89.9 avec p = 90.1 !*).
- Quand le quantitatif ne veut plus rien dire et devient arbitraire, il vaut souvent mieux se tourner vers des solutions *qualitatives*.
- En utilisant plutôt des règles floues, on peut obtenir une fonction continue et progressive, donc un contrôle mieux adapté, sans avoir eu besoin de *recourir à un modèle explicitement continu !*

Introduction (très rapide) à la logique floue

- L'objectif de la logique floue est de capturer l'imprécision des données et d'assurer un raisonnement dans l'imprécision *bien fondé*.
- Les domaines de valeurs sont sudivisés en sous-ensembles flous représentant les « qualités », comme un pourcentage considéré comme TRÈS-FAIBLE, FAIBLE, MOYEN, ÉLEVÉ, TRÈS-ÉLEVÉ.
- C'est une extension de la logique classique, où l'appartenance à un ensemble (et donc les valeurs de vérité) n'est plus binaire (vrai ou faux) mais un degré entre 0 et 1.
- Un sous-ensemble flou $S_i \in S$ est défini par le degré d'appartenance des valeurs de S à S_i, la multi-appartenance étant admise.

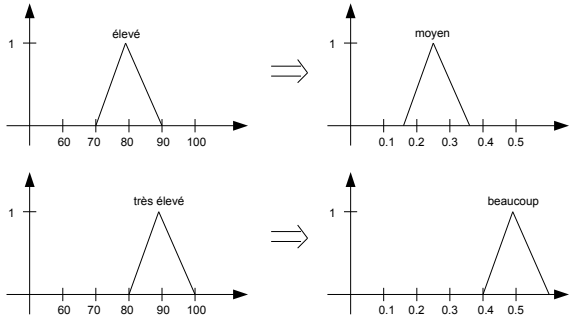
Ex. : Définissons ci-contre un sous-ensemble flou de pourcentage dit TRÈS-ÉLEVÉ.
Le degré d'appartenance de la valeur 85% est de 0.5.



Règles d'inférence

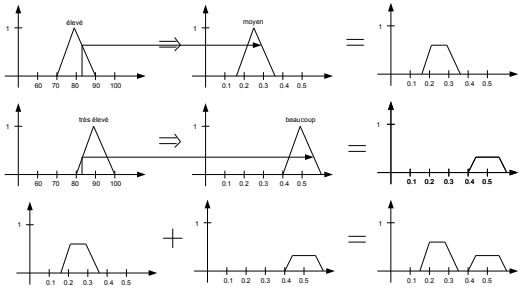
- Une régle d'inférence floue a pour prémisses et conclusion des sous-ensembles flous.

Ex. : Définissons deux régles
R1 : *si* (pourcentage) élevé *alors* (quantité de mémoire allouée) moyen
R2 : *si* très élevé *alors* beaucoup



Raisonnement flou

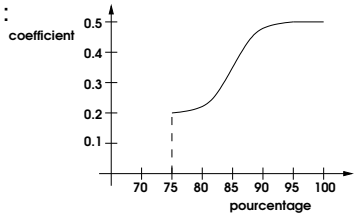
- Différentes procédures de raisonnement ont été proposées pour la logique floue, qui toutes ont pour objectif d'être compatibles avec la logique classique dans le cas *sans* imprécision, et de fournir des résultats satisfaisants dans le cas *avec* imprécision.
- Un exemple de raisonnement très visuel avec $p = 82.5\%$:



Puis, *defuzzification* par centre de gravité donne un coefficient d'environ 0.35.

Autres intérêts du contrôle flou

- Cette *discrétisation souple* produit des résultats continus et progressifs. Exemple :



- La procédure d'inférence utilisant toutes les règles applicables pour fondre en un résultat unique tous les résultats des différentes règles fournit *de facto* un *mécanisme de combinaison* pouvant être pertinent dans certains cas de contrôles multiples.
- Comme déjà mentionné, l'approche floue est beaucoup utilisée en contrôle industriel et il existe des bibliothèques complètes dans de nombreux langages, dont Java.

Plan

Introduction à la fonction de planification I

Planification

Activité consistant à produire un enchaînement d'actions valides permettant d'atteindre un but à partir d'un état initial donné.

- Deux grandes approches de planification :
  - la *recherche dans un espace d'états* manipule un *graphe des états*, où les transitions sont des actions, pour explorer différents agencements temporels d'actions (séquences, alternatives, répétitions) et choisir celui la plus apte à passer de l'état initial à un état but ;
  - la *recherche dans l'espace des plans* manipule des *plans partiels puis complets*, sélectionnant ou remplaçant progressivement les actions les plus susceptibles de compléter ou d'améliorer le plan partiel pour se rapprocher du plan final ou optimal.

Introduction à la fonction de planification II

- Quand le plan est sous-contraint, les marges de manœuvre dans les actions sont préservées en produisant les plans les plus *permissifs* possibles.
- Le planificateur comporte :
  - un *modèle déclaratif de planification* décrivant le système, son espace d'états, ses actions et leurs contraintes,
  - un *moteur de planification* raisonnant sur le modèle de planification pour produire des plans,
  - Un *moteur d'exécution* des plans ordonnançant, exécutant et supervisant la réalisation des actions.

  Idéalement, modèle et moteurs sont indépendants, mais en pratique ils sont souvent liés par des heuristiques et techniques de raisonnement rendant les planificateurs de différents modèles incompatibles les uns avec les autres.
- La supervision de l'exécution des plans peut constater l'échec d'un plan, auquel cas il est possible de répondre par *replanification* ou par *réparation* de plan puis poursuite ou ré-exécution du plan modifié.

Plans et planificateurs

- La planification est l'un des éléments-clés de l'intelligence artificielle *située*, c'est-à-dire la conception d'agents (logiciels ou physiques) capables d'agir de manière intelligente et autonome dans un environnement, comme les robots autonomes.
- Malheureusement, la complexité exponentielle de la planification en général n'a pas permis de produire un planificateur *universel*, utilisable pour *tous* les problèmes.
- La planification a donc fait l'objet de très nombreuses recherches, utilisant différents formalismes pour différentes classes de problèmes, avec des résultats théoriques et pratiques utilisables dans le domaine des systèmes auto-adaptables.
- En particulier, il existe de nombreux logiciels de planification, la difficulté principale étant de comprendre leurs formalismes puis d'exprimer les problèmes à résoudre dans ces formalismes.

Planification et auto-adaptabilité

- En auto-adaptabilité, la fonction de planification est chargée de transformer une *action* d'adaptation sélectionnée par la fonction analyse/décision et en une *opération* d'adaptation effectivement réalisable :
  - l'action d'adaptation sélectionnée indique « *déclarativement* » le nouvel état de l'entité adaptable à atteindre, alors que
  - l'opération est une description *opérationnelle*, ou une « recette », qui définit de manière précise une séquence de modifications réalisables par les actionneurs disponibles.
- Si dans les cas les plus simples, l'adaptation sélectionnée correspond à un appel d'un actionneur sur des paramètres prévus, dans les cas plus complexes, elle va demander d'orchestrer des appels à plusieurs actionneurs, en respectant des contraintes temporelles.
- *C'est cette orchestration qui peut être exprimée sous la forme d'un* ***plan****.*

Planification dynamique, replanification et réparation

- Dans un contexte réel, un plan peut être invalidé par des modifications de l'état du monde se produisant de manière incontrôlée en parallèle avec le processus de planification puis d'exécution du plan.
- La *planification dynamique*, la *replanification* et la *réparation* de plans proposent des techniques permettant de reviser un plan pendant sa construction et son exécution pour tenir compte de tels changements.
- Dans le contexte des systèmes auto-adaptables, de telles techniques deviennent essentielles dès que la durée d'exécution et la portée spatiale des adaptations sont telles qu'on ne peut plus supposer que les évolutions non contrôlées du système n'auront pas d'impact sur les adaptations en cours.

Planification décentralisée

- La nature répartie des systèmes auto-adaptables soulève le problème de la construction de *plans répartis*.
- Certes, il reste possible de définir des planificateurs centralisés produisant des plans répartis, mais cette centralisation n'est pas toujours possible, lorsque les informations d'état sont cachées dans les différentes entités par ailleurs autonomes, ni même envisageable, lorsque le nombre d'entités croit.
- Pour répondre à ces difficultés, des planificateurs décentralisés ont été développés c'est-à-dire, des planificateurs aussi indépendants que possible ne se concertant que lorsque des éléments du plan à construire nécessitent l'atteinte simultanée d'états cohérents par les différentes entités.
- *C'est la forme de planification privilégiée pour les systèmes auto-adaptables répartis.*

Plan

Exécution des adaptations

- Côté entité de contrôle, la fonction d'exécution des adaptations orchestre les appels aux actionneurs, allant de simples appels d'actionneurs sur des paramètres données jusqu'à l'utilisation d'un moteur d'exécution de plans.
- Un moteur d'exécution de plans est chargé de veiller :
  - au respect des conditions de déclenchement et à l'atteinte des effets prévus pour chaque appels des actionneurs ;
  - au respect des contraintes temporelles par le déclenchement, la synchonisation et les attentes nécessaires entre les actions élémentaires ;
  - à la détection des échecs (exceptions) en cours ou en fin d'exécution des plans pour déclencher la réparation ou la replanification.
- Moteur d'exécution et actionneurs de l'entité adaptable coopèrent en échangeant les informations et les signaux nécessaires.

Exécution de plans décentralisée

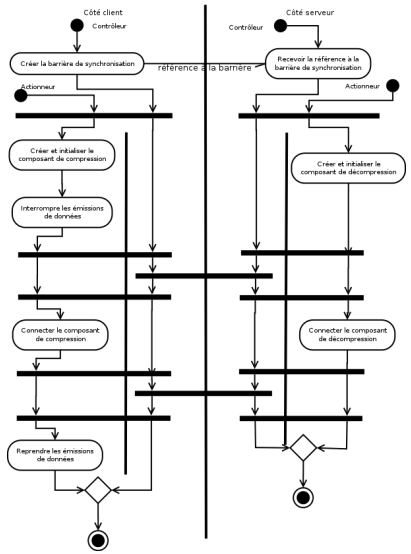
- Le pendant de la planification décentralisée est la *décentralisation de la supervision* entre plusieurs moteurs.
- La supervision décentralisée suppose l'échange de signaux et de données entre moteurs d'exécution (et/ou entre actionneurs), ce qui peut se faire par synchronisation explicite entre ces derniers au prix d'un *couplage fort* entre les deux entités.
- Une autre approche, cherchant à diminuer le couplage, consiste à utiliser des modèles de *coordination* dans cette fonction exécution des entités de contrôle, aspect qui sera abordé au cours 12.
- La *passage à l'échelle* de cette coordination présente cependant des difficultés notoires et va donc nécessiter des techniques novatrices de l'algorithmique répartie ; ce sera un des thèmes abordés au cours 14.

Exemple : retour sur l'exemple de la communication WiFi

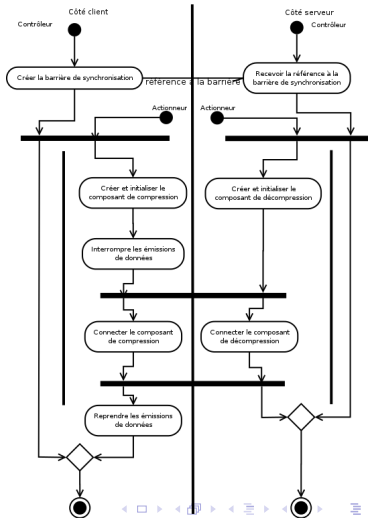
- Concernant la synchronisation des adaptations, nous avons présenté au cours 8 une solution fondée sur la coopération directe entre les actionneurs par le partage d'une même barrière de synchronisation fait au moment de leur initialisation.
- Nous présentons maintenant des solutions fondées sur la coopération des moteurs d'exécution des entités de contrôle :
  - en créant dynamiquement la barrière au niveau de l'un ou l'autre des moteurs d'exécution qui s'échangent la référence et la fournissent ensuite à leurs actionneurs respectifs ;
  - en évitant que les deux actionneurs se synchronisent directement l'un avec l'autre en les synchronisant plutôt avec leur moteur d'exécution respectif, moteurs qui se synchronisent entre eux pour assurer indirectement la synchronisation entre les actionneurs.

Choix des entités à synchroniser

Contrôleurs en synchronisation pair-à-pair



Actionneurs en synchronisation pair-à-pair



Récapitulons...

- La **fonction de collecte des données** requiert l'organisation des appels de capteurs et de la réception des données, la mise en cohérence temporelle des ensembles de données utilisées conjointement, la synthèse éventuelle de l'état du modèle de contrôle et l'insertion judicieuse des opérations de traitement et d'historisation des données.
- La **programmation des contrôleurs réactifs** peut utiliser des techniques de natures différentes, depuis la programmation impérative classique jusqu'aux techniques qualitatives de la logique floue, en passant par des langages spécialisés comme les langages synchrones.
- La **fonction de planification**, essentielle dans le cas d'**adaptations complexes et réparties**, détermine le plan d'exécution des actionneurs permettant de réaliser l'action d'adaptation choisie par la phase de décision/contrôle.
- L'ensemble des fonctions de l'entité de contrôle nécessitent une **coopération** entre entités de contrôle comme entre les entités adaptables, ce qui peut se faire grâce à des techniques de synchronisation et d'échange d'informations connues au prix d'un couplage fort entre les entités.

Pour aller plus loin : sélection de lectures recommandées

- *A Robust Layered Control System for a Mobile Robot*, R.A. Brooks, A.I. Memo 864, MIT, 1985.
- *Fuzzy Control*, K. Michels et al., Springer, 2006.
- *A QoI-aware Framework for Adaptive Monitoring*, B. Le Duc et al., ADAPTIVE 2010.
- *Distributed Implementation of Discrete Event Control Systems based on Petri Nets*, R.P. Moreno et al., 2008.