

PC3R

Cours 09 - Programmation Synchrone

Romain Demangeon

PC3R MU1IN507 - STL S2

08/04/2021

- ▶ Cours 09 - 08/04/21 : Programmation **synchrone**
- ▶ Cours 10 - 15/04/21 : **Ouverture** (présentation de PPC)
- ▶ TD 09 : **synchrone** (Esterel)
- ▶ TD 10 : **synchrone** (Lustre)
- ▶ **Pas** de TME 09/10 (projet).
- ▶ **Projet**:
 - ▶ **deadline**: Dimanche **16 Mai**,

- ▶ Programmation **standard**:
 - ▶ on **exécute** un programme avec des arguments quand on en a **besoin**.
 - ▶ le programme **calcule** puis **termine**.
 - ▶ on récupère un **résultat**.
 - ▶ on peut **relancer** le programme plus tard.
- ▶ Programmation **interactive**:
 - ▶ le programme **interagit** avec l'environnement,
 - ▶ des **événements** produisent des **réactions**,
 - ▶ le **rythme** est dicté par le programme.
- ▶ Programmation **réactive**:
 - ▶ le **rythme** est dicté par l'environnement.
 - ▶ le **temps de réaction** doit être très court.

- ▶ **Principe:** réagir régulièrement aux évènements de l'environnement.
 - ▶ **théorie:** réaction immédiate,
 - ▶ **pratique:** système interactifs à faible temps de réponse.
- ▶ **Implémentation:**
 - ▶ **automates:** simple et efficace.
 - ▶ **multi-threading:** puissant et non-déterministe.
- ▶ **Langages**
 - ▶ **impératif:** Esterel, Marvin
 - ▶ **flux:** Lustre, Lucid Sychrone
 - ▶ **outils:** Scade

- ▶ **INRIA (Sophia-Antipolis)**: dans les années 1980-1990, exploration de la programmation synchrone et développement d'*Esterel*
 - ▶ exploration de directions connexes: *fair threads*, *Argos*, ...
- ▶ **Gérard Berry**: Professeur au Collège de France à l'origine (avec son équipe) du langage *Esterel*.
 - ▶ cours CdF sur *Esterel* en 2017-2018.
- ▶ **Esterel Technologies** (2000-2018):
 - ▶ start-up commercialisant des produits basés sur *Esterel* et *Lustre*
 - ▶ maintient les outils *SCADE*
 - ▶ description et vérification de systèmes temps réels.
 - ▶ absorbée par *ANSYS* en 2018.

- ▶ **Objectif**: réaliser des **modèles** de systèmes **distribués**.
- ▶ Langage **impératif**.
 - ▶ programmation **familière** (comparé aux langages de flux)
- ▶ **Parallélisme** explicite.
- ▶ Programmation **réactive**:
 - ▶ gestion simultanée de multiples **événements**
 - ▶ le programme **ne termine pas**.

► Instants:

- le temps est découpé en **instants**.
- pendant un instant, chaque acteur du système (processus) effectue des actions **simultanément**.
- les instants apparaissent **explicitement** dans le code (principe de **coopération**)
- il y a une **synchronisation** à la fin de chaque instant (les processus s'attendent).

► Signaux:

- les signaux permettent aux processus de **communiquer**.
- un signal ne dure qu'**un instant**: il est présent ou non dans l'instant courant.
- la magie de l'**implémentation par automate** permet de gérer de manière déterministe la présence d'instant
 - si p1 émet s dans un instant et p2 attend s dans ce même instant, la "communication" aura lieu.
- les signaux peuvent être **valués** (transporter de l'information)
- tick (;) est le signal explicite de fin d'instant.
- on peut fournir au système une "**entrée**" correspondant à des signaux p ; q ; p q

Premiers mots-clefs

- ▶ `emit s`: émet le signal `s` pour l'état courant.
- ▶ `present s`: booléen qui vaut vrai ssi `s` est présent dans l'instant courant.
- ▶ `await s`: bloque le processus jusqu'à la présence de `s`
- ▶ `pause`: attends la fin de l'instant courant.
- ▶ `[||]`: sépare explicitement un processus en deux processus **parallèles**: termine quand les deux côtés ont terminé.
- ▶ `[await I1 || await I2]; emit 0`
 - ▶ attend que I1 et I2 ait été produits pour envoyer 0
 - ▶ `;I1;I2; → ;;0;`
 - ▶ `I2;;I1; → ;;0;`
 - ▶ `;;I1 I2; → ;;0;`
- ▶ `await I1 || await I2; emit 0`
 - ▶ attend que I2 ait été produit pour envoyer 0
 - ▶ `;I1;I2; → ;;0;`
 - ▶ `I2;;I1; → 0;;;`
 - ▶ `;;I1 I2; → ;;0;`

- ▶ un **module** est un programme *Esterel*:
 - ▶ une **déclaration** des signaux **entrée** et **sortie**,
 - ▶ **corps** écrit de manière impérative,
 - ▶ définition possible de **sous-modules**,
 - ▶ modules **paramétrés** (généricité).

```
module M :  
  % Interface  
  input I ;  
  output O1, O2 ;  
  
  % Corps  
  loop  
    present I then  
      emit O1  
    else  
      emit O2  
    end present ;  
    pause  
  end loop  
end module
```

- ▶ **déclaration**:
 - ▶ **utilisation** des signaux: input, output, inputoutput, (return)
 - ▶ **types** des signaux: purs, valués, ...
- ▶ **corps**:
 - ▶ **boucles** (systèmes non terminants)
 - ▶ syntaxe **explicite** (end)
- ▶ **exemple**:
 - ▶ $;;I; \rightarrow 02;02;01;02$
 - ▶ $I;;I;;I \rightarrow 01;02;01;02;01$

Exemple classique: *ABRO*

Spécification

Emit an output O as soon as two inputs A and B have occurred. Reset this behavior each time the input R occurs.

```
module ABRO:
  input A, B, R;
  output O;
  loop
    [ await A || await B ];
    emit O
  each R
end module
```

- ▶ construction loop ... each
- ▶ A;A;A B; B; A B \rightarrow ;;0;;
- ▶ A;A;A B; R B; A B \rightarrow ;;0;;0
- ▶ A;R;B \rightarrow ;;;

Expressions de signaux

- ▶ `await` et `present` testent la présence de signaux dans l'instant courant
 - ▶ `await s` débloquent **instantanément** sa continuation quand `s` est présent.
 - ▶ `await s` **finit** l'instant courant quand on l'atteint.
 - ▶ `await I; emit 0`
 - ▶ `;I → ;0`
 - ▶ `I; → ;`
 - ▶ `await immediate s` **ne finit pas** l'instant courant.
 - ▶ `await immediate I; emit 0`
 - ▶ `;I → ;0`
 - ▶ `I; → 0;`
 - ▶ c'est équivalent à :
`if present I then emit 0 else await I ; emit 0 end present`
- ▶ **expressions** booléennes `present [I1 and I2] or [not I3]`
- ▶ attente de **plusieurs** occurrences
 - ▶ `await 3 S` c'est `await S; await S; await S`
 - ▶ dans `await n S`, l'entier `n` peut être une **variable**.

```
loop
  [ present S
    then emit O
    end present
  ||
    emit S ]
; pause
end loop
```

- ▶ Emet 0 à **chaque instant**, de manière déterministe.
- ▶ Différent du comportement intuitifs de **threads**
 - ▶ pas de **sémantique d'entrelacement**
- ▶ provient de l'**implémentation** (par automates) d'*Esterel*
 - ▶ détecte les **circularités éventuelles** et rejette les programmes.
 - ▶ cf. Cours de Gérard Berry ou documents officiels.

- Programmation **impérative** classique avec des variables:

```
module M :  
  output Equal ;  
  constant C : integer ;  
  var X := 0 : integer, Y : integer in  
  Y := C ;  
  loop  
    if (X = Y)  
      then emit Equal ; X := 0 ; Y := C  
      else X := X + 1  
      end if ;  
    pause  
  end loop  
end var  
end module
```

- constantes/variables, affectation, occurrence.
- émet Equal tous les C instants.

- ▶ **déclaration** `input I : boolean ;`
- ▶ **déclaration + initialisation** `input I := true : boolean ;`
- ▶ **lecture de la dernière valeur** `?I`
- ▶ **émission valuée** `emit O (13)`

```
module Observe :  
  input I := 0 : integer ;  
  output O : integer ;  
  
  loop  
    emit O (?I) ;  
    pause  
  end loop  
end module
```

`;I(1);;I(2) → O(0);O(1);O(1);O(2)`

- ▶ **ne fonctionne pas**: `emit S (?S + 1)`
 - ▶ **principe** général: un processus ne doit pas émettre et écouter un même signal pendant un même instant.
- ▶ **fonctionne** (`pre` prend la valeur à l'instant précédent):

```
module COUNT:
  input I;
  output COUNT := 0 : integer;
  every I do
    emit COUNT(pre(?COUNT) + 1)
  end every
end module
```

Combinaison de signaux valués

► impossible:

```
module DEUXVALEURS:  
  output O : integer;  
  [  
    emit O(1)  
    ||  
    emit O(0)  
  ]  
end module
```

- pour les **signaux valués simples**, une seule émission est possible lors d'un même instant.
- combine autorise les émissions multiples.
- on doit donner une **opération de combinaison** explicite.
- possible:

```
module DEUXVALEURS:  
  output O : combine integer with +;  
  [  
    emit O(1)  
    ||  
    emit O(0)  
  ]  
end module
```


Combinaison (II)

```
► module M :  
  input I ;  
  output O : combine integer with + ;  
  var X := 0 : integer in  
  loop  
    [ emit O (X) ; X := X + 1  
      || present I then emit O (10) ] ;  
    pause  
  end loop  
end var  
end module
```

```
► ;;;I;;I → 0;1;2;13;4;5;16
```

- ▶ signaux valués légers:
 - ▶ pas d'information de présence,
 - ▶ équivalent à variable externe en lecture seule

```
module Thermometre :  
  input Calculate ;  
  output Fahrenheit : float ;  
  sensor Celsius : float ;  
  
  function c2f (float) : float ;  
  
  loop  
    emit Fahrenheit (c2f (? Celsius)) ;  
  each Calculate  
end module
```

- ▶ function: fonction externe (en C).
 - ▶ s'exécute en temps nul.

```
module M :  
  type T ;  
  procedure Increment (T) (int) ;  
  function Init () : T ;  
  function Test (T) : boolean ;  
  var X := Init () : T in  
  loop  
    if (Test (X))  
      then call Increment (X) (1)  
      end if ;  
    pause  
  end loop  
end var  
end module
```

► fonctions et procédure.

Préemption

- ▶ la **préemption** permet de **sortir** d'un bloc en présence d'un signal.

- ▶

```
abort
  loop
    emit O ;
    pause
  end loop
when A ;
emit Aborted
```

- ▶ $;;A; \rightarrow 0;0;Aborted;$

- ▶ la **préemption faible** laisse le bloc finir l'instant.

- ▶

```
weak abort
  loop
    emit O ;
    pause
  end loop
when A ;
emit Aborted
```

- ▶ $;;A; \rightarrow 0;0;0 \text{ Aborted};$

- ▶ On peut **raffiner** le comportement d'avortement:

- ▶

```
abort ... when case ... do ... (...) end abort
```

Suspension

- ▶ la **suspension** permet de **geler** un processus dans son état courant

- ▶

```
suspend
  loop
    emit O ;
    pause
  end loop
when S
```

- ▶ $;S;S; \rightarrow 0;;;0$

- ▶

```
module threadSynchrone:
  input start, stop, susp, res;
  await immediate start;
  abort
    signal dodo in [
      loop
        await susp;
        abort
          sustain dodo
        when res
      end loop
    ||
      suspend
        run PROG()
      when dodo
    ] end signal
  when stop;
end module
```

Exception

- ▶ `trap T in ... end trap` permet de définir un bloc avec un comportement exceptionnel
- ▶

```
trap T in
  loop
    present I
    then exit T
    end present ;
    emit O
  each tick
end trap ;
emit E
```
- ▶ $;;I \rightarrow 0;0;E$
- ▶ **raffiné** `en trap T1, ..., Tn in .. handle T1 do ...`
`(...) end trap`

```
module M :  
  input I ;  
  output O ;  
  ...  
end module  
  
module P :  
  input I1 , I2 ;  
  output O ;  
  
  run M1/M [ signal I1/I ]  
  ||  
  run M2/M [ signal I2/I ]  
end module
```

- ▶ **paramétrisation** par **instantiation** des signaux/variables du sous-module.

Exemple

```
module SPEED:
  input Centimeter, Second;
  relation Centimeter # Second;
  output Speed : integer;
  loop
    var Distance := 0 : integer in
      abort
        every Centimeter do
          Distance := Distance+1
        end every
      when Second do
        emit Speed(Distance)
      end abort
    end var
  end loop
end module
```

```
module REGUL:
  function Regfun (integer, integer)
    : integer;
  input Centimeter, Second;
  sensor GasPedal : integer;
  relation Centimeter # Second;
  output Regul : integer;
  signal Speed : integer in
    run SPEED
  ||
    await Speed;
    sustain Regul(Regfun(?Speed, ?GasPedal)
  end signal
end module
```

- ▶ **relations** entre signaux (#, =>) utilisées pour l'optimisation et la vérification.
- ▶ every S do ...: **similaire** à loop ... each S mais attends un premier S
- ▶ sustain: émission à **chaque** instant.

Tâches Asynchrones

- ▶ utiliser `call` sur une fonction C **bloque** le système pendant son exécution.
 - ▶ un appel de fonction est **immédiat** du point de vue d'Esterel.
- ▶ `exec TASK t return R` permet d'**exécuter** la task C et d'**attendre** sa terminaison.
- ▶ les tâches peuvent prendre **plusieurs instants** et "renvoie" une valeur en **émettant un signal**.

```
module M :  
  type Coords, Traj ;  
  input Current : Coords ;  
  output NewTrajectory : Traj ;  
  return R ;  
  task ComputeTrajectory (Traj) (Coords) ;  
  var T : Traj in  
  [ loop  
    await Current ;  
    exec ComputeTrajectory (T) (? Current)  
    return R ;  
    emit NewTrajectory (T)  
  end loop ]  
  || (...) ]  
end var  
end module
```

Tâches (II)

- ▶ Deux tâches **ne peuvent pas** avoir le **même signal** de retour.

```
▶ module OneTask :  
  task TASK (integer) (integer);  
  return R;  
  var X := 0 : integer in  
    exec TASK(X)(1) return R  
  end var  
end module  
  
module TwoTasks :  
  return R1, R2;  
  run Task1 / OneTask [signal R1 / R]  
  ||  
  run Task2 / OneTask [signal R2 / R]  
end module
```

- ▶ l'exécution nécessite un *runtime* spécifique, permettant la gestion du *synchrone*.
- ▶ compilation vers C: `str1 saucisse.str1 → saucisse.c`
- ▶ compilation depuis C: `gcc -c saucisse.c saucisse.main.c`
puis `gcc -o saucisse saucisse.o saucisse.m.o`

Bilan

Langage *synchrone* déterministe pour la *modélisation* de systèmes concurrents et temps réel.

Exemple: Réveil

```
module reveil_matin :  
  input Minute ;  
  input AlarmAt : integer ;  
  input CancelAlarm ;  
  output WakeUp ;  
  output Time : integer ;  
  [  
    var elapsed : integer in  
      elapsed := 0 ;  
    every Minute do  
      elapsed := elapsed + 1 ;  
      emit Time (elapsed) ;  
    end every  
  end var  
  ||  
  every AlarmAt do  
    abort  
    await ?AlarmAt Minute ;  
    emit WakeUp ;  
    when CancelAlarm ;  
  end every]  
end module
```

Exemple: Téléphone (I)

```
module telephone :  
  input Seconde ; input Decrocher ; input Saisie_numero ;  
  input Appel ; input Raccrocher ;  
  output Temps_communication : integer ; output Sonnerie ;  
  output Echec_appel ;  
  loop  
  var echec : boolean in  
    await Decrocher ;  
    echec := false ;  
    abort  
    await 10 Seconde ;  
    echec := true ;  
  when Saisie_numero ;  
    if not echec then  
      abort  
      var total := 0 : integer in  
        every Seconde do  
          total := total + 1 ;  
          emit Temps_communication (total)  
        end every  
      end var  
    when Raccrocher ;  
    else await Raccrocher ;  
  end if  
end var  
end loop
```

Exemple: Téléphone (II)

```
||
loop
var echec : boolean in
  await Appel ;
  echec := false ;
  abort
  abort
    every Seconde do
      emit Sonnerie ;
    end every ;
  when 20 Seconde ;
    emit Echec_appel ;
    echec := true ;
  when Decrocher ;
  if not echec then
    abort
    var total := 0 : integer in
      every Seconde do
        total := total + 1 ;
        emit Temps_communication (total) ;
      end every
    end var
  when Raccrocher ;
  end if
end var
end loop
end module
```

- ▶ Les programmes contenant des problèmes de causalité sont rejetés à la compilation.

- ▶ **rejeté:**

```
signal S1, S2 in
  emit S1;
  present S2 then
    present S1 else emit S2 end
  end
end signal
```

- ▶ **rejeté:**

```
signal S1, S2 in
  present S1 then emit S2 end;
  pause;
  present S2 then emit S1 end
end signal
```

- ▶ **rejeté:**

```
signal S in
  present S else emit S end;
end signal
```

- ▶ **rejeté:**

```
present S1 then
  emit S2
  ||
  present S2 else emit S1 end
end
```

► Principes:

- une horloge principale donne le rythme,
 - les données sont des flux: c'est à dire des valeurs produites à des ticks d'horloge
 - les programmes sont des équations permettant de créer, modifier, combiner des flux.
 - les arguments des fonctions sont des flux, les résultats sont des flux.
- Programmation de flux synchrone: le temps est logique (abstrait), à chaque instant on calcule des sorties en fonction des entrées présentes.

- ▶ **LUSTRE** : langage synchrone défini en 1985 par P. Caspi et N. Halbwachs (Vérimag, Grenoble)
 - ▶ vision **fonctionnelle**,
 - ▶ précision du contrôle par les **horloges**,
 - ▶ capacité de conserver des valeurs dans des **registres** (pre) entre deux étapes de calcul.
- ▶ **SCADE** : environnement de développement industriel développé par la société Esterel-Technologies
 - ▶ contient un noyau **Lustre** (mais pas seulement)
 - ▶ programmation **graphique** (graphes), génération de code C
 - ▶ utilisé dans le **logiciel embarqué** critique (Airbus, ...)

Exemple

```
node Module1(Y,Z : int)
returns (X : int) ;
let
  X = (Y * 2) + Z ;
tel
```

```
node Module1(Y,Z : int)
returns (X : int) ;
var S : int ;
let
  X = S + Z ;
  S = Y * 2 ;
tel
```

- ▶ les fonctions sont des **noeuds** (représentation graphique sous forme de **réseau**)
- ▶ le **corps** des fonctions est constitué d'**équations désordonnées**:
 - ▶ elles définissent les flux de sortie **en fonction des** flux d'entrée.
 - ▶ **une** équation par flux de sortie.
 - ▶ elles peuvent manipuler des flux **internes**.

- ▶ flots **constants**
 - ▶ 2 c'est le **flot** 2;2;2;2;2;...
- ▶ **opérations élémentaires** de flots
 - ▶ si X c'est X1;X2;X3;X4;...
 - ▶ si Y c'est Y1;Y2;Y3;Y4;...
 - ▶ alors $X + Y$ c'est $X1 + Y1; X2 + Y2; X3 + Y3; X4 + Y4; \dots$
- ▶ **registre** pre utilisé comme un délai
 - ▶ si X c'est X1;X2;X3;X4;...
 - ▶ alors **pre** X c'est nil;X1;X2;X3;...
- ▶ **initialisation** \rightarrow modifie la **première valeur** d'un flux
 - ▶ si X c'est X1;X2;X3;X4;...
 - ▶ si Y c'est Y1;Y2;Y3;Y4;...
 - ▶ alors $X \rightarrow Y$ c'est X1;Y2;Y3;Y4...
- ▶ l'opérateur primitif **fby** est défini par \rightarrow **pre**
 - ▶ si X c'est X1;X2;X3;X4;...
 - ▶ si Y c'est Y1;Y2;Y3;Y4;...
 - ▶ alors $X \text{ fby } Y$ c'est X1;Y1;Y2;Y3;Y4...

- **Convolution** (moyenne des deux dernières valeurs):

```
node Convolution (X: real)
returns (Y: real);
let
  Y = (X + 0 -> pre X)/2;
tel
```

- **Fronts montants** (passage de faux à vrai):

```
node Edge (X : bool)
returns (Y: bool);
let
  Y = false -> X and not pre (X);
tel
```

- **Fronts descendants** (passage de vrai à faux):

```
node Falling_Edge (X : bool)
returns (Y: bool);
let
  Y = Edge(not X);
tel
```

Exemples (II)

► Bascule:

```
node SWITCH1 (set, reset, initial: bool) returns (level: bool);
let
    level = initial -> if set then true
                       else if reset then false
                       else pre(level);
tel
```

► Compteur:

```
node COUNTER (init, incr: int; X, reset: bool) returns (N: int);
var PN: int;
let
    PN = init -> pre N;
    N = if reset then init
        else if X then PN + incr
        else PN;
tel}
```

Dépendences cycliques

- ▶ un *deadlock* se produit si deux flux sont *mutuellement dépendants*.

```
node integrator(F,STEP,init: real) returns (Y: real);
let
  Y = init -> pre(Y) + ((F + pre(F))*STEP)/2.0;
tel

node sincos(omega:real) returns (sin , cos: real);
let
  sin = omega * integrator(cos,0.1,0.0);
  cos = omega * integrator(-sin,0.1,1.0);
tel
```

- ▶ on peut (souvent) s'en sortir *en utilisant pre*:

```
node sincos(omega : real) returns (sin , cos: real);
var pcos,psin: real;
let
  pcos = 1.0 fby(cos);
  psin = 0.0 fby sin;
  sin = omega * integrator(pcos,0.1,0.0);
  cos = omega * integrator(-psin,0.1,1.0);
tel

node integrator(F,STEP,init: real) returns (Y: real);
let
  Y = init -> pre(Y) + ((F + pre(F))*STEP)/2.0;
```

- ▶ *Lustre* permet un **contrôle précis** du temps (des fréquences des signaux), grâce à la définition d'**horloges**.
- ▶ chaque noeud a comme **horloge** l'horloge de ses paramètres (par défaut, c'est l'horloge initiale des tick).
- ▶ `s1 when s2` prend un flux de valeurs `s1` et un flux booléen `s2` **sur la même horloge** et produit un flux correspondant aux valeurs de `s1` émises seulement quand `s2` est vrai.
 - ▶ si `X` c'est `X1;X2;X3;X4;X5;X6;X7;X8;...`
 - ▶ si `Y` c'est `true;false;true;false;true;true;false;false;...`
 - ▶ alors `X when Y` c'est `X1;;X3;;X5;X6;;;...`
- ▶ `current s1` **règle un flux** sur l'horloge du noeud.
 - ▶ `current(X when Y)` c'est `X1;X1;X3;X3;X5;X6;X6;...`
 - ▶ ce **n'est pas** `X`

```
node Retard(const size: int; x: bool) returns (y: bool);  
var T : bool^(size+1);  
let  
  T[0]= x;  
  T[1..size]= false^size -> pre(T[0..size-1]);  
  y = T[size];  
tel;  
  
node Main(E: bool) returns (S: bool);  
let  
  S = Retard(5, E);  
tel;
```

- ▶ \wedge définit un **tableau** (un flux de tableau, donc).
- ▶ d'autres opérations: par exemples, red fait un *fold*


```
node Recursive (const d:int; x:bool) returns(y:bool);  
let  
  y = with d=0 then x else (false -> pre(Recursive(d-1, x)));  
tel  
  
node Main (A:bool) returns (S:bool);  
let  
  S = Recursive(10, A);  
tel
```

- récursion avec cas d'arrêt explicite.

- ▶ Langages **synchrones réactifs**:
 - ▶ basés sur les notions d'**instants** et de **signaux**.
 - ▶ **combinent** des signaux/flux pour créer d'autres signaux/flux.
 - ▶ **Expressivité**: opérateurs temporels, manipulation d'horloge, ...
 - ▶ **simplicité** de la programmation.
 - ▶ **complexité** du *runtime*:
 - ▶ **rejet** de programmes **mal-formés** (circularités, ...)
- ▶ Utilisation en tant que **modèle industriel** pour les **systèmes embarqués critiques** (véhicules, ...)

- ▶ **Résumé:**
 - ▶ se souvenir de l'**existence** de langages synchrones et de leur **utilité** pour construire des modèles de **systèmes embarqués**.
 - ▶ comprendre le **principe de programmation synchrone**.
- ▶ **TD / TME:**
 - ▶ **TD 9:** Exercices d'Esterel.
 - ▶ **TD 10:** Exercices de Lustre.
 - ▶ **TMEs 9 et 10:** Dédiés au projet. (pas de robot cette année)
- ▶ **Séance prochaine:**
 - ▶ **Ouverture:** Pi en Go (préparation à PPC).