



Université de
Bourgogne
2008-2009

L3 Informatique

Jonathan ACEITUNO
Paul ALBERT

In Silico

Projet Systèmes et Réseaux I

Résumé

Ce projet a pour but de réaliser un outil facilitant l'expérimentation informatique de façon reproductible et d'en inclure les résultats éventuels dans des rapports. En plus des fonctionnalités de base qu'on pourrait attendre d'un tel programme, le système

propose :

- ▶ l'utilisation d'un fichier de paramètres pour l'appel d'un programme cible;
- ▶ la construction d'un tel fichier de paramètres de manière interactive;
- ▶ la vérification de la version du programme cible à l'expérimentation;
- ▶ la possibilité de reprendre une expérimentation sur une panne ou un arrêt;
- ▶ la possibilité d'exécuter plusieurs mesures en parallèle pendant une expérimentation;
- ▶ plusieurs interfaces utilisateur selon le système hôte;
- ▶ l'utilisation facultative d'un programme externe pour traiter les mesures.

Abstract

The goal of this project is to build a tool helping reproducible computer experimentations and to include their results into reports. Beside basic functionalities

one can expect from such a tool, the system offers :

- ▶ an option for using a parameter file when calling the target program;
- ▶ an interactive shell for building such a file;
- ▶ version matching for the target program when running an experimentation plan;
- ▶ an option making it possible for a failed or stopped experimentation plan to continue;
- ▶ parallel execution for an experimentation plan;
- ▶ different user interfaces according to the host system's potential;
- ▶ the optional use of an external program in order to compute measures

Table des matières

Introduction	5
Cahier des charges	6
Partie technique	
Fonctionnalités générales	
Objets du système	
Définition d'un plan d'expérimentation	
Exécution d'un plan d'expérimentation	
Définition d'un agrégateur	
Exécution d'un agrégateur	
Définition d'un commentaire sur un agrégat	
Publication d'un rapport	
Obtention d'informations sur les objets du système	
Interface adaptée au système hôte	
Vérification de version du programme cible	
Analyse	12
Difficultés à résoudre	
Développement du projet	
Interface changeante : lib/interface.sh	
Récupération d'informations : lib/info.sh	
Vérification de version : lib/md5.sh	
Définition d'un plan d'expérimentation : lib/define_plan.sh	
Définition d'un agrégateur : lib/define_aggregator.sh	
Fonctionnalités minimalistes : lib/set_comment.sh, lib/usage.sh, lib/error.sh	
Exécution d'un agrégateur : lib/run_aggregator.sh	
Exécution d'un plan d'expérimentation : lib/run_plan.sh	
Programme principal : experiment	

Jeux de tests

19

Conclusion

22

Introduction

Le projet concerne un système de gestion d'expérimentations informatiques couvrant l'ensemble des manoeuvres, de la prise des mesures à leur exploitation dans un rapport par le biais d'un graphique. L'objectif est donc de réaliser un tel système en utilisant un des langages de programmation vus dans l'UE Systèmes et Réseaux.

Le système réalisé est capable de proposer à l'utilisateur de définir un plan d'expérimentation dans lequel il entrera les conditions de l'expérience, le programme utilisé et les paramètres d'appel. Il peut ensuite exécuter ce plan grâce au système qui enregistrera les résultats des mesures. Le système donne la possibilité de traiter ces mesures et de définir les éléments d'intérêt avec une certaine flexibilité (traitements sur certains éléments de ces mesures), par le biais de la notion d'agrégateur. Ces derniers objets peuvent être lancés sur un recueil de mesures choisi. Cette opération résulte en un agrégat donnant toutes les indications pour générer un graphique, qui doit peut être commenté grâce au système et inclus dans un rapport. Les objets produits par le système et destinés à un traitement futur portent des informations accessibles à la demande. Afin d'éviter de multiples manipulations intermédiaires extérieures au système de la part de l'utilisateur et de garantir la possibilité d'élaborer des traitements complexes, des fonctionnalités supplémentaires ont vu le jour : un éditeur de paramètres d'appel pour un plan d'expérimentation, plusieurs interfaces adaptées au système de l'utilisateur, la possibilité de reprendre l'exécution du plan en cas d'erreur, la possibilité d'utiliser un programme d'agrégation externe qui transforme les mesures, l'utilisation d'un fichier de paramètres externe pour un plan d'expérimentation, la possibilité d'exécuter plusieurs mesures en parallèle si le système hôte le permet, et enfin un système de vérification de version pour s'assurer que le programme cible n'a pas changé entre la définition d'un plan et son exécution. Les trois premières fonctionnalités ont été techniquement difficiles à réaliser.

Remarque : Étude de l'existant

Nous n'avons pas, à ce jour, été confrontés à des programmes similaires à celui réalisé. Les outils de tests en tous genres, dont les très connus tests de performances (*benchmarks*) restent ce qui s'en rapproche le plus. Si l'on étend l'étude au domaine du vivant, nous pouvons comparer le flot d'exécution du programme au cours normal d'une expérience de chimie ou de sciences physiques : définition d'un protocole → expérience → prise de mesures → construction d'un graphique → déductions (cette dernière partie étant symbolisée par le rapport final).

Cahier des charges

La partie qui suit consiste en une description formelle des fonctionnalités du système à réaliser. Elle définit les contraintes et les objectifs pour chaque partie de la réalisation.

Partie technique

Le système doit être exécutable sur un hôte de type GNU/Linux.

Il doit présupposer que sont présents les langages de script les plus courants sur ce type d'hôte, à savoir **bash**, **sed**, **awk** ainsi que les programmes UNIX et GNU de base : **make**, **cat**...

Le programme doit pouvoir exécuté avec les paramètres suivants :

- ▶ *define plan*
- ▶ *define aggregator*
- ▶ *run plan*
- ▶ *run aggregator*
- ▶ *comment*
- ▶ *publish*
- ▶ *info*

Le système doit pouvoir utiliser **Gnuplot** et produire des fichiers compatibles avec lui pour l'affichage de graphiques.

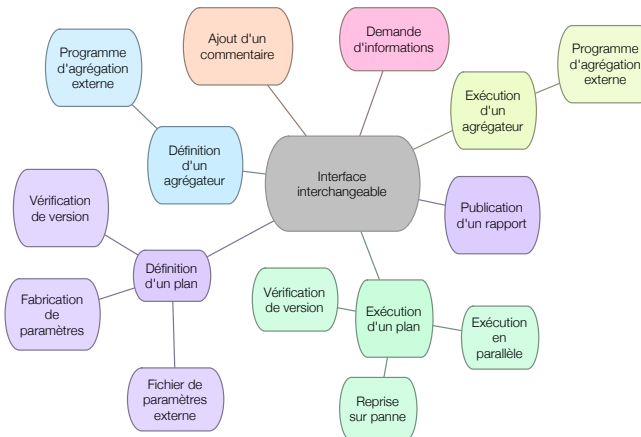
Fonctionnalités générales

Le programme doit comporter les fonctionnalités suivantes dont le détail sera donné ci-dessous :

- ▶ définition d'un plan d'expérimentation
 - fabrication interactive de paramètres
 - utilisation d'un fichier de paramètres
- ▶ exécution d'un plan d'expérimentation
 - exécution en parallèle
 - reprise sur panne
- ▶ définition d'un agrégateur
 - utilisation d'un programme externe d'agrégation
- ▶ exécution d'un agrégateur (= production d'un agrégat)
- ▶ définition d'un commentaire sur un agrégat
- ▶ publication d'un rapport à partir d'un modèle et d'un agrégat
- ▶ obtention d'informations sur un objet produit par le système

- ▶ interface adaptée au système hôte
- ▶ vérification de version du programme cible

Les fonctionnalités sont décrites par le graphe suivant :



Le système doit s'arrêter à la moindre erreur, en informant l'utilisateur sur la nature de celle-ci.

Objets du système

Le système possède les objets suivants : plan d'expérimentation, agrégateur, agrégat, recueil de mesures. Tous possèdent en commun, en plus de leur nom (limité aux caractères alphanumériques et aux caractères accentués, sans caractères d'espace), les informations suivantes :

- ▶ leur date de création
- ▶ les caractéristiques du système hôte

Le système doit être capable d'enregistrer et d'ouvrir ces objets, ainsi que de les identifier et d'en tirer les informations nécessaires.

Plan d'expérimentation

Un plan d'expérimentation est décrit par l'emplacement du programme qui sera la cible des expérimentations, des paramètres passés au programme cible ainsi qu'un facteur de répétition. Ce dernier détermine le nombre de fois qu'une même mesure doit être répétée dans l'optique d'en sortir une valeur moyenne.

Mesure

Une mesure est le résultat de l'exécution du programme cible dans un plan d'expérimentation. Elle comprend sa place dans le plan (numéro de mesure et numéro de répétition), les paramètres passés au programme cible ainsi que le résultat de l'exécution du programme.

Recueil de mesures

Un recueil de mesures est défini par un plan d'expérimentation (ce dernier étant la source des mesures) ainsi que l'ensemble des mesures tirées de ce plan.

Agrégateur

Un agrégateur définit la manière dont les mesures doivent être traitées pour parvenir à produire un graphique. Par conséquent un agrégateur doit comprendre le titre du graphique sortant, une description textuelle des abscisses et des ordonnées du graphique sortant, les parties des mesures à assigner aux abscisses et aux ordonnées du graphique sortant ainsi qu'un traitement optionnel à réaliser sur les mesures.

Agrégat

Un agrégat est le produit de l'exécution d'un agrégateur. Il doit comporter la référence vers l'agrégateur qui l'a créé ainsi que les mesures utilisées lors de l'agrégation. Il définit la structure du graphique sortant. L'agrégat peut être accompagné d'un commentaire.

Définition d'un plan d'expérimentation

Cette fonctionnalité doit permettre à l'utilisateur du système de définir un plan d'expérimentation en précisant toutes ses caractéristiques, puis de l'enregistrer dans un fichier.

Paramètres

Les paramètres passés au programme cible pouvant différer, le système doit prendre en compte des jeux de paramètres. Un jeu de paramètres est défini par une ligne de paramètres.

L'exemple suivant comprend deux jeux de paramètres :

```
toto 1 cuillère
toto 2 "jambon persillé"
```

L'appel d'un programme cible exemple avec ces jeux de paramètres est produit par l'exécution des lignes suivantes :

```
exemple toto 1 cuillère
exemple toto 2 "jambon persillé"
```

Exemple 1 : Paramètres

Fabrication interactive de paramètres

Le système doit fournir une interface permettant la fabrication d'un jeu de paramètres générique d'où seront issus les jeux de paramètres, en prenant en compte un nombre de jeux de paramètres donné à l'avance par l'utilisateur.

Les paramètres doivent être insérés un par un dans la ligne générique et il doit être possible, au moins, de les enlever un par un, à la manière d'une pile. Trois types de paramètres doivent pouvoir être insérés au moyen de cette interface :

- ▶ **paramètre fixe** : ce paramètre est recopié au même emplacement dans tous les jeux de paramètres générés. Aucun calcul n'est réalisé et il prend toujours la même valeur. Le premier paramètre de l'exemple 1 est un paramètre fixe.
- ▶ **paramètre entier à croissance monotone** : ce paramètre est défini par un numéro de début et un pas. A chaque jeu de paramètre généré, le paramètre est augmenté du pas, en partant du numéro de début. Le second paramètre de l'exemple 1 est un paramètre entier à croissance monotone avec un numéro de début de 1 et un pas de 1.
- ▶ **paramètre variable** : ce paramètre est défini par un texte différent pour chaque jeu de paramètres. Le troisième paramètre de l'exemple 1 est un paramètre variable avec pour valeurs "cuillère" pour le jeu 1 et "jambon persillé" pour le jeu 2.

La génération des jeux de paramètres doit suivre leur fabrication et le fichier de paramètres ainsi construit doit être inclus au système par la fonctionnalité suivante.

Utilisation d'un fichier de paramètres

Un fichier de paramètres fourni par l'utilisateur doit pouvoir être inclus dans le plan d'expérimentation en cours de création. Le fichier de paramètres doit être un fichier texte où chaque ligne est un jeu de paramètres différent. Le nombre de lignes du fichier donne le nombre total de jeux de paramètres. La validité de ce fichier de paramètres est à la charge de l'utilisateur.

Exécution d'un plan d'expérimentation

Cette fonctionnalité doit permettre à l'utilisateur du système d'exécuter un plan d'expérimentation préalablement défini. Le système doit exécuter le programme cible pour chaque jeu de paramètres et enregistrer les mesures résultantes dans un recueil de mesures.

Exécution en parallèle

Le système doit pouvoir déterminer s'il est possible d'exécuter plusieurs mesures en même temps et, dans ce cas, de proposer à l'utilisateur d'utiliser cette possibilité. La gestion des accès concurrentiels est à la charge du système.

L'exécution du plan d'expérimentation doit dans tous les cas être transparente pour l'utilisateur.

Reprise sur panne

Le système doit pouvoir proposer un mécanisme de reprise sur panne, qui consiste à lui proposer de reprendre ou de supprimer l'expérimentation où elle s'était arrêtée en cas de panne ou d'arrêt volontaire.

Définition d'un agrégateur

Cette fonctionnalité doit permettre à l'utilisateur du système de définir un agrégateur qui pourra être utilisé pour fabriquer un agrégat à partir d'un recueil de mesures quelconque. Toutes les caractéristiques de l'agrégateur doivent être demandées à l'utilisateur et l'objet résultant doit être enregistré dans un fichier.

Afin que le processus de production d'un graphique fonctionne, **Gnuplot** nécessite que les données qu'il reçoit contiennent au moins deux colonnes (X et Y). Cette obligation n'a pas à être vérifiée par le programme cible puisqu'elle dépend des programmes ou des valeurs que fournit l'utilisateur.

Utilisation d'un programme d'agrégation externe

Le système doit pouvoir proposer d'utiliser un programme d'agrégation externe qui aura la charge de traiter les mesures brutes, afin de permettre toutes sortes de calculs sur ces résultats. Le contenu du programme d'agrégation externe doit être inclus dans le fichier objet de l'agrégateur : c'est cette copie qui doit être appelée¹.

Exécution d'un agrégateur

Cette fonctionnalité doit permettre à l'utilisateur du système de produire un agrégat à partir d'un agrégateur donné. Le système doit demander à l'utilisateur quel recueil de mesures prendre en compte et produire un objet agrégat, éventuellement en utilisant un programme d'agrégation pour traiter les mesures. Cet objet doit être également lisible par le programme **Gnuplot** afin d'en tirer un graphique. Un fichier image au format *PNG* représentant un tel graphique doit être produit à l'exécution de l'agrégateur.

¹ Cela ne limite pas les possibilités car un traitement "en ligne" des mesures reste possible. Prenons pour exemple un traitement en ligne où chaque résultat de mesure est transformé en deux colonnes symbolisant le nombre de mots du résultat et ce nombre incrémenté, qu'on aurait réalisé avec **bash** grâce à la ligne de commande suivante :

```
cat resultats | awk '{print length($0) " " length($0)+1;}'
```

Si le fichier resultats est composé des trois lignes "coucou", "sandwich" et "saucisson", alors le résultat du traitement en ligne sera composé des trois lignes "6 7", "8 9", "9 10". Ce traitement en ligne peut aussi être réalisé dans un script **bash** d'une ligne :

```
awk '{print length($0) " " length($0)+1;}'
```

Lorsque ce programme sera "pipé", l'entrée standard sera dirigée sur le programme **awk** de la même façon que ci-dessus.

Définition d'un commentaire sur un agrégat

Cette fonctionnalité doit permettre à l'utilisateur du système d'écrire un commentaire sur un agrégat donné. Le programme doit garder ce commentaire attaché à un agrégat particulier et afficher le graphique lors de l'écriture du commentaire, si c'est possible.

Publication d'un rapport

Cette fonctionnalité doit permettre à l'utilisateur du système de publier un rapport au format *PDF*² selon un modèle donné. Le système doit demander un modèle sous forme de fichier **LaTeX** où sont incluses, s'il le souhaite, des balises `@graph@` et `@comment@`. Le système doit pouvoir remplacer les occurrences de ces balises par le graphique et le commentaire d'un agrégat qui doit être demandé à l'utilisateur.

Obtention d'informations sur les objets du système

Cette fonctionnalité doit permettre à l'utilisateur du système d'obtenir des informations sur les objets produits et utilisés par le système, à savoir un plan, un agrégateur, un agrégat ou un recueil de mesures. Ces informations doivent être adaptées au type d'objet, indiquer les deux informations communes à tous les objets (date de création et caractéristiques du système hôte) et refléter une partie de son contenu, à savoir :

- ▶ pour un **plan d'expérimentation**, l'utilisateur doit savoir son nom (donné par le nom de l'objet), le nombre de jeux de paramètres à exécuter et le facteur de répétition;
- ▶ pour un **agrégateur**, l'utilisateur doit savoir son nom, le titre du graphique sortant, la description textuelle des abscisses et des ordonnées du graphique sortant ainsi que la manière dont les résultats seront traités pour parvenir au graphique;
- ▶ pour un **recueil de mesures**, l'utilisateur doit savoir son nom, le nom du plan d'expérimentation utilisé ainsi que les conditions d'exécution du plan (notamment si l'exécution a été faite en parallèle ou non);
- ▶ pour un **agrégat**, l'utilisateur doit savoir son nom, le nom de l'agrégateur qui l'a formé ainsi que le nom du recueil de mesures utilisé pour le créer.

Interface adaptée au système hôte

Le système doit pouvoir adapter son interface au système hôte. Les utilisateurs de terminaux sans possibilité d'affichage en couleur doivent avoir une interface brute en ligne de commande. Les utilisateurs de terminaux avec cette possibilité doivent avoir une interface comprenant des composants tels que des boîtes de dialogue et des boutons. Enfin, les utilisateurs possédant un système de fenêtrage compatible **X11** doivent avoir une interface adaptée à cet environnement. Les possibilités et les fonctionnalités doivent être strictement identiques avec les trois interfaces. L'utilisateur doit avoir la possibilité de forcer le système à utiliser l'une de ces interfaces.

² Le format *HTML* aurait aussi bien pu être utilisé (à l'aide du programme **latex2html**, par exemple), la génération de plusieurs formats de fichier n'apporte rien de plus.

Vérification de version du programme cible

Cette fonctionnalité doit permettre à l'utilisateur de s'assurer que la version du programme cible est la même lors de la définition du plan d'expérimentation que lors de son exécution. Les informations données par le système pour un plan d'exécution doivent indiquer si le programme a changé ou non.

Analyse

Le système ne comporte aucune difficulté conceptuelle, ainsi cette partie contient uniquement la description de la mise en œuvre du système. Nous détaillerons les modules qui composent le système ainsi que les difficultés que nous avons rencontré.

Difficultés à résoudre

Les difficultés sont d'ordre technique et nous les résumerons sous forme de questions.

- ▶ Comment créer un format de fichier et l'utiliser pour la lecture/écriture ?
- ▶ Comment vérifier qu'un fichier est du format attendu ?
- ▶ Comment récupérer des informations d'un fichier ?
- ▶ Comment avoir trois interfaces différentes qui disent la même chose, renvoient la même chose, et qui, comme si ça ne suffisait pas, s'adaptent au système de l'utilisateur avec une nonchalance sans égal ?
- ▶ Comment vérifier qu'un programme n'a pas changé de version ? Et si ce n'est pas un programme mais une commande, comment ignorer cette vérification ?
- ▶ Comment générer un graphique ?
- ▶ Comment traiter des balises dans un fichier et les remplacer ?
- ▶ Comment assurer un verrou sur un fichier susceptible d'être utilisé par plusieurs processus en même temps ?
- ▶ Comment construire pas à pas une ligne (par exemple un jeu générique de paramètres) et proposer de revenir en arrière ?
- ▶ Comment reprendre l'exécution d'une série de commandes très répétitive là où elle s'est arrêtée ?
- ▶ Comment extraire un programme encapsulé dans un fichier qu'on sait lire ?
- ▶ Comment générer un fichier PDF à partir d'un fichier LaTeX ?
- ▶ Comment ne pas détester faire des rapports ?
- ▶ Comment savoir où se trouve le programme en cours d'exécution par rapport à l'endroit où il est exécuté ?
- ▶ Comment gérer les erreurs ?
- ▶ Comment faire pour créer un fichier lisible par deux programmes complètement différents ? C'est totalement dingue !

Les réponses à ces questions sont dans le détail des modules qui composent le système.

Développement du projet

Outils de développement utilisés

Nous avons choisi de réaliser le projet avec les outils les plus classiques de l'environnement GNU/Linux. Nous utilisons principalement le langage de script **bash** conjointement avec les outils **sed** et **awk** (de façon minimaliste, toutefois), ainsi que quelques autres programmes qui ne se trouvent pas forcément sur une installation standard mais qui correspondent toutefois aux critères du cahier des charges : **flock**, **make**, **md5sum**, **pdflatex**, **epstopdf**, **gnuplot**, **dialog** et son alter ego démoniaque **Xdialog**.

Structure des sources

Le répertoire du projet contient le script principal `experiment`, un répertoire d'exemples et un répertoire `lib` dans lequel sont stockés tous les modules. Ces modules sont des scripts qui définissent des fonctions. Chaque module est lié à une fonctionnalité particulière.

Interface changeante : `lib/interface.sh`

Nous choisissons de décrire ce module en premier car il est en rapport avec toutes les autres parties du programme : c'est le lien entre l'utilisateur et la machine. Nous avons décidé de présenter le programme de plusieurs manières afin de l'adapter à différents environnements. Ainsi, selon les souhaits de l'utilisateur ou les capacités du système, le programme s'affichera en ligne de commande brute, ou alors en utilisant la librairie `ncurses` (via le programme **dialog**) ou bien en utilisant une librairie graphique de haut niveau (**GTK+** ou **Qt**) selon les préférences du serveur graphique client (via le programme **Xdialog**).

Nous avons déterminé trois primitives d'interaction avec l'utilisateur : les messages (bloquants ou non), les *prompts* (demande d'une ligne de texte) et les choix dans une liste. Ces trois primitives sont implémentées sous les trois formes dans trois fonctions. L'utilisation de ces dernières est totalement transparente pour le programmeur.

Choix de l'interface

Si aucun choix n'est forcé, le programme déterminera l'affichage qui fonctionne le mieux sur la machine client. Le choix peut être forcé en modifiant la variable d'environnement `DIALOG`, qui est soit vide (pas d'utilisation d'un



Figure 1 : Trois interfaces pour la même commande.

programme de type **dialog**), soit initialisée au nom d'un programme qui sera appelé (soit **dialog**, soit **Xdialog**).

La détermination du bon programme, si rien n'a été décidé jusqu'alors, se fait dans la fonction `gr_init`, appelée au début du programme, par l'intermédiaire de tests. Le programme utilise la commande **which** pour déterminer l'emplacement des programmes dont il veut tester l'existence. La fonction d'initialisation du système d'interface `gr_init` crée aussi un fichier temporaire au nom aléatoire (grâce à la variable d'environnement `$RANDOM`) qui servira tout au long du programme. Ce dernier sera effacé par un appel à `gr_end` qui doit précéder toute sortie du programme, même forcée. Cette difficulté est contournée par l'utilisation de la commande **trap** qui définit le comportement du programme lors d'une interruption de son exécution (par exemple, par l'utilisation du raccourci `CTRL+C` sur un PC).

Message

Un message permet de donner des informations à l'utilisateur. Ces informations peuvent être l'avancement des opérations en cours, par exemple. Certains messages nécessitent que l'utilisateur accuse réception (en confirmant le message), d'autres servent juste à éviter de lui faire perdre patience devant un programme muet et n'ont pas besoin de sa confirmation. C'est pourquoi la fonction `message` peut avoir une option qui détermine si le message est "bloquant" ou non. Ce choix change quelque chose sauf quand le programme s'affiche en ligne de commande, car il est sensé voir le message puisqu'il est affiché sur la sortie standard (et que son terminal défile et possède un historique).

Invite de saisie (*prompt*)

Un *prompt* affiche un message à l'écran puis attend la saisie d'une ligne de la part de l'utilisateur. Si l'expression "invite de saisie" peut sembler chaleureuse, il est bon de savoir que, tout comme pour les choix dans une liste, tout refus de coopérer se solde par l'arrêt brutal du programme. La fonction `prompt` imite le comportement de la commande `read` : elle demande en premier paramètre un nom de variable dans lequel elle écrira. Pour réaliser ce comportement, on utilise la commande `eval` qui va évaluer une chaîne de caractères puis la faire exécuter à `bash`.

Choix dans une liste

La fonction `liste` agit comme la fonction précédente : elle remplit une variable dont le nom est passé en argument à la fonction. Les autres arguments sont le titre de la demande et sa description, puis des couples élément/description qui constitueront les choix possibles. On peut prendre pour exemple l'appel suivant :

```
liste variable "Titre" "Choisissez quelque chose : " "element1" "Un choix sûr"
"element2" "Un choix moins sûr"
```

La fonction récupère les couples de choix possibles dans les variables positionnelles (`$1`, `$2...`) grâce à la commande `shift` qui décale vers la gauche d'un nombre voulu. Ainsi le nombre d'éléments est donné par `$#` et on peut fabriquer un test pour vérifier qu'il n'y a pas de couple incomplet, grâce à l'opérateur modulo : `[$# % 2) -eq 0]`.

L'appel de `dialog` avec ces paramètres ne pose pas de problèmes puisque c'est la syntaxe qui est utilisée. `dialog` renvoie sur la sortie d'erreur le nom de l'élément sélectionné (par exemple, `element1`). Ce nom d'élément se retrouvera dans `$variable` à la sortie de l'exécution de notre fonction. C'est lorsque `dialog` n'est pas utilisé qu'il faut utiliser la commande `select`, incluse dans `bash`, qui permet d'afficher une liste de choix. Malheureusement, cette commande n'utilise pas de couples élément/description mais directement des descriptions, et de plus elle renvoie des numéros. La difficulté a été de permettre à la fonction de sortir de la même résultat quelle que soit l'interface. Pour cela, chaque variable positionnelle est mise tour à tour dans un tableau `elements` ou dans un tableau `descriptions` avec le même indice (qui est `$(($# / 2)`), vu que le nombre de couples élément/description multiplié par deux est pair). Ainsi il a fallu utiliser une astuce qui simule l'indirection en `bash` : soit une variable `toto` contenant 3, et une variable `reference` contenant `toto`, il faut utiliser `reference` pour afficher le contenu de la variable dont il référence le nom. Cette astuce utilise la commande `eval`. Une fois que les tableaux sont faits, `select` utilise la variable d'environnement `$REPLY` pour donner le numéro choisi. L'élément du tableau `elements` correspondant sera affecté à la variable qu'on veut.

Récupération d'informations : `lib/info.sh`

Ce module contient des fonctions qui servent à créer, à extraire et à afficher des informations sur les fichiers utilisés par le système.

Ligne d'informations

Pour symboliser les informations communes à tous les types d'objets, nous avons utilisé une "ligne d'informations". Cette dernière regroupe la date de création d'un fichier (grâce au programme `date`) et les caractéristiques du système hôte (grâce au programme `uname`). Ces deux informations sont mises bout à bout et jointes par le symbole `*` lors de la création de la ligne par la fonction `create_info_line`. Les informations qu'on peut tirer d'un fichier objet sont placées dans l'en-tête du fichier, c'est-à-dire la première ligne. Ces informations sont séparées par un caractère `:`. Il est donc important que la ligne d'informations, si elle fait partie de l'en-tête,

ne contienne pas de caractères : qui pourraient fausser les interprétations de notre programme. Le programme **tr** est utilisé pour remplacer toutes les occurrences de ce caractère par un symbole rare et sans danger. Il sera utilisé pour réaliser l'opération inverse lors de l'extraction de la ligne d'informations de l'en-tête d'un fichier par la fonction `extract_info_line`. Afin de séparer les éléments de l'en-tête, on utilise la variable d'environnement `IFS` qui détermine le caractère de séparation des données. Un appel à la commande **set** avec pour argument la ligne d'en-tête permettra alors de séparer cette ligne en fonction de `$IFS` et d'envoyer les éléments séparés dans `$1, $2...` Cette technique est utilisée un peu partout dans le programme et abolit la nécessité d'utiliser **awk**.

Vérification de type

La fonction `is_type` compare le premier élément de la première ligne (la ligne d'en-tête) à un nom de type que l'on lui donne. Les types utilisées sont nommés : *plan*, *mesures*, *agregateur* et *# agregat*. La particularité de ce dernier nom sera expliqué plus tard. La fonction de vérification de type prend en arguments le type qu'on souhaite vérifier et le nom du fichier à vérifier, puis utilise **cat** et **head** pour séparer la première ligne du reste. L'*astuce IFS* est utilisée pour avoir le premier élément qui est comparé. Le reste des éléments de l'en-tête est envoyé dans un fichier temporaire bien connu pour des examens ultérieurs.

Fonction `info_get`

Une des commandes du programme consiste à donner des informations sur un type de fichier particulier. L'en-tête est examiné, et pour chaque type de fichier des informations différentes en sont extraites. L'*astuce IFS* permet de réduire à néant toute difficulté dans cette partie.

Vérification de version : `lib/md5.sh`

Ce module implémente la fonctionnalité de vérification de version d'un fichier en utilisant le programme **md5sum** qui calcule la *somme MD5* d'un fichier donné. Si le fichier n'est pas donné, alors la vérification sera ignorée (le programme **md5sum** sera appelé sur un fichier toujours présent sur un système donné, comme `/bin/false`). La somme MD5 d'un fichier contient l'emplacement de ce fichier, de façon à ce qu'un appel à **md5sum** avec les bonnes options en lui passant une somme MD5 lui permet de dire si le fichier a changé. Deux fonctions encapsulent la fonctionnalité : `calc_md5` et `match_md5`. La présence de ce programme qui fait tout fait de cette fonctionnalité quelque chose d'assez facile)

Définition d'un plan d'expérimentation : `lib/define_plan.sh`

Ce module implémente la commande `define_plan` du programme.

Fonction `define_plan`

La fonction de définition d'un plan d'expérimentation demande à l'utilisateur les informations nécessaires (facteur de répétition, programme utilisé...), enregistre les données sur la version actuelle du programme utilisé (qui peut être une commande interne de **bash**, par exemple), puis demande comment faire les jeux de paramètres qui constitueront les différents appels au programme cible lors de l'exécution du plan. Une fois toutes ces informations rassemblées, la fonction écrit le fichier de plan en respectant le format.

Utilisation d'un fichier de paramètres externe

Le nom du fichier externe est demandé dans ce cas et, après avoir vérifié qu'il s'agit bien d'un fichier par le test `[-f "$nomfichier"]`, celui-ci est copié dans le fichier temporaire le plus utilisé au monde. Le nombre de jeux de paramètres est le nombre de lignes qu'on peut avoir en utilisant la commande `cat "$TMPFILE" | wc -l`. Cette fonctionnalité présente peu de difficultés.

Construction de paramètres

La fonction `build_params` interagit avec l'utilisateur pour lui proposer de créer une ligne de paramètres générique qui générera la totalité des jeux de paramètres qu'il veut. Clarifions la terminologie :

toto 1	"avion à poils"	paramètre
toto 2	"jambon à roulettes"	jeu de paramètres
toto 3	"tonnerre de brest"	tous les jeux de paramètres

Figure 3 : Terminologie clarifiée.

La création de ces jeux de paramètres est le résultat de l'interprétation de la ligne générique suivante : `toto varint(1,1) varpar("avion à poils", "jambon à roulettes", "tonnerre de brest")`. En réalité cette ligne ne ressemble pas tout à fait à cela, et ce n'est même pas une ligne mais un tableau.

Le programme commence par demander à l'utilisateur combien de jeux de paramètres il veut générer. Ensuite, un invite lui demande quel type de paramètre il veut rajouter (fixe, entier à croissance monotone, variable) ou s'il veut enlever le dernier élément. La gestion de ces éléments se fait simplement du moment qu'on sait comment utiliser des tableaux avec **bash**.

La deuxième partie de cette fonctionnalité consiste en la création d'un fichier de paramètres à partir de la ligne générique du dessus. Chaque élément du tableau constituant la ligne générique est lu et transformé en la valeur qu'il doit prendre, et ce pour chaque jeu de paramètres à construire.

Le fichier généré est placé dans `$TMPFILE` pour revenir au cas où l'on choisit un fichier de paramètres externe.



Définition d'un agrégateur : `lib/define_aggregator.sh`

Ce module implémente la fonctionnalité de définition d'un agrégateur via la fonction `define_aggregator`. Cette fonction ne présente pas de difficultés non rencontrées jusqu'à présent. Le programme demande un programme optionnel d'agrégation externe (qui sera copié dans le fichier agrégateur), la colonne des résultats qui servira à constituer les valeurs en abscisses (ce peut être un nombre ou une commande **gnuplot**), et la colonne pour les ordonnées. La fonction vérifie que le programme d'agrégation utilisé existe et est exécutable. Aucune difficulté n'a été recensée pour cette fonctionnalité.

Fonctionnalités minimalistes : `lib/set_comment.sh`, `lib/usage.sh`, `lib/error.sh`

Ces modules sont tellement simples qu'ils se passeraient presque de toute explication. Ils correspondent respectivement à la fonctionnalité d'ajout d'un commentaire à un agrégat (la chose la plus difficile étant de vérifier que l'objet demandé est bien un agrégat), à la fonction qui affiche l'aide du programme, et à la fonction `error` qui termine le programme en fermant tout correctement et en avertissant l'utilisateur d'une erreur.

Exécution d'un agrégateur : `lib/run_aggregator.sh`

Ce module implémente la fonctionnalité de production d'un agrégat par l'exécution d'un agrégateur via la fonction `run_aggregator`. Après avoir récupéré toutes les caractéristiques de l'agrégateur et demandé un fichier de recueil de mesures (issu d'une expérimentation), le programme extrait les mesures brutes du recueil (grâce à **awk**, en prenant seulement la quatrième colonne de chaque ligne) et les envoie sur l'entrée standard du

programme d'agrégation préalablement extrait dans un répertoire et exécuté. Le résultat de ces opérations constitue les colonnes qui seront exploitées par **gnuplot**, comme tout le reste de ce fichier. L'agrégat est lui-même un script **gnuplot**, où l'en-tête (du point de vue de notre système) est toujours la première ligne, précédée d'un caractère de commentaire **#** pour **gnuplot**. Le reste des lignes du fichier est écrit en fonction des données de l'agrégateur et constitue les commandes **gnuplot** qui préparent le graphique. Les descriptions textuelles des graphes se font par les commandes suivantes :

```
set xlabel "Texte"
set ylabel "Texte"
```

Le dessin du graphique se fait dans **gnuplot** par `plot` dans une ligne telle que :

```
plot "-" using 1:2 title "Titre du graphique" smooth unique
```

Le fait d'utiliser `"-"` comme nom de fichier fait attendre à **gnuplot** des valeurs sur l'entrée standard jusqu'à la commande **end**. L'option *title* sert à donner un titre à la courbe réalisée (et non pas au graphique lui-même), et l'option *smooth unique* permet de réaliser une moyenne des valeurs *y* pour chaque point *x*, en reliant les points entre eux. Par exemple, lorsqu'on appelle un programme **factorielle** avec un paramètre entier de 1 à 50 et un facteur de répétition de 100, et qu'on veut récupérer le temps passé à calculer cette factorielle, les mesures contiendront à chaque fois cent points ayant la même abscisse. L'option *smooth unique* calcule la valeur moyenne pour chaque abscisse et permet d'obtenir une seule courbe au final. Cette difficulté aurait pu être résolue dans le code de notre programme par une fonction qui calcule une moyenne, mais le fait de compter sur **gnuplot** pour effectuer cette opération permet une plus grande flexibilité. En effet, les colonnes que regarde **gnuplot** (précisées par l'option *using*) peuvent être soit des nombres entiers, soit des valeurs complexes compréhensibles par **gnuplot** uniquement (comme $\log(\$2)$). Avoir à calculer une moyenne sur une colonne dont le numéro n'est pas un nombre est un peu moins arrangeant.

Une fois le script **gnuplot** (qui, ne l'oublions pas, est un agrégat) généré, le programme fait une image *PNG* du graphique grâce à cette ligne :

```
echo "set terminal png; set output \"\$nom_agregat.png\" | cat - \"\$nom_agregat\"
| gnuplot"
```

Les deux lignes rajoutées par **echo** se retrouvent au tout début du script **gnuplot** que ce dernier exécute. Les deux lignes servent à dire à **gnuplot** que le résultat de la commande `plot` (un graphique) doit se retrouver dans un fichier au format *PNG*.

Exécution d'un plan d'expérimentation : lib/run_plan.sh

Ce module implémente la fonctionnalité de production d'un recueil de mesures par l'exécution d'un plan via la fonction `run_plan`. Après avoir demandé le nom du recueil de mesures, le programme crée un répertoire portant son nom et y copie le plan d'expérimentation ainsi que deux fichiers très importants :

- ▶ le script qui doit exécuter une mesure particulière et enregistrer ses résultats;
- ▶ un fichier *Makefile* qui servira pour l'exécution du programme **make**.

Les copies de ces deux fichiers sont modifiées à des endroits repérés par des balises pour refléter les spécificités du plan d'expérimentation courant. Ces modifications sont effectuées grâce au programme d'édition de flux **sed**.

Une fois **make** fini (avec un code de retour correct), le répertoire et les fichiers temporaires sont supprimés et le fichier de mesures résultant est placé au bon endroit à la place du répertoire.

Script d'exécution d'une mesure

Le script attend un descripteur de mesure du type *mesureX_Y* où *X* est le numéro du jeu de paramètres testé et *Y* est le numéro de la répétition courante. Grâce à **sed**, le script connaît ces numéros *X* et *Y* et exécute le programme cible avec les arguments qu'il faut en regardant dans le plan d'expérimentation nommé *plan* du répertoire où il se trouve (le fameux répertoire temporaire). Ensuite la ligne de résultats est écrite dans le fichier *mesures* de ce même répertoire.

Le script sait s'il faut faire un verrou sur le fichier de mesures grâce à la présence d'un fichier *parallel*. Dans le cas où ce verrou est à faire, le script utilise le programme **flock** pour le réaliser. Ce programme fonctionne à la manière d'un verrou tournant (*spinlock*) et assure un accès non simultané au fichier de mesures.

Fichier *Makefile*

La fonctionnalité de reprise sur panne a été réalisée en utilisant **make**. Il était nécessaire de faire savoir à **make** où en étaient précisément les mesures à chaque instant pour implémenter correctement la fonctionnalité de reprise sur panne. Nous résolvons le problème par la création, pour chaque mesure et donc chaque exécution du script dont on parlait à la sous-section précédente, d'un fichier portant un nom explicite : *measureX_Y*. Tous ces fichiers sont détruits à la fin de l'expérimentation et ne contiennent rien. Ils sont créés avec la commande **touch**. Nous avons trouvé cette solution acceptable.

Le programme principal doit tester la présence du répertoire temporaire et du fichier *Makefile*, entre autres, pour être sûr qu'un plan d'exécution était en cours et s'est arrêté. Il demande alors à l'utilisateur s'il veut le continuer ou supprimer l'expérimentation pour recommencer ultérieurement.

Une autre difficulté réside dans la réalisation du fichier *Makefile*. En effet, nous devons soit créer le *Makefile* brutalement en écrivant toutes les cibles grâce à un script **bash**, ce qui avait peu d'intérêt par rapport au cours, ou alors utiliser les possibilités de **make** et construire un fichier qui contenait uniquement deux variables représentant le nombre de jeux de paramètres et de répétitions de chaque jeu de paramètres, et construire des modèles de règle génériques en respectant bien les dépendances. Nous avons réussi à opter pour la deuxième solution grâce à des fonctions incluses dans **make** telles que `$(addprefix [...])` ou `$(foreach [...])`. Il en résulte un fichier assez compréhensible et petit qui réside dans `lib/makefile.template`.

Publication d'un rapport : `lib/set_publish.sh`

Ce module implémente la fonctionnalité de production d'un rapport à partir d'un modèle au format *LaTeX* via la fonction `set_publish`. Aucune difficulté autre que celles qui ont déjà été traitées n'est venu nous perturber durant l'écriture de ce module.

Programme principal : `experiment`

Ce fichier contient le programme principal. Les premières opérations sont de récupérer l'emplacement du programme principal par rapport à l'endroit où il est exécuté grâce à la commande **dirname**, puis de charger tous les modules avec la commande **source**. Les options de choix arbitraire d'une interface sont déterminées grâce à la commande **getopts**. Une fois toutes les fonctions chargées et une variable `$DIALOG` déterminée, le système d'interface est initialisé (appel de la fonction `gr_init`). Enfin, le programme agit comme un aiguillage en regardant les arguments passés au programme. En fonction de ceux-ci il utilise **shift** pour oublier ces arguments et envoie le reste aux fonctions qu'il faut (par exemple, `define_plan`). A la fin, le programme termine le système d'interface par un appel à `gr_end` et c'est fini !

Jeux de tests

Cette partie présente une démonstration de l'utilisation du programme, pas à pas, dans un terminal en l'absence du programme **dialog** et de session **X**.

Tour d'horizon

Lorsque l'on l'exécute seule ou avec des arguments non conformes à sa syntaxe, elle renvoie quelques lignes sur la syntaxe correcte à utiliser sous cette forme:

```
Les commandes suivantes sont disponibles :
experiment define plan <nom du plan d'expérimentation>
experiment define aggregator <nom de l'agrégateur>
experiment run plan <nom du plan>
experiment run aggregator <nom de l'agrégateur>
experiment comment <nom de l'agrégat>
experiment publish <modèle>
experiment info <nom de modèle/plan/agrégateur/agrégat/numéro de mesure>
```

```
Les options suivantes sont disponibles :
-n    Affichage en texte simple
-d    Affichage avec le programme dialog
-x    Affichage avec le programme Xdialog
```

La marche à suivre pour procéder à une expérimentation veut alors que l'on commence par définir un plan d'expérimentation par la commande:

```
$ experiment define plan monplan
```

La saisie des paramètres d'expérimentation est alors demandée :

```
Commande du programme : echo
Facteur de répétition : 10
```

Puis viens le moment de définir les paramètres passés au programme:

```
Choix. Paramètres passés au programme :
1) Construire les paramètres
2) Ouvrir un fichier de paramètres
```

Ici nous allons construire les paramètres (choix n°1) :

```
Nombre de jeux de paramètres : 10
```

Nombre de fois où la commande sera exécutée, ses paramètres variant (en faisant abstraction du facteur de répétition)

Vous devez maintenant construire une ligne génératrice qui permettra de générer tous les jeux de paramètres.

Paramètres d'appel. La ligne d'appel est :

Comment continuer ?

- 1) Ajouter un paramètre fixe
- 2) Ajouter un paramètre entier à croissance monotone
- 3) Ajouter un paramètre variable
- 4) Enlever le dernier paramètre
- 5) Terminé

← Pour l'instant cette ligne est vide, elle se remplit au fur et à mesure de l'ajout des paramètres

Nous pouvons par exemple ajouter un paramètre entier à croissance monotone !

Entrez le numéro de début : 5

Entrez le pas : 5

Paramètres d'appel. La ligne d'appel est :

varint×5×5

Comment continuer ?

- 1) Ajouter un paramètre fixe
- 2) Ajouter un paramètre entier à croissance monotone
- 3) Ajouter un paramètre variable
- 4) Enlever le dernier paramètre
- 5) Terminé

On a terminé, on choisit la dernière option :

Création de l'objet plan...

Opération terminée.

Notre plan d'expérimentation étant maintenant défini, on peut donc passer à la définition de l'agrégateur :

```
$ experiment define aggregator monagreg
```

On doit alors saisir ici le chemin de notre programme d'agrégation (qui doit par ailleurs être exécutable)

Les valeurs mesurées seront envoyées au programme d'agrégation sur l'entrée standard, ligne par ligne, au moyen d'un pipe du style :

```
cat RESULTATS | programme_agregation
```

Emplacement du programme d'agrégation (vide ou invalide si pas de programme) : /home/davidgroooooossambly/progagreg.sh

Les numéros demandés seront utilisés dans l'affichage du graphique sur gnuplot. La commande générée aura cette forme : plot <donnees> using <col_abcisses>:<col_ordonnees>.

Libre à vous de donner un numéro de colonne du résultat ou bien une directive compréhensible par gnuplot.

Ce code n'étant pas vérifié, vous serez responsable des erreurs qui y seraient glissées.

Numéro de la colonne du résultat pour les abscisses (à partir de 1) : 1

Numéro de la colonne du résultat pour les ordonnées (à partir de 1) : 2

Titre du graphique : Mon Titre

Texte des abscisses : abs

Texte des ordonnées : ord

Opération terminée.

← Le programme d'agrégation étant un simple programme qui recopie la 1^{ère} colonne en 2^{ème} colonne ligne par ligne

Définition de l'agrégateur terminée, on peut alors lancer le plan d'expérimentation

```
$ experiment run plan monplan
```

Nom de l'expérimentation : mesures

← On choisit le nom du fichier dans lequel seront stockées les mesures

Choix. Mode d'exécution :

- 1) Normale
- 2) Plusieurs expérimentations en même temps

On prend le choix 1. L'expérimentation s'exécute...

[...]

```
Création de mesure9_10...
* mesure9 complète.
Création de mesure10_1...
Création de mesure10_2...
Création de mesure10_3...
Création de mesure10_4...
Création de mesure10_5...
Création de mesure10_6...
Création de mesure10_7...
Création de mesure10_8...
Création de mesure10_9...
Création de mesure10_10...
* mesure10 complète.
Opération terminée.
```

Reste à utiliser l'agrégateur :

```
$ experiment run aggregator monagreg

Nom de l'agrégat : results
Recueil de mesures concerné : mesures
Création de l'agrégat en cours...
Création d'un graphique...
Opération terminée.
```

On peut alors commenter l'agrégat :

```
$ experiment comment results
Commentaire sur le graphique : Le meilleur graphique (du monde)
```

Puis générer le fichier PDF à partir d'un modèle déjà créé :

```
$ experiment publish modele.tex
Agrégat utilisé : results
Création du graphique...
[...]
```

Finalement, on peut afficher les informations sur un des fichiers précédemment créé :

```
$ experiment info monagreg
Type: agrégateur
Date de création: lundi 5 janvier 2009 à 02h05m36s
Système utilisé: Linux pullz-laptop 2.6.27-9-generic #1 SMP Thu Nov 20 21:57:00
UTC 2008 i686 GNU/Linux
```

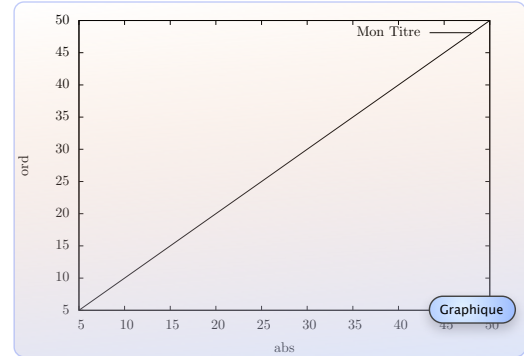
```
Titre du graphique: Mon Titre
Description des abscisses: abs
Description des ordonnées: ord
Programme d'agrégation: inclus
Fabrication des abscisses: 1
Fabrication des ordonnées: 2
```

Etude de la factorielle

Anne Onyme
January 5, 2009

1 La factorielle

Une factorielle c'est .



Mais... Le meilleur graphique (du monde) Commentaire

Figure 2 : Rapport généré et mise en évidence des éléments rajoutés

Conclusion

Notre interprétation du projet a constitué un programme qui effectue les opérations de base demandées et qui, en plus, possède quelques fonctionnalités originales, comme la vérification de version, la reprise sur panne, l'exécution en parallèle, l'utilisation de fichiers externes pour les paramètres et le programme d'agrégation. Parmi ces fonctionnalités originales, certaines ont été plus difficiles à réaliser : l'interface adaptée au système hôte, celle qui construit un fichier de paramètres interactivement, une partie de la fonctionnalité "reprise sur panne" et la syntaxe ambiguë mais finalement tellement sympathique de **bash** nous ont donné beaucoup de fil à retordre.

Comme toute réalisation à rendre en un temps donné, ce projet peut encore être beaucoup amélioré. La prise en charge des noms de fichiers avec des espaces n'est pas complète (bien qu'elle pourrait l'être en rajoutant quelques guillemets au bon endroit), l'utilisation d'un autre système de plan d'expérimentation que **make** aurait pu nous inciter à utiliser un verrou à attente active à base de **wait** et de PID (cela aurait encore plus collé avec les thématiques de l'UE). Nous pourrions aussi faire quelques parties du programme en langage C. Certaines parties auraient pu plus exploiter la puissance des descripteurs de fichiers et des sous-shells pour réaliser certaines fonctionnalités de manière plus élégante et moins compréhensible. Au niveau du rapport, nous n'avons pas respecté les consignes données sur le site puisque celui-ci décrit comme obligatoires des sections impossibles à écrire (étude de l'existant, analyse objet...) ! Si encore plus de temps nous avait été donné, nous aurions probablement écrit une bibliographie qui se serait révélée assez fournie en entrées de type `man` `commande` et nous aurions indiqué la provenance de la photographie du jambon à roulettes.