

# Manuel d'utilisation de la procédure de vérification

Rédacteur principal: Julien Troufflard

25 novembre 2015

## Table des matières

<b>1</b>	<b>Introduction</b> ( <i>Gérard Rio</i> )	<b>2</b>
<b>2</b>	<b>Arborescence du projet</b>	<b>2</b>
<b>3</b>	<b>Lancement de la batterie de tests</b>	<b>2</b>
<b>4</b>	<b>Organisation des tests</b>	<b>5</b>
4.1	Classement des tests	5
4.2	Lancement et vérification d'un test	6
4.2.1	Lancement d'un test	6
4.2.2	Vérification	6
4.3	Contenu d'un répertoire de test	7
4.4	Recherche de tests	9
<b>5</b>	<b>Ajout d'un nouveau test</b>	<b>9</b>
5.1	Vue d'ensemble	9
5.2	Méthodologie	11
5.3	Commandes usuelles CVS	11
5.4	Fichiers obligatoires	12
5.5	Fichiers facultatifs en rapport avec l'exécution du test	14
5.6	Exemples de tests	15
5.6.1	Calcul classique : comparaison de données maple	15
5.6.2	Calcul utilisant un fichier de commandes interactives	16
5.6.3	Calcul utilisant un script libre de vérification	16
5.6.4	Calcul consistant à créer un fichier .info (Herezh -n)	16
5.6.5	Exemple avancé sur l'utilisation d'un .verif pour un test multiparamètres	17
<b>6</b>	<b>Évolution des mises à jour</b>	<b>18</b>

# 1 Introduction *(Gérard Rio)*

Ce manuel présente la procédure de vérification d'une série de tests dédié à Herezh++. Cette vérification est effectuée via principalement un script perl. La première version de ce script a été mise en place par Laurent Mahéo au cours de sa thèse au début des années 2000. Puis plusieurs évolutions mineures ont été introduite. Enfin dernièrement, suite à un élargissement des objectifs, le script a été entièrement refondé par Julien Troufflard en 2015. On trouvera dans ce document une description de l'organisation de l'arborescence, du fonctionnement et enfin de la méthodologie nécessaire pour introduire un nouveau test.

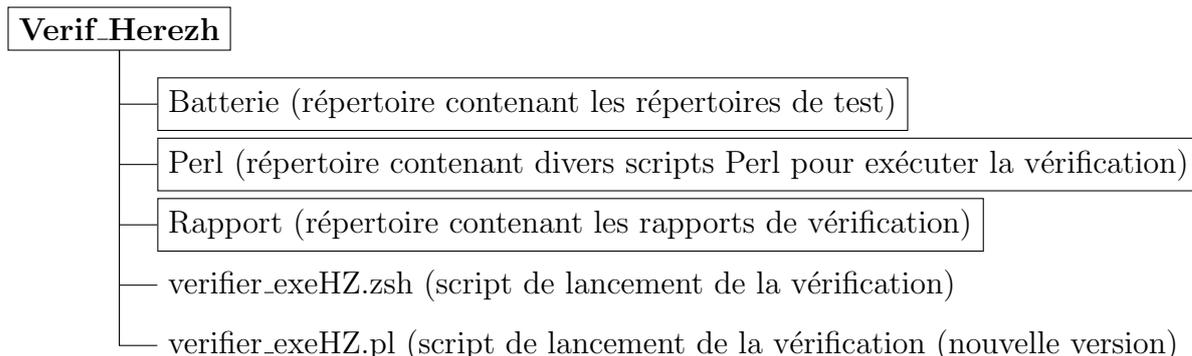
Deux objectifs principaux sont recherchés :

- la détection d'évolutions malencontreuses dans le fonctionnement d'Herezh++
- la présentation d'exemples de fonctionnement et de mise en données des différentes possibilités offertes par Herezh++.

Enfin l'idée est que rien n'est définitif, et que des évolutions sont prévisibles en fonction des demandes et attentes des utilisateurs.

## 2 Arborescence du projet

L'arborescence du projet CVS est la suivante :



## 3 Lancement de la batterie de tests

Il y a actuellement deux scripts permettant de lancer la batterie de tests. Le script `verifier_exeHZ.zsh` est la version originale historique. Le script `verifier_exeHZ.pl` est une version plus récente offrant plus de possibilités sous forme d'options. Pour l'instant, ces deux scripts existent. A terme, si `verifier_exeHZ.pl` s'avère satisfaisant, la version `.zsh` disparaîtra du projet.

Avant de détailler l'utilisation de ces scripts, voici un résumé de la procédure pour lancer la batterie :

- 1- se placer dans un répertoire contenant des tests et le répertoire Perl
- 2- exécuter le script `verifier_exeHZ.zsh` ou `verifier_exeHZ.pl [...options...]` en lui indiquant l'exécutable à tester

Le rapport obtenu après exécution de la batterie indique pour chaque test si il est réussi ainsi que le temps de calcul obtenu par la commande `time`.

### Interrompre l'exécution de la batterie (\*\*procédure temporaire\*\*) :

En l'état actuel, il est possible de stopper l'exécution de la batterie avec `ctrl+c`. Cependant, la gestion de cette interruption n'est pas encore bien gérée. Il faut faire un certain nombre de `ctrl+c` jusqu'à ce que le programme s'arrête (typiquement, il faut rester appuyer sur `ctrl+c`). Ensuite, il faut vérifier s'il y a des processus Herezh++ qui persistent en fond de tâche et tuer ces processus (`kill -9`). Ces processus sont faciles à repérer avec la commande `ps` grâce au nom de leur commande associée qui est de la forme : `testHZ[un entier] -f fichier.info`. Et enfin, éventuellement, on peut effacer les répertoires temporaires encore présents dans `/tmp` pour éviter leur accumulation (ces répertoires ont un nom de la forme : `/tmp/test.pl_[un entier]`).

**NB : cette procédure d'interruption n'altère pas les répertoires originaux de tests de la batterie. Bien que non optimale, cette procédure est sûre et n'aura aucune influence sur le prochain lancement de la batterie.**

### Script `verifier_exeHZ.zsh` :

Le script `verifier_exeHZ.zsh` lance automatiquement tous les tests présents dans le répertoire courant et ses sous-répertoires. Un test est un répertoire dont le nom commence par `Test_R` ou `Test_L` (voir section 4). Le script peut être lancé dans n'importe quel répertoire contenant le répertoire `Perl`. Il produit un rapport de tests pour les tests `Test_R` (tests rapides) et un rapport pour les tests `Test_L` (tests longs). Ces rapports sont contenus dans le répertoire `Rapport` (éventuellement créé si il n'existe pas au lancement) sous le nom `rapport_test_R.txt` et `rapport_test_L.txt`. Le script affiche automatiquement ces rapports si l'éditeur de texte `nedit` est disponible.

Le script `verifier_exeHZ.zsh` prend un argument : le nom de l'exécutable Herezh++. Typiquement, cet exécutable est présent dans un répertoire pointé par la variable environnement `$PATH`. Mais on peut également donner un chemin absolu ou relatif.

Exemples :

- cas d'un exécutable accessible via la variable environnement `$PATH` :  
`verifier_exeHZ.zsh HZppfast_Vn-1`  
`verifier_exeHZ.zsh HZppfast64`
- cas d'un exécutable spécifié par un chemin absolu ou relatif :  
`verifier_exeHZ.zsh /Users/dupont/bin/HZpp`  
`verifier_exeHZ.zsh ./HZppfast`  
`verifier_exeHZ.zsh ../mon_rep/HZppfast`

### Script `verifier_exeHZ.pl` :

Le script `verifier_exeHZ.pl` fonctionne de manière identique. Toutes les informations mentionnées pour `verifier_exeHZ.zsh` sont valables pour `verifier_exeHZ.pl`. Cette version offre des possibilités supplémentaires sous forme d'options. Ces options sont consultables en tapant `verifier_exeHZ.pl -h` dans un terminal. En l'absence d'options, `verifier_exeHZ.pl` fonctionne exactement comme `verifier_exeHZ.zsh`.

Exemples d'options :

- ne lancer que les tests rapides (option -R) :

```
verifier_exeHZ.pl -R HZppfast_Vn-1
```

- ne lancer que les tests longs (option -L) :

```
verifier_exeHZ.pl -L /Users/dupont/bin/HZpp
```

- ne lancer que les tests ayant un indicateur ECHEC dans un rapport de test (option -rpt nom\_rapport) :

```
verifier_exeHZ.pl -rpt ./Rapport/rapport_test_R.txt HZppfast (*)
```

(\*) Remarque : dans le cas de l'option -rpt, le rapport de test s'appellera  
Rapport/rapport\_test\_debugECHEC.txt

## 4 Organisation des tests

### 4.1 Classement des tests

On distingue actuellement deux sortes de tests selon le temps de calcul : les tests "Rapides" et les tests "Longs". A titre indicatif, un test "Rapide" est supposé durer moins de 30 secondes. Chaque test est contenu dans un répertoire dont le nom commence par `Test_R` (test rapide) ou par `Test_L` (test long). Les répertoires de tests sont rangés dans le répertoire `Batterie` selon l'arborescence montrée sur la figure 1. Le classement actuel est le suivant :

- `FONCTION_UNIQUE` : répertoire dédié au simple test d'une fonctionnalité (une loi de comportement, un mode de calcul, un type d'élément, ...). En général, ce sont des tests "Rapides".
- `CALCUL_COMPLET` : exemples complets issus par exemple de travaux de recherche ou applications industrielles (par exemple : une mise en forme de tôle, un essai de traction avec localisation, un déploiement de structure souple, ...). En général, ce sont des tests "Longs".
- `AUTRES` : tests difficiles à classer. Ce répertoire peut aussi servir de lieu de dépôt temporaire avant transfert vers une rubrique dédiée.

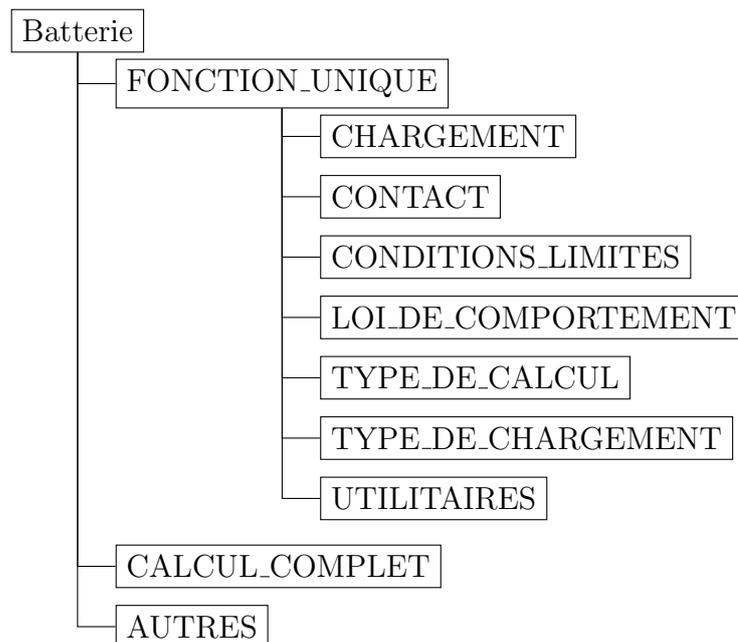


Figure 1 – Arborescence des tests dans le répertoire Batterie

## 4.2 Lancement et vérification d'un test

### 4.2.1 Lancement d'un test

Le script `verifier_exeHZ.zsh` ou `verifier_exeHZ.pl` a pour fonction de rechercher les répertoires de tests (répertoires commençant par `Test_R` ou `Test_L`). Chaque répertoire de test contient un unique fichier `.info` (vide ou non) et un certain nombre  $N$  de fichiers `.CVisu[i]` où  $[i]$  est un numéro de 1 à  $N$ . Pour chaque fichier `.CVisu[i]`, un calcul est lancé. Le premier calcul est celui correspondant au fichier `.CVisu1`. Les calculs suivants pour  $[i]>1$  sont des RESTART (si possible) du calcul précédent à partir du dernier incrément sauvegardé (selon option SAUVEGARDE du fichier `.info`).

Pour chaque calcul, un éventuel fichier d'extension `.commande[i]` contient des commandes interactives pour répondre aux différents menus interactifs proposés par Herezh++. Par défaut, un calcul Herezh est exécuté avec l'option `-f` suivie du nom du fichier `.info` (calcul classique). Si le fichier `.info` ne contient pas le mot-clé `dimension`, l'option utilisée est `-n` suivie du fichier `.info` (création interactive d'un fichier `.info`). Il est également possible de définir complètement les arguments donnés à Herezh++ dans un fichier `.argument[i]`. Pour plus d'informations sur les fichiers facultatifs, on peut se référer à la section 5.5.

L'exécution d'un test se fait dans un répertoire temporaire dans `/tmp`. On s'assure ainsi que le contenu du répertoire d'origine ne sera jamais altéré, quelque soit le déroulement du test.

### 4.2.2 Vérification

Pour chaque calcul, une vérification est faite pour déterminer si le test est réussi ou non. Le rapport de test contient les indicateurs OK ou ECHEC. Il y a actuellement 3 types de vérification possible :

- comparaison des données sorties au format maple :

il s'agit du mode de vérification par défaut. Le fichier `.CVisu[i]` définit les sorties au format maple. Le répertoire de test contient alors un fichier `.maple.ref[i]` qui définit les valeurs de référence. Si l'écart absolu et relatif entre les données de référence et les données du calcul sont inférieures à une tolérance, alors le test est réussi. Les tolérances par défaut sont en absolu  $1e-6$  et en relatif  $1e-3$ . Il est possible de choisir librement les tolérances dans un fichier `.precision[i]`. Pour une grandeur  $X$  générée à l'issue du calcul  $[i]$  et sa valeur de référence  $X_{ref}$  contenue dans le fichier `.maple.ref[i]`, le test est réussi si les relations de comparaison suivantes sont respectées :

$$\begin{aligned} \text{comparaison absolue : } & |X - X_{ref}| \leq \varepsilon_{absolu} \\ \text{comparaison relative : } & \left| \frac{X - X_{ref}}{X_{ref}} \right| \leq \varepsilon_{relatif} \quad \forall X_{ref} \neq 0 \end{aligned} \quad (1)$$

**IMPORTANT :** si les fichiers `.maple` et `.maple.ref` contiennent plusieurs lignes de données, seule la dernière ligne est traitée

- comparaison libre définie par un script :

la vérification peut être librement définie par un script qui a pour fonction de renvoyer OK ou ECHEC. Ce script est un fichier exécutable d'extension `.verif[i]`, signifiant que le calcul n°[i] sera vérifié par un script libre.

- vérification simple de l'exécution :

en l'absence de fichier `.maple.ref` ou de script `.verif`, la vérification est considérée OK si le calcul s'est lancé normalement et terminé normalement (que le calcul ait convergé ou non).

### 4.3 Contenu d'un répertoire de test

Un répertoire de tests contient des fichiers obligatoires et des fichiers facultatifs.

Les fichiers obligatoires sont :

- un fichier `README` (informations sur le test, voir trame figure 2)
- un **unique** fichier `.info` (éventuellement vide dans certains cas)
- les fichiers `.CVisu[i]` (éventuellement vides dans certains cas)
- tous les fichiers nécessaires à l'exécution des calculs (maillage `.her`, etc...)

Les fichiers facultatifs en lien avec l'exécution d'un test sont :

- les fichiers maple de référence (fichiers d'extension `.maple.ref[i]`)
- les fichiers de précision (fichiers d'extension `.precision[i]`)
- les fichiers de commandes interactives (fichiers d'extension `.commande[i]`)
- les scripts de vérification libre (fichiers exécutables d'extension `.verif[i]`)
- les fichiers donnant les arguments à l'exécutable Herezh++ (fichiers d'extension `.argument[i]`)

D'autres fichiers facultatifs peuvent être contenus dans un répertoire de test. Il n'y a pas de restriction mis à part qu'un répertoire de test ne doit pas contenir des fichiers trop lourds en terme d'espace disque. Typiquement, il peut être utile d'ajouter les fichiers suivants :

- des fichiers de mise en données du test pour d'autres codes de calcul (par exemple : fichier `.inp` pour Abaqus)
- des fichiers donnant des résultats donnés par d'autres codes de calcul ou par des solutions analytiques (bien que le fichier `README` puisse être utilisé pour écrire ces données)

---

Auteur

---

liste des auteurs sous la forme : Prenom Nom (email)  
remarque : un auteur par ligne

---

Mots-cles

---

liste des mots-cles (un par ligne)  
remarque : un mot-cle peut contenir des espaces  
(exemple : contraintes planes)

---

But du test

---

du texte...

---

Description du calcul

---

du texte...

---

Grandeurs de comparaison

---

du texte...

---

Informations sur les fichiers facultatifs

---

du texte...

---

Comparaison avec des solutions analytiques

---

du texte...

---

Comparaison avec des codes de calcul

---

du texte...

---

Figure 2 – Trame du fichier README présent dans chaque répertoire de test. Les rubriques en bleu sont obligatoires. Les rubriques en marron sont importantes pour la traçabilité des tests mais non obligatoires.

## 4.4 Recherche de tests

Un document pdf de nom `catalogue_tests.pdf` est présent à la racine du projet CVS. Il dresse la liste actuelle des tests et résume en quelques phrases leur contenu. Un index en fin de document permet de faire une recherche par mot-clé.

Ce document est rédigé automatiquement sur la base du contenu des rubriques "Auteur", "Mots-cles", "But du test" et "Description du calcul" des fichiers README (voir figure 2).

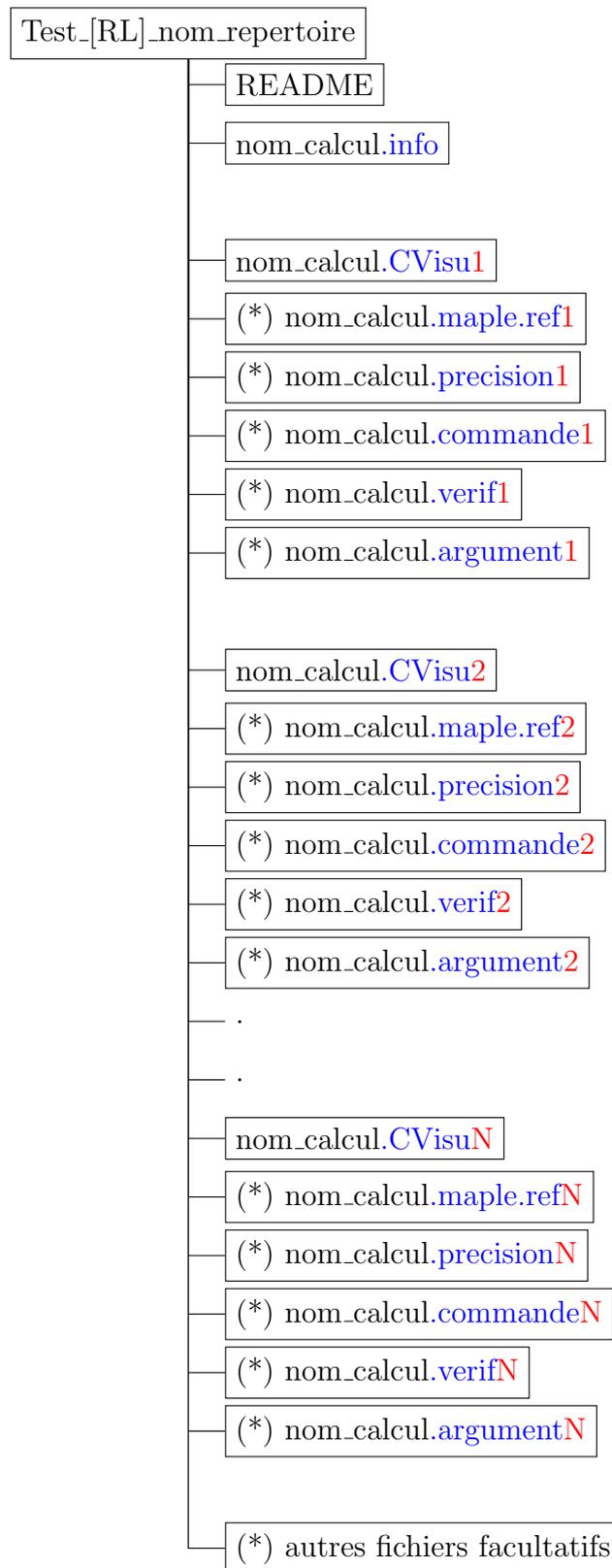
## 5 Ajout d'un nouveau test

### 5.1 Vue d'ensemble

Créer un nouveau test consiste à ajouter un nouveau répertoire dans le répertoire `Batterie`. Le choix de son emplacement dans l'arborescence est libre en s'inspirant tout de même des descriptifs donnés en section 4.1. Le contenu du répertoire est celui défini en section 4.3. Les fichiers facultatifs dépendent du type de traitement Herezh++. Dans tous les cas, l'arborescence du répertoire de test est de la forme montrée sur la figure 3 et les règles suivantes sont à respecter :

- Le nom du nouveau répertoire est choisi en utilisant uniquement les lettres de a à z (sans accent), les chiffres de 0 à 9, le signe moins "-" et le tiret bas "\_". Si le test est un test "Rapide" (moins de 30 secondes), le nom commence par `Test_R`, sinon il commence par `Test_L`. Pour savoir si un nom de répertoire existe déjà, il y a un script dédié présent dans le répertoire `Perl` : `verif_existence_repertoire_test.pl` (option `-h` pour l'aide). Pour l'utiliser, il faut se placer à la racine du projet (c'est-à-dire au même niveau que le répertoire `Batterie`) et taper `Perl/verif_existence_repertoire_test.pl nom_repertoire`.
- Un seul fichier d'extension `.info` doit être présent dans le répertoire. Ce fichier doit être présent même s'il est vide.
- Pour chaque calcul, il est nécessaire de créer un fichier `.CVisu` (`.CVisu1`, `.CVisu2`, etc...) même si celui-ci est vide. Ces fichiers servent non seulement à Herezh++ mais également au script `Perl/test.pl` pour repérer les calculs à lancer (un calcul par fichier `.CVisu`).
- tous les fichiers `.CVisu[i]` et les fichiers facultatifs nécessaires à l'exécution d'un test (`.verif[i]`, `.commande[i]`, etc...) ont la même racine que le fichier `.info` (par exemple, si on a le fichier `nom_fichier.info`, l'éventuel fichier `.verif1` doit avoir pour nom `nom_fichier.verif1`)
- le fichier `README` contient obligatoirement les rubriques "But du test" et "Description du calcul" montrées sur la figure 2. Ces deux rubriques serviront à générer automatiquement une documentation pdf (catalogue de tests). Dans le même but, les rubriques "Auteur" et "Mots-cles", qui ne sont pas obligatoires, donnent des informations utiles (merci de les renseigner).

**Remarque importante :** ne pas écrire des lignes de 3 tirets ou plus (`---` ou plus) dans le corps d'une rubrique. Ces lignes servent à repérer les titres de rubriques.



(\*) fichiers facultatifs

Figure 3 – Contenu d'un répertoire de test

La section 5.2 propose une méthodologie de travail pour construire un nouveau test. La section 5.3 est un rappel des commandes CVS de base. Les sections 5.4 et 5.5 donnent des informations sur les fichiers obligatoires et les fichiers facultatifs en lien avec l'exécution du test. Notamment, certains fichiers ont un format bien précis pour être exploitables. Ensuite, la section 5.6 a pour but de balayer les cas les plus courants de test et de donner des exemples.

## 5.2 Méthodologie

Avant l'ajout définitif du répertoire de test dans la batterie, il est plus que conseillé de construire pas à pas le test dans un répertoire en dehors du projet CVS. Dans ce répertoire, on pourra tester son fonctionnement sans risquer d'altérer le reste du projet. La méthodologie préconisée est :

- 1) créer un répertoire quelconque (appelons ce répertoire : `rep_tmp`)
- 2) copier le répertoire `Perl` et le script `verifier_exeHZ.pl` (ou `verifier_exeHZ.zsh`) dans `rep_tmp`
- 3) choisir un nom de répertoire de test (voir section 5.1) et le créer dans `rep_tmp` (appelons ce répertoire : `Test_R_nom_test`)  
(au besoin, pour choisir le nom du test :  
utiliser le script `verif_existence_repertoire_test.pl` au niveau du répertoire `Verif_Herezh/Batterie`)
- 4) créer tous les fichiers nécessaires au calcul dans `Test_R_nom_test` (voir sections 5.4 et 5.5)
- 5) se placer dans `rep_tmp` et exécuter le script `verifier_exeHZ.pl(.zsh)`  
(si le test fonctionne, un rapport de test en bonne et due forme est produit)
- 6) si il y a un dysfonctionnement : modifier les fichiers de calcul jusqu'à ce que le test fonctionne
- 7) créer le fichier `README` dans `Test_R_nom_test` (voir section 5.4)
- 8) choisir un répertoire de destination dans la batterie (voir section 4.1)  
(appelons ce répertoire `rep_dest`)
- 9) enregistrer le test dans le projet CVS (voir section 5.3) :

Remarque : Ne pas hésiter à s'inspirer des répertoires déjà existants dans la batterie

- > Recopie du répertoire de test vers le projet :  
`cp -R rep_tmp/Test_R_nom_test Verif_Herezh/Batterie/.../rep_dest/.`
- > Ajout du répertoire sous CVS :  
`cvs add Verif_Herezh/Batterie/.../rep_dest/Test_R_nom_test`
- > Ajout de son contenu :  
`cvs add Verif_Herezh/Batterie/.../rep_dest/Test_R_nom_test/*`
- > Actualisation définitive du projet CVS en se plaçant dans le répertoire du projet :  
`cvs commit -m 'ajout nouveau test etc...'`

## 5.3 Commandes usuelles CVS

- importer le projet : `cvs co -P Verif_Herezh (*)`

- actualiser sa version locale du projet : `cvs update -dP (**)`
- ajouter un fichier ou un répertoire :
  - fichier texte : `cvs add nom_fichier`
  - fichier binaire : `cvs add -kb nom_fichier`
- effacer un fichier (\*\*\*) :
  - 1) effacer le fichier `nom_fichier`
  - 2) `cvs remove nom_fichier`

(\*) l'option `-P` est très recommandée car elle permet de supprimer automatiquement les répertoires vides dès l'import du projet (voir également remarque (\*\*\*)).

(\*\*) Toujours actualiser sa version locale avant de modifier le projet. Cette commande actualise le répertoire courant ainsi que tous ses sous-répertoires

(\*\*\*) a priori, c'est la même démarche pour effacer un répertoire, bien qu'il soit souvent constaté que le répertoire effacé persistait même si il est vide.

## 5.4 Fichiers obligatoires

### • README :

1) La trame du fichier est montrée sur la figure 2. La syntaxe de l'intitulé des deux rubriques obligatoires "But du test" et "Description du calcul" doit être strictement respectée. Il est fortement encouragé de renseigner les rubriques "Auteur" et "Mots-cles". Les autres rubriques sont indiquées à titre de proposition et pour des questions d'harmonie sur la forme d'un test à l'autre. Comme montré sur la figure 2, chaque titre de rubrique est précédé et suivi d'une ligne d'au moins trois tirets (--- et plus). Les lignes de tirets doivent être réservées aux titres des rubriques (Ce motif sert à repérer la fin d'une rubrique. L'insertion d'une ligne de tirets dans le corps d'une rubrique mettra en défaut le script de génération automatique du catalogue de tests!!).

2) Dans les rubriques obligatoires, il est possible d'insérer des figures grâce à l'utilisation des balises `\figures:`, `\legende:` et `\fin_legende`. Ces figures apparaîtront dans le catalogue de tests. Une figure est constituée de un ou plusieurs fichiers image et d'une légende. Pour déclarer une figure, la ligne doit commencer par la balise "`\figures:`" suivie du noms des fichiers image (il peut y en avoir plusieurs). Ensuite, sur la même ligne, on doit trouver la balise de début de légende "`\legende:`". La légende est constituée de tout le texte compris entre les balises "`\legende:`" et "`\fin_legende`". A noter que le texte de cette légende peut être écrit sur plusieurs lignes mais ces retours à la ligne ne seront pas pris en compte dans la mise en forme du document pdf. Les formats d'images sont ceux supportés par `pdflatex` et `\includegraphics`, c'est-à-dire typiquement : `.pdf`, `.png` et `.jpg`. Les fichiers image doivent être situés dans le répertoire du test (c'est-à-dire le même répertoire que le fichier README). Concernant les dimensions des images, il faut savoir que le corps du texte du catalogue pdf a une largeur de 17cm et une hauteur de 24cm. Il est donc nécessaire de soit redimensionner les figures en fonction de ces longueurs, soit indiquer un facteur d'échelle entre crochets à la fin du nom de fichier (pas d'espace entre la fin du nom de fichier et les

crochets : `nom_fichier.pdf[0.5]`). On peut également insérer un espace entre 2 figures, tout simplement en mettant une valeur en cm entre crochets et pas de nom de figure (voir exemples ci-dessous).

Exemples :

> Exemple sur une ligne :

```
\figures: maillage.pdf \legende: Aperçu du maillage \fin_legende
```

> Exemple sur une ligne avec facteur d'échelle de 60% :

```
\figures: maillage.pdf[0.6] \legende: Aperçu du maillage \fin_legende
```

> Exemple sur une ligne avec 2 figures échelle 40% et un espace de 1.2cm entre elles :

```
\figures: fig_1.pdf[0.4] [1.2] fig_2.pdf[0.4] \legende: ... \fin_legende
```

> Exemple multilignes :

```
\figures: coupe_SIG11.pdf \legende: Vue en coupe de  
la répartition des contraintes x \fin_legende
```

> Exemple multilignes avec plusieurs fichiers image :

```
\figures: vue_dessus.png vue_cote.png \legende: Vue de dessus  
(à gauche) et vue de côté (à droite) \fin_legende
```

3) Dans les rubriques obligatoires, il est possible d'insérer des formules mathématiques Latex entre `$` ou `$$`.

Exemples :

> caractère mathématique dans le texte :

- dans le `README` :

le coefficient de Poisson est  $\nu=0.3$ , ce qui est classique pour un acier

- résultat dans le catalogue :

le coefficient de Poisson est  $\nu = 0.3$ , ce qui est classique pour un acier

> formule mathématique écrite après un passage à la ligne :

- dans le `README` :

la loi de Hooke 1D s'écrit :  $\sigma = E\varepsilon$

- résultat dans le catalogue :

la loi de Hooke 1D s'écrit :

$$\sigma = E\varepsilon$$

#### ● `.info` :

Ce fichier est l'unique fichier d'extension `.info` présent dans le répertoire. Il est obligatoire mais peut être vide (par exemple dans le cas d'une création de fichier `.info` avec l'option Herezh `-n`).

Il n'y a pas de restriction majeure sur le contenu de ce fichier mais il est important de noter que dans le cas d'un test avec plusieurs fichiers `.CVisu`, chaque calcul ultérieur au n°1 tentera de faire un `RESTART` du calcul précédent. C'est pourquoi il faut faire attention au paramètre `controle` → `SAUVEGARDE`. Le paramètre suffisant est :

```
controle  
SAUVEGARDE DERNIER_CALCUL
```

Si le calcul utilise un menu interactif, un fichier `.commande` sera nécessaire.

- `.CVisu[i]` :

La présence de ces fichiers est obligatoire mais ils peuvent être vides à partir du moment où le but du test n'est pas de comparer des résultats Herezh maple ou Gmsh

## 5.5 Fichiers facultatifs en rapport avec l'exécution du test

- `.precision` :

Ce fichier permet de modifier les précisions par défaut pour la comparaison des données au format maple pour un certain nombre de colonnes du fichier `.maple`. Par défaut, les précisions sont  $1e-6$  en absolu,  $1e-3$  en relatif. Le format du fichier est le suivant :

```
#eventuellement des commentaires...
[no colonne]  prec_absolue  prec_relative
.
.
.
#eventuellement des commentaires...
[no colonne]  prec_absolue  prec_relative
```

Il est important de noter que les crochets autour d'un numéro de colonne sont obligatoires. Par exemple, si le fichier `.maple` contient une ligne de la forme :

```
temps X Y Z SIG11
```

Il est possible de modifier les précisions uniquement pour la colonne 5 avec le fichier `.precision` suivant :

```
#precisions pour la contrainte SIG11
#          absolue  relative
[5]       1.e-2    5.e-2
```

- `.commande` :

Le but de ce fichier est de fournir automatiquement les réponses à un menu interactif. Sa présence équivaut à lancer un calcul Herezh++ de la manière suivante :

```
HZppfast [-f|-n] nom_calcul.info < nom_calcul.commande
```

Chaque ligne contient une réponse suivie par un retour à la ligne. En particulier, il ne faut pas oublier le retour à la ligne à la fin de la dernière ligne.

- `.verif` :

Ce fichier est un exécutable dont la fonction est de fournir le résultat de la vérification. Il doit effectuer les traitements nécessaires pour déterminer si le test est OK ou ECHEC. Le contenu du script est totalement libre mais il doit obligatoirement indiquer le résultat de la vérification en affichant la ligne suivante :

```
resultat verification : OK (ou ECHEC)
```

En plus de cette ligne, cet exécutable peut tout à fait afficher d'autres informations à titre facultatif.

Il est important de noter que cet exécutable reçoit 2 arguments lorsqu'il est appelé. Le programmeur pourra donc utiliser s'il le souhaite les arguments suivants :

- 
- argument 1 : nom de l'exécutable Herezh++ (en chemin absolu)
  - argument 2 : nom du fichier de redirection de l'affichage du calcul  
Herezh++ (fichier `.log`)
- 

Cet exécutable peut être programmé dans n'importe quel langage, mais pour éviter les problèmes de compilateur, il est préférable d'éviter les langages nécessitant une compilation (par exemple : C, C++, Fortran, etc...). Il est préférables d'utiliser les langages interprétés (par exemple : sh, zsh, Perl, Python, etc...). Pour augmenter la portabilité de ces scripts, il faut éviter de mettre un chemin absolu vers l'interpréteur dans l'en-tête. On utilisera la forme : `#!/usr/bin/env` suivi du nom de l'interpréteur (exemples : `#!/usr/bin/env perl` ou bien `#!/usr/bin/env python`).

Remarque : le script `.verif` étant lancé après le calcul, il peut tout à fait servir de moyen détourné pour modifier le fichier `.info` avant le calcul suivant (par exemple, pour tester un même mot-clé mais avec diverses valeurs de paramètres). Voir exemple section 5.6.5.

- **.argument :**

Ce fichier permet de définir librement les arguments donnés à l'exécutable Herezh++. Il peut contenir autant de lignes que nécessaire. Il est important de noter que l'affichage produit par Herezh++ est toujours redirigé dans un fichier `.log` avec la commande `tee`. Il est donc inutile voire bloquant de rajouter une redirection supplémentaire parmi les arguments. Au final, pour un fichier contenant N lignes, la commande Herezh++ lancée sera :

```
HZppfast ligne_1 ligne_2 ... ligne_N | tee nom_calcul.log
```

- **.maple.ref :**

Ce fichier est utile pour la comparaison de données au format maple. Le nombre de colonnes de ce fichier doit être en accord avec le fichier `.maple` produit par la lecture du `.CVisu` associé. Il peut contenir un nombre quelconque de lignes de données mais seule la dernière sera utilisée pour la comparaison de `.maple` explicitée à la section 4.2.2.

## 5.6 Exemples de tests

### 5.6.1 Calcul classique : comparaison de données maple

Il s'agit du cas le plus courant consistant à réaliser des calculs générant des données au format maple à comparer avec les données contenues dans un fichier `.maple.ref`. Les fichiers `.CVisu[i]` définissent des sorties au format maple. Le calcul  $n^{\circ}[i]$  utilise le fichier `.CVisu[i]` pour générer un fichier `.maple` qui sera comparé au fichier `.maple.ref[i]` qui doit contenir

exactement le même nombre de colonnes que le fichier `.maple`.

Le fichier `.info` doit générer automatiquement les sorties maple en choisissant parmi les 2 possibilités suivantes :

- 1- lecture automatique : `TYPE_DE_CALCUL`→avec plus lectureCommandesVisu
- 2- sortie au fil du calcul : `para_affichage`→`FREQUENCE_SORTIE_FIL_DU_CALCUL`

A noter que quelque soit la fréquence de sortie des résultats, seule la dernière ligne de données du fichier `.maple` est vérifiée.

Le test `Batterie/FONCTION_UNIQUE/TYPE_DE_CALCUL/Test_R_non_dynamique` est un exemple de calcul classique utilisant les précisions par défaut pour la comparaison des données.

### 5.6.2 Calcul utilisant un fichier de commandes interactives

Le test `Batterie/FONCTION_UNIQUE/UTILITAIRES/Test_R_fusion-maillage` est un exemple de calcul utilisant un fichier de commandes interactives (réponses au menu interactif de l'utilitaire "avec plus fusion\_maillages"). Ce fichier spécial est le fichier d'extension `.commande1`.

### 5.6.3 Calcul utilisant un script libre de vérification

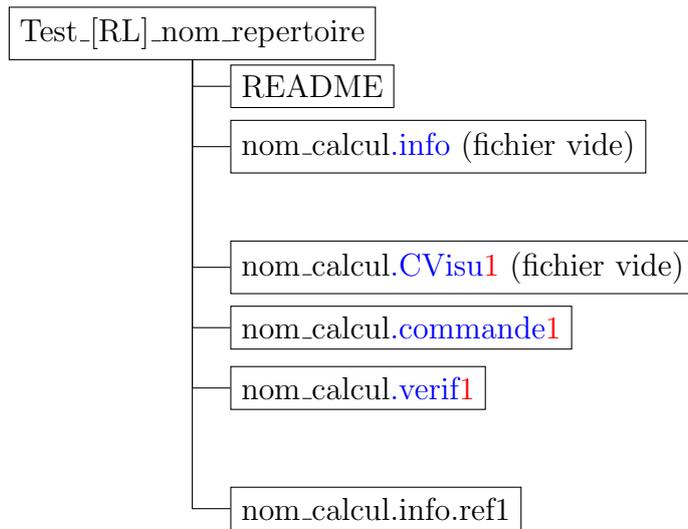
Le test `Batterie/FONCTION_UNIQUE/UTILITAIRES/Test_R_suppression-noeuds-non-references` est un exemple de calcul utilisant un script de vérification (comparaison entre deux fichiers `.her`). Ce fichier spécial est le fichier d'extension `.verif1`.

### 5.6.4 Calcul consistant à créer un fichier `.info` (Herezh -n)

Le script `Perl/test.pl` lance un calcul `Herezh++` avec l'option `-n` si le fichier `.info` ne contient pas le mot-clé `dimension`. Pour cela, on crée simplement un fichier `.info` vide. Un fichier `.commande` sera forcément nécessaire pour répondre au menu interactif de création.

Il ne s'agit pas d'un calcul classique. Donc, sauf cas particulier, la vérification sera faite via un script `.verif` afin de comparer le fichier `.info` créé à un fichier `.info` de référence. Attention au fait que le fichier `.info` de référence ne doit pas avoir l'extension `.info` car il ne peut y avoir qu'un seul fichier `.info` dans le répertoire. On peut par exemple utiliser l'extension `.info.ref1`.

Actuellement, il n'y a pas d'exemple dans le répertoire `Batterie` mais l'arborescence pour ce genre de test devrait avoir la forme suivante :



### 5.6.5 Exemple avancé sur l'utilisation d'un .verif pour un test multiparamètres

\*\*\*\* en cours d'écriture \*\*\*

## 6 Évolution des mises à jour

- 2015-06-29 (Julien Troufflard) :
  - création d'un nouveau script (`Perl/genere_catalogue_tests.pl`). Il génère automatiquement une documentation sur les tests (résumé et index pour recherche par mots-clés). Ce document est placé à la racine du projet sous le nom `documentation_tests.pdf`. Le mode de fonctionnement (lecture des fichiers README de chaque test) impose une nouvelle contrainte sur la forme des fichiers README (interdiction d'utiliser des lignes de 3 tirets ou plus dans le corps d'une rubrique README car ce motif signifie la fin d'une rubrique). Cette contrainte est le choix actuel mais peut être modifié à tout moment en définissant une balise de fin de rubrique à utiliser dans les fichiers README (par exemple : `fin_rubrique`) et en indiquant sa syntaxe dans la variable `$MOTIF_FIN_RUBRIQUE` du script `Perl/genere_catalogue_tests.pl`.
  - Apparition d'une nouvelle rubrique README : *Auteur* (renseigne le ou les auteurs du test avec prénom, nom, adresse mail). rubrique qui apparaît dans la documentation des tests.
- 2015-07-13 (Julien Troufflard) :
  - changement du nom du fichier de catalogue de tests (désormais s'appellera `catalogue_test.pdf` au lieu de `documentation_tests.pdf`)
  - le fichier `.pdf` de la document utilisateur principale est désormais archivé sous CVS. Il s'appelle `doc_procedure_verification.pdf` et est placé à la racine du projet (il s'agit simplement du fichier généré par `Doc/doc_procedure_verification.tex`)
- 2015-07-15 (Julien Troufflard) :
  - ajout d'une nouvelle version du script de lancement de la batterie. Cette nouvelle version en Perl s'appelle `verifier_exeHZ.pl`. Elle reproduit la même chose que la version actuelle en zsh mais propose en plus des options pour éviter de lancer tous les tests. La version précédente `verifier_exeHZ` est conservée mais renommée en `verifier_exeHZ.zsh`. A terme, cette version zsh sera supprimée du projet CVS.
- 2015-07-16 (Julien Troufflard) :
  - modif du script de génération automatique du catalogue de tests (script `Perl/genere_catalogue_tests.pl`). Désormais, il est possible d'insérer des figures via une syntaxe par balises dans les fichiers README des tests. La méthode est expliquée à la section 5.4 "Fichiers obligatoires" (fichier README)
  - ajout du script `Perl/verif_existence_repertoire_test.pl`. Permet de savoir si un nom de répertoire de test existe déjà dans la batterie. Est utile au moment de l'ajout d'un nouveau test pour aider à choisir le nom et éviter de choisir un nom déjà existant (même si ce n'est pas dans le même répertoire de destination car le

catalogue des tests ne tient pas compte du chemin complet). Explication de son utilisation à la section 5.1

- méthodologie pour ajouter un nouveau test : il n'y avait aucune section exposant clairement les étapes à suivre pour créer et ajouter un test, d'où la création de la section 5.2

- 2015-09-29 (Julien Troufflard) :

- modif du script Perl/verifier\_exeHZ.pl : le package `Term::ReadKey` n'est pas forcément installé sur toutes les machines. Pour éviter d'être pénalisé par ce package, son existence est vérifiée via la subroutine `check_install` du package `Module::Load::Conditional` et, si il existe, l'appel à `Term::ReadKey` est fait via `require` au lieu de `use`. Cette manière de faire pourra être appliquée partout où il y a un doute sur l'existence d'un package et définir ainsi un moyen de contourner l'absence d'un package.

- modif du script Perl/test.pl : il y avait une faille dans la subroutine `lancement_commande`. Sur certaines machines, la redirection `tee` ne se fait pas instantanément. Ceci conduisait à croire que le fichier de redirection n'était pas créé et donc renvoyait un problème lié à la redirection de l'affichage. Désormais, un délai est accordé pour attendre la création de ce fichier (environ 2 secondes maximum avec un check de l'existence du fichier de redirection toutes les millisecondes). De manière indirecte, ceci a conduit à créer un nouvel indicateur de status pour signaler si le calcul ne se lance pas (`probleme_lancement_calcul`).

- 2015-09-30 (Julien Troufflard) :

- modif de l'en-tête de tous les scripts perl (y compris les `.verif` de la batterie) : amélioration de la portabilité en modifiant le shebang selon [https://en.wikipedia.org/wiki/Shebang\\_%28Unix%29#Portability](https://en.wikipedia.org/wiki/Shebang_%28Unix%29#Portability). Désormais, les scripts commencent par `#!/usr/bin/env perl` au lieu du chemin absolu `#!/usr/bin/perl` qui pourrait ne pas fonctionner sur certaines machines à l'architecture atypique (et actualisation de la section 5.5 pour signaler cette habitude à prendre pour les fichiers `.verif`).

remarque : modif également pour le script zsh `verifier_exeHZ.zsh`

- 2015-11-24 (Julien Troufflard) :

- modif script Perl/genere\_catalogue\_tests.pl :
  - 1) possibilité d'insérer un espace "`\hspace`" entre les figures `\figures` (voir section 5.4 concernant le fichier README)
  - 2) possibilité d'insérer des formules et caractères mathématiques dans les fichiers README entre simples `$` ou doubles `$$` (voir section 5.4 concernant le fichier README)

- 2015-11-25 (Julien Troufflard) :

- modif script Perl/genere\_rapport.pl : modification de l'en-tête du rapport de tests avec notamment l'affichage du nom de la machine (via variable environnement `$HOST`)

- modif script Perl/test.pl :
  - 1) gestion de bugs et améliorations diverses :
    - remplacement de "cp -nf" par l'enchaînement "rm -f" + "cp" (car dans certaines versions de cp, l'option -n n existe pas)
    - modification majeure : désormais, un test s'exécute dans un repertoire de travail sur /tmp (voir variable \$repertoire\_de\_travail). L'intérêt majeur est de ne jamais modifier le répertoire d'origine du test quelque soit le déroulement du test. Cette stratégie permet actuellement d'interrompre la batterie avec ctrl+c sans conséquence sur le contenu des répertoires de test (voir procédure d'interruption à la section 3). De plus, une conséquence indirecte bénéfique de ceci est que même quand un test se déroule correctement, il se pouvait que certains fichiers soient légèrement modifiés (par exemple : le numéro de version Herezh en en-tête du fichier cube\_soude.her du test FONCTION\_UNIQUE/UTILITAIRES/Test\_R\_fusion-noeuds-voisins). En conséquence, cela génèrait un archivage CVS supplémentaire avec envoi intempestif d'un mail aux membres du projet pour chacun de ces fichiers, alors qu'en réalité, ils n'étaient pas fondamentalement et sciemment modifiés. Ce problème mineur est désormais résolu.
    - comparaison maple : désormais, si toutes les grandeurs d'un test sont OK, un affichage allégé est produit dans le rapport (juste une ligne pour indiquer que toutes les grandeurs sont OK sans détailler les comparaisons)
  - 2) affichage des temps de calcul :
    - dans la subroutine lancement\_commande() : lancement de Herezh via la tournure `tsch -c "time HZ -f fic.info" | tee fic.log` pour récupérer le temps CPU dans le fichier .log (l'utilisation de `tcsh -c`, c'est uniquement pour que le résultat de `time` soit bien récupéré sur STDOUT, donc dans `fic.log`)
    - juste après l'appel à `lancement_commande()` ⇒ saisie du temps de calcul dans le .log et affichage dans le rapport de test (juste après le nom du test)
- 2015-11-25(bis) (Julien Troufflard) :
  - modif scripts Perl/genere\_catalogue\_tests.pl et Perl/genere\_rapport.pl : suppression de l'option -s dans le shebang (inutile et en plus génère un bug sur Linux quand on utilise la tournure `#!/usr/bin/env perl -s`)
  - modif script Perl/test.pl :
    - remplacement de la subroutine `return_nb_decimales()` par la subroutine `return_nb_decimales_first()` (même chose mais en mieux : renvoie la position de la première décimale non nulle au lieu du nombre total de décimales). Pour rappel, ce traitement n'a pas pour but d'arrondir les résultats pour la comparaison. Il s'agit juste d'un arrondi à but cosmétique pour afficher, dans le rapport, la grandeur avec un nombre de décimales adapté à la précision.
    - comparaison maple :
      - 1) modification de la comparaison pour gérer le cas où le nombre est très petit (par exemple : 1.e-15 comparé à 1.e-30 génère une erreur relative énorme). Ce problème avait été déjà pensé dans la version historique de la vérification Herezh (Laurent Mahéo) mais malheureusement non traité dans la nouvelle version. Désormais, on traite ce problème en arrondissant les grandeurs à 1.e-12 près.

2) correction d'une erreur dans la comparaison absolue : par erreur, la comparaison se faisait toujours par rapport à la précision absolue par défaut (`$PREC_ABSOLU`) sans tenir compte de l'éventuelle modification par fichier `.precision`