Fiche 2 — Consulter, rediriger, rechercher

Révision et mise à niveau

Emmanuel BRUNO
Université de Toulon, LIS UMR CNRS 7020
emmanuel.bruno@univ-tln.fr

2026-11-10

Table des matières

1 Objectifs pédagogiques 1
2 Préparation de l'environnement de travail 2
3 Le jeu de données de la fiche 2
4 Point de contrôle 3
5 Parcours pédagogiques 3
5.1 ◯ Lire et afficher 3
5.1.a Consulter un fichier : `cat` 3
5.1.b Tête et queue : `head` et `tail` 3
5.1.c Compter : `wc` 4
5.1.d Anatomie d'une commande 4
5.2 ◯ Les redirections 4
5.2.a Rediriger le résultat d'une boucle 5
5.3 ◯ Rechercher avec `grep` 6
5.3.a Deux dialectes : BRE et ERE 7
5.3.b Référentiel de motifs ERE 7
5.3.c Lire la documentation : `man` 8
6 Auto-vérification 8
7 À retenir 9

1 Objectifs pédagogiques

À l'issue de cette fiche, vous serez capable de :

- afficher un fichier avec `cat`, `head` et `tail` ;
- compter des lignes avec `wc` ;
- rediriger une sortie vers un fichier avec `>` et `>>` ;
- rechercher avec `grep` et ses options usuelles (`-n`, `-v`, `-i`, `-c`) ;
- distinguer un motif de base (BRE) d'un motif étendu (ERE).

⚠️ Périmètre de cette fiche

Les coupes avec `cut`, `tr` et `sed` sont traitées dans la **fiche 3** ; la composition `cmd1 | cmd2` dans la **fiche 4**. Les notations `2>`, `2>&1` et la commande `tee` apparaissent seulement comme repères dans les tableaux : elles ne sont pas exécutées ici. De même, `less` et `man` sont **interactifs** : décrits, jamais exécutés dans une cellule.

2 Préparation de l'environnement de travail

Le `rm -rf` est **borné à ce sous-répertoire** ; `"${WORK:?}"` interdit toute suppression si la variable était vide.

Si une cellule plus bas affiche un message commençant par `ERREUR :` — dans cette fiche `ERREUR : setup` ou `ERREUR : répertoire absent` —, la garde a fait `exit 1`, ce qui **termine le shell de la cellule** : le moteur de calcul en redémarre un neuf, sans variable ni répertoire courant, et la cellule en échec est malgré tout marquée comme exécutée, pas comme fausse. Relancer la préparation puis **toutes** les cellules qui suivent, dans l'ordre, est la seule reprise sûre : la préparation recrée le répertoire depuis zéro.

```
WORK=~/work/assignments/shell_tp/fiche02
rm -rf "${WORK:?}" && mkdir -p "${WORK}" && cd "${WORK}" || { echo "ERREUR :
setup"; exit 1; }
```

3 Le jeu de données de la fiche

Première écriture avec `>` : elle **crée** et **écrase**. Les suivantes avec `>>` : elles **ajoutent** à la fin.

```
cd ~/work/assignments/shell_tp/fiche02 || { echo "ERREUR : répertoire absent —
exécuter la cellule de préparation"; exit 1; }
echo "Ligne 1 : bienvenue dans la fiche 2" > demo.txt
echo "Ligne 2 : consulter des fichiers" >> demo.txt
echo "Ligne 3 : rediriger les sorties" >> demo.txt
echo "Ligne 4 : rechercher avec grep" >> demo.txt
echo "Ligne 5 : ERROR : une erreur à retrouver" >> demo.txt
echo "Ligne 6 : une ligne de commentaire" >> demo.txt
echo "Ligne 7 : ERROR : une seconde erreur" >> demo.txt
echo "Ligne 8 : WARNING : un avertissement" >> demo.txt
echo "Ligne 9 : la fin approche" >> demo.txt
echo "Ligne 10 : dernière ligne du fichier" >> demo.txt
echo "2026-11-04 08:00:01 INFO  service démarré" > app.log
echo "2026-11-04 08:00:02 ERROR impossible d'ouvrir le port 8080" >> app.log
```

```
echo "2026-11-04 08:00:03 DEBUG état interne : ready" >> app.log
echo "2026-11-04 08:00:04 INFO  connexion acceptée" >> app.log
echo "2026-11-04 08:00:05 ERROR délai dépassé vers la base" >> app.log
echo "2026-11-04 08:00:06 DEBUG requête exécutée en 12 ms" >> app.log
echo "2026-11-04 08:00:07 INFO  arrêt demandé" >> app.log
echo "2026-11-04 08:00:08 WARNING database : reconnexion en cours" >> app.log
echo "warning: entrée invalide ignorée" >> app.log
echo "alpha" > fichier.txt
echo "beta" >> fichier.txt
echo "gamma" >> fichier.txt
echo "Jeu de données prêt."
```

4 Point de contrôle

`pwd` doit afficher un chemin terminé par `shell_tp/fiche02`. Sinon, réexécutez la cellule de préparation **puis toutes les cellules qui suivent, dans l'ordre** : relancer la seule préparation laisserait la fiche dans un état half-fait.

```
pwd
```

5 Parcours pédagogiques

Trois onglets : **Lire et afficher**, **Les redirections**, **Rechercher avec `grep`**, à travailler dans cet ordre.

5.1 ◯ Lire et afficher

5.1.a Consulter un fichier : `cat`

`cat` envoie le contenu sur la **sortie standard**, c'est-à-dire le terminal.

```
cat demo.txt
```

Sur un fichier long, le début défile hors de l'écran : c'est le rôle de l'afficheur interactif `less` (flèches, `Espace`, `q` pour quitter). Il ne se lance pas dans une cellule, qui attendrait une touche sans jamais se terminer.

```
less demo.txt
```

5.1.b Tête et queue : `head` et `tail`

`head` affiche les **premières** lignes, `tail` les **dernières** ; `-n` fixe le nombre.

```
echo "--- head -n 3 ---"
head -n 3 demo.txt
echo "--- tail -n 3 ---"
tail -n 3 demo.txt
```

`tail -f` suit l'écriture en continu : interactif, donc réservé au terminal.

5.1.c Compter : `wc`

`wc` (word count) donne lignes, mots, octets ; `-l`, `-w`, `-c` pour un seul chiffre.

```
wc demo.txt app.log
echo "---"
wc -l demo.txt
wc -l app.log
```

Repères pour toute la fiche : `demo.txt` fait **10 lignes**, `app.log` en fait **9**. Piège classique : `wc -l` compte des **sauts de ligne** ; une dernière ligne non terminée n'est pas comptée. `echo` ajoute toujours ce saut, nos fichiers sont fiables.

5.1.d Anatomie d'une commande

```
commande  options  arguments
grep      -n       ERROR   app.log
```

- la **commande** : le programme à exécuter ;
- les **options** : préfixées par `-`, elles changent le comportement ;
- les **arguments** : sur quoi la commande travaille.

Les options se regroupent : `grep -i -n` et `grep -in` sont équivalents.

5.2 ◯ Les redirections

Trois flux, trois descripteurs :

Flux	Descripteur	Par défaut
`stdin`	0	le clavier
`stdout`	1	le terminal
`stderr`	2	le terminal

Opérateur	Effet
`>`	**écrase** `stdout` dans le fichier (le crée)
`>>`	**ajoute** à la fin (le crée)

Opérateur	Effet
`2>`	idem pour `stderr`
`2>&1`	envoie `stderr` là où va `stdout`
`tee`	affiche **et** écrit

Le shell intercepte ces caractères **avant** de lancer la commande.

```
echo "Premier jet" > sorties.txt
echo "--- apres le premier > ---"
cat sorties.txt
echo "Deuxieme jet" > sorties.txt
echo "--- apres le second > : le premier jet a disparu ---"
cat sorties.txt
echo "Ajout avec >>" >> sorties.txt
echo "--- apres >> ---"
cat sorties.txt
wc -l sorties.txt
```

C'est l'erreur la plus fréquente du débutant : taper `>` là où on voulait `>>`. Le fichier compte au final **2 lignes**.

5.2.a Rediriger le résultat d'une boucle

Placé après `done`, le `>` collecte tout ce que la boucle affiche.

```
for i in $(seq 1 20); do
  echo "note $i : issue de fichier.txt"
done > notes.txt
wc -l notes.txt
head -n 2 notes.txt
tail -n 2 notes.txt
```

`notes.txt` contient **20 lignes** : le compte de `wc -l` sert de contrôle.

> **i** Exercice — Créer un jeu de données
>
> Dans `exercice.txt`, écrivez exactement **3 lignes** de votre choix, vérifiez avec `wc -l`, puis provoquez l'écrasement avec un second `>` et constatez-le.

5.3 ◯ Rechercher avec `grep`

```
echo "--- grep ERROR ---"
grep "ERROR" app.log
echo "--- grep -n ERROR ---"
grep -n "ERROR" app.log
echo "--- grep -c -v DEBUG ---"
grep -cv "DEBUG" app.log
```

`grep` affiche la **ligne entière**, pas seulement le mot trouvé. Ci-dessus : lignes **2** et **5**, puis **7** lignes hors `DEBUG`.

Option	Effet
`-n`	numéro de chaque ligne trouvée
`-v`	**inverse** la sélection
`-i`	ignore la **casse**
`-c`	**compte** au lieu d'afficher
`-l`	nom des fichiers seulement
`-E`	motifs **étendus** (ERE)

> ⚠ `grep` est sensible à la casse
>
> `app.log` contient un `WARNING` majuscule **et** un `warning:` minuscule. Sans `-i`, l'un seul est vu ; avec `-i`, les deux. C'est la première cause de « ça ne marche pas ».

```
echo "--- grep -c WARNING ---"
grep -c "WARNING" app.log
echo "--- grep -ic warning ---"
grep -ic "warning" app.log
echo "--- grep -in database ---"
grep -in "database" app.log
echo "--- grep -n ^warning: ---"
grep -n "^warning:" app.log
```

Les quatre comptes attendus : **1**, **2**, ligne **8**, ligne **9**. L'ancre `^` ne trouve que la ligne 9, là où `warning` seul aurait aussi trouvé la ligne 8.

5.3.a Deux dialectes : BRE et ERE

Par défaut `grep` parle **BRE** (motifs de base) ; `-E` active l'**ERE** (motifs étendus). Les deux dialectes se comportent à l'identique pour la plupart des métacaractères et ne diffèrent que sur deux points utiles : l'opérateur « un ou plusieurs » et l'alternance.

Identiques dans les deux dialectes, à écrire tels quels :

- `^ERROR` — début de ligne ;
- `ERROR$` — fin de ligne ;
- `t?s` — zéro ou un caractère ;
- `ab*c` — zéro ou plusieurs `b` ;
- `[0-9]` et `[a-z]` — classes de caractères ;
- `.{5}` — exactement cinq caractères quelconques.

Ce qui change :

- « un ou plusieurs `b` » : BRE `ab\+` → ERE `ab+` ;
- « `a` ou `b` » (alternance) : BRE `a\|b` → ERE `a|b`.

Le corollaire pratique : **en ERE, un tube non protégé est toujours une alternance**. Pour rechercher un tube littéral, placez-le entre crochets dans les deux dialectes : `a[|]b`.

Les motifs de ce qui suit contiennent des tubes ; ils sont donc donnés en liste et non en tableau, parce qu'un `|` à l'intérieur d'une cellule Markdown ferme la colonne et rend le motif illisible.

5.3.b Référentiel de motifs ERE

À copier et adapter, toujours avec `grep -E` :

- `^ERROR` — lignes commençant par `ERROR` ;
- `ERROR$` — lignes finissant par `ERROR` ;
- `warn(ing)?` — `warn` ou `warning` ;
- `(INFO|ERROR)` — l'un ou l'autre de ces deux niveaux ;
- `(0[1-9]|1[0-2])-(0[1-9]|[12][0-9]|3[01])` — une date au format `MM-JJ` ;
- `.{5}` — exactement cinq caractères.

Vérifions trois de ces motifs sur le fichier de journal :

```
echo "--- ERE : INFO ou ERROR (compte) ---"
grep -Ec 'INFO|ERROR' app.log
echo "--- ERE : warn ou warning ---"
grep -En 'warn(ing)?' app.log
echo "--- ERE : horodatage complet ---"
grep -En '2026-11-04 08:00:0.' app.log
```

Valeurs attendues : **5** lignes pour le premier (`INFO` ×3 et `ERROR` ×2), la **ligne 9** pour le second — le `WARNING` majuscule de la ligne 8 n'est pas trouvé, `grep` est sensible à la casse — et les **neuf** lignes d'horodatage pour le troisième.

5.3.c Lire la documentation : `man`

`man grep` ouvre le manuel. Comme `less`, interactif : dans un terminal, jamais en cellule.

```
man grep
```

> **i** Exercice — Recherche basique
>
> Dans `app.log`, retrouvez : (1) le numéro de la ligne qui parle de `database` ; (2) le nombre de lignes qui ne sont pas `DEBUG` ; (3) les lignes commençant par la date `2026`.
>
> Corrigé : `grep -in "database" app.log`, `grep -cv "DEBUG" app.log`, `grep -n "^2026" app.log`.

6 Auto-vérification

```
cd ~/work/assignments/shell_tp/fiche02 || { echo "ERREUR : répertoire absent — exécuter la cellule de préparation"; exit 1; }
total=0
ok=0
check() {
  total=$((total + 1))
  if [ "$2" = "$3" ]; then
    ok=$((ok + 1))
    echo "OK   $1 : $2"
  else
    echo "KO   $1 : obtenu [$2], attendu [$3]"
  fi
}
check "demo.txt lignes" "$(grep -c '' demo.txt)" "10"
check "app.log lignes" "$(grep -c '' app.log)" "9"
check "ERROR" "$(grep -c 'ERROR' app.log)" "2"
check "WARNING exact" "$(grep -c 'WARNING' app.log)" "1"
check "warning avec -i" "$(grep -ic 'warning' app.log)" "2"
check "hors DEBUG" "$(grep -cv 'DEBUG' app.log)" "7"
check "notes.txt lignes" "$(grep -c '' notes.txt)" "20"
echo "OK : ${ok} / ${total}"
```

Sept `OK` : votre environnement est conforme. Un `KO` signifie qu'une cellule de préparation n'a pas été exécutée — reprenez depuis le haut.

7 À retenir

Question	Réponse
Afficher un petit fichier	`cat fichier`
Début / fin d'un fichier	`head -n 5` / `tail -n 5`
Long fichier, à l'écran	`less fichier` (interactif)
Compter des lignes	`wc -l fichier`
Écrire en écrasant	`cmd > fichier`
Écrire en ajoutant	`cmd >> fichier`
Chercher	`grep "motif" fichier`
Ignorer la casse	`grep -i`
Numéroter	`grep -n`
Compter les résultats	`grep -c`
Tout sauf le motif	`grep -v`
Motifs étendus	`grep -E`

Trois réflexes évitent l'essentiel des erreurs : **vérifiez `pwd` avant d'écrire**, **distinguez `>` de `>>`**, **souvenez-vous que `grep` est sensible à la casse**.

La **fiche 3** reprend ces fichiers avec `cut`, `tr` et `sed`. La **fiche 4** assemble les commandes en pipelines.