Monitorização de interfaces de rede em bash

João Eduardo dos Santos Alcatrão Nºmec:76763

Neste relatório pretendo explicar o meu raciocínio ao escrever o script netifstat.sh e todas as suas funcionalidades e opções (conforme as indicações presentes no guião que explica o objetivo do trabalho).

Objetivo do trabalho:

O propósito do trabalho é criar um script (netifstat.sh) em Bash que consiga fornecer estatísticas sobre a quantidade de dados recebidos e transmitidos nas várias interfaces de rede detetadas pelo sistema. Além disto, é pedido que estes dados sejam obtidos em função de um intervalo de tempo, que é o único parâmetro obrigatório que este script deve receber. Também é pedido que sejam geradas e implementadas opções que podem, ou não, requerer argumentos, que servem para formatar, ordenar ou filtrar esses dados, e é pedida ainda uma opção que requer a obtenção de novos dados e estatísticas indefinidamente, até o script parar.

O modo em que estas funcionalidades foram implementadas vai ser falado em detalhe ao longo deste relatório.

Raciocínio inicial:

Ao ler o guião, surgiu-me uma questão: onde posso obter a informação e dados requeridos para resolver o problema? No próprio guião haviam várias respostas a esta pergunta, mas a resposta que mais me agradou e que acabei por usar (e que foi aludida pelo professor na aula) encontra-se em /proc/net/dev. Este ficheiro virtual contém vários detalhes e aspetos das redes de interface do sistema, incluindo a quantidade de dados recebida e transmitida por cada interface.

Estes dados pareceram-me ser (e conter) mesmo os dados que o problema requeria. Eram dados que representavam sobretudo estatísticas totais (desde que o sistema foi iniciado até ao momento presente) sobre as interfaces, mas bastava filtrá-los e formatá-los numa tabela para obter algo muito semelhante ao produto final mostrado nas imagens do guião.

E assim o fiz, e guardei todas as informações que considerei relevantes para o problema em 2 variáveis (separadas temporalmente pelo valor do parâmetro obrigatório requerido):

```
function sono {
ini=( $(cat /proc/net/dev | awk {'print $1, $2, $10'} | grep -v -i 'bytes\|receive') )
sleep $alienmir
fim=($(cat /proc/net/dev | grep -v -i 'bytes\|receive' | awk {'for(i=1;i<3;i++) print $i;print $10'}))
}
```

Fig.1: Função “sono”, a primeira função que foi escrita no script, que obtém, em 2 instâncias separadas (em segundos) pelo argumento obrigatório, a informação e dados considerados relevantes, e os guarda em 2 variáveis distintas.

Tendo estas variáveis (arrays “ini” e “fim”, em que cada um contém o nome de cada interface detetada e as suas quantidades de dados recebidos e enviados, todos seguidos, de forma que a cada 3 índices se possa encontrar o mesmo atributo [nome, dados recebidos ou dados enviados] da interface seguinte), poderia manipulá-las de modo a obter a quantidade de dados transmitidos e recebidos no período de tempo indicado (valor guardado na variável “alienmir”, e executado com o comando ‘sleep’), bem como as suas respetivas taxas de transferência, e imprimi-los em formato de tabela no ecrã. Esta manipulação manifestou-se na forma de ciclos “for” e manuseamento de arrays em Bash, como se mostra na implementação das funções “UFO” e “Apparate”, que calculam os valores desejados e os imprimem no stdout, respetivamente.

O cálculo de TX e RX fez-se, para cada interface, através da subtração dos seus respetivos dados transferidos antes do sleep aos dados transferidos após o sleep. Os valores TRATE e RRATE de cada interface foram obtidos através da divisão de TX e RX pelo argumento obrigatório (e argumento do comando sleep; “alienmir”).

```
function UFO {
NETIF=()
RX=()
TX=()
RRATE=()
TRATE=()

#NETIF
for ((i=0; i<${#ini[@]}; i=i+3)); do
        NETIF+=(${ini[i]})
done

#RX
for ((i=1; i<${#ini[@]}; i=i+3)); do
        RX+=($((${fim[i]}-${ini[i]})))
done

#TX
for ((i=2; i<${#ini[@]}; i=i+3)); do
        TX+=($((${fim[i]}-${ini[i]})))
done

#RRATE
for ((i=0; i<${#RX[@]}; i++)); do
        conta=$(echo "${RX[i]}, $alienmir" | awk '{printf "%.1f \n", $1/$2}')
        RRATE+=( $conta )
done

#TRATE
for ((i=0; i<${#TX[@]}; i++)); do
        conta=$(echo "${TX[i]}, $alienmir" | awk '{printf "%.1f \n", $1/$2}')
        TRATE+=( $conta )
done
}



function apparate {
for (( i=0; i<${#NETIF[@]}; i++ )); do
        echo ""${NETIF[i]}"     "${TX[i]}"      "${RX[i]}"      "${TRATE[i]}"   "${RRATE[i]}""
done
}
```

Fig.2: Funções “UFO” (que calcula os valores desejados) e “Apparate” (que imprime em formato de tabela esses mesmos valores).

Feitos os cálculos para obter os valor de TX, RX, TRATE E RRATE para cada interface, estes ficam aptos para serem impressos em formato de tabela. Para cada interface detetada, é enviado para o stdout 1 linha com 5 colunas, sendo a 1ª coluna correspondente ao nome da interface e as restantes correspondentes aos 4 atributos acima mencionados, usando o comando ‘echo’ para gerar estas mesmas linhas. O produto disto é uma tabela bastante semelhante à fornecida como exemplo no guião.

E isto pareceu-me ser (um)a solução inicial para o problema inicial (obter estatísticas de redes de interface conforme mostrado no guião). Faltava-me agora pensar nas opções, e integrá-las no script.

Raciocínio acerca das opções:

Ao me deparar com o problema das opções (como as implementar, por onde começar) decidi estudar um dos comandos recomendados no guião, ao qual o professor aludiu que poderia ser de grande uso para lidar com opções: o comando 'getopts'.

Comecei por me familiarizar com a sua sintaxe, com a sua estrutura e com o seu uso, e assim criei uma primeira versão das opções. Nesta versão, cada opção fazia o que eu queria que fizesse, mas fazia-o de maneira isolada, exclusiva das outras opções. O que não era ideal, pois não era o que o guião pedia. É incompatível usar a opção -k e a opção -m em simultâneo, mas não me fez muito sentido que devesse ser uma incompatibilidade usar a opção -k e -v. Ou -m e -p. Portanto, esta primeira versão deu lugar a uma outra versão, em que eram usadas "flags" para sinalizar que opções deviam de ser processadas, e que combinações deviam de ser proibidas. O uso das flags também permitiu que permutações distintas do mesmo conjunto de opções (ex: -c e -p, ou -p e -c) não produzissem resultados diferentes ou inconsistentes (que no caso destas 2 opções dadas como exemplo, era algo que acontecia com a primeira versão do tratamento de opções).

```
flag_c=0
flag_c_arg="tee"
flag_bytes_arg="tee"
flag_p=0
flag_p_arg="tee"
flag_ordem_arg="tee"
flag_v=0
flag_v_arg="tee"
flag_l=0

flag_ordem=0
flag_bytes=0

while getopts "c:bkmp:trTRvl" option; do
        case $option in
                c) #filtro de redes
                if [ $flag_c -eq 0 ]; then
                        flag_c=1
                        if [[ -z "${OPTARG// }" ]]; then #ver se o argumento não existe ou é vazio ou é espaço branco
                                echo "Providencie um parâmetro à opção -c"
                                exit 1
                        fi
                        flag_c_arg="grep -i $OPTARG.*:"
                else
                        echo "Insira apenas 1 filtro de redes"
                        exit 1
                fi
                ;;

                b) #imprimir em bytes (default)
                if [[ $flag_bytes -eq 0 ]]; then
                        flag_bytes=1
                else
                        echo "Especifique apenas 1 parâmetro de quantidade para representar os bytes transferidos"
                        exit 1
                fi
                ;;

                k) #imprimir em kilobytes
                if [[ $flag_bytes -eq 0 ]]; then
                        flag_bytes_arg=' { printf ( "%s\t", $1) ; for ( i=2 ; i<=NF; i++ )  printf ("%.1f\t", $i/=1024 ); print ""  } ' #em
incluido nesta expressão e com whitespace desta maneira (unexpected ''' ou missing { near end of file). Normalmente, deixa usar whitespace
```

Fig.3: Inicialização das flags e começo da estrutura do comando getopts, com algumas validações.

```
p) #imprimir até 'p' interfaces
if [[ $flag_p -eq 0 ]]; then
        flag_p=1
        if [[ ! $OPTARG =~ ^[0-9]+$ ]]; then
                echo "Indique um número natural à opção -p"
                exit 1
        fi
        flag_p_arg="head -"$OPTARG" "
else
        echo "Insira apenas 1 (ou nenhum) nº (limite máximo) de interfaces a serem visualizadas"
        exit 1
fi
;;

t) #ordenar segundo TX
if [[ $flag_ordem -eq 0 ]]; then
        flag_ordem=1
        flag_ordem_arg="sort -k2 -n "
else
        echo "Parâmetros de ordenação inválidos (especifique apenas 1 ordem, com ou sem inverso)"
        exit 1
fi
```

Fig.4: Tratamento das opções -p e -T.

O comando getopts serviu assim para definir o que é suposto as opções fazerem, serviu para validar estas mesmas e os seus parâmetros, e atualizar estes conforme suposto. Neste último ponto surgiu outra dúvida: devo criar ou alterar os arrays que contém todos os dados de modo a ficarem apenas com os dados que as opções pedem, ou devo verificar e incluir logo à partida que dados entram nas variáveis iniciais, em virtude das opções? Na verdade, não pensei muito numa abordagem ou noutra e, tendo notado da primeira versão das opções que todas estas (exceto a -l) são essencialmente filtragens, substituições, e restrições de tamanho aos dados a apresentar, e que virtualmente tinha usado ‘pipes’ para todas elas, decidi fazer uma cadeia de pipes que filtra toda a informação de acordo com as opções dadas e que produz um resultado consistente com as mesmas. Para conseguir tal, foi necessário descobrir um “placeholder” (o comando ‘tee’) para as opções não usadas não terem impacto no produto final.

Entrando em maior detalhe em como as opções foram conseguidas, todas as opções de ordenação (-t, -r, -T, -R) tiram partido do comando ‘sort’ (e dos argumentos de sort k2, k3, k4 e k5, respetivamente) para ordenar a tabela impressa (existe uma variável para guardar o tipo de ordenação, sendo esta usada para estas 4 opções, visto que são mutuamente exclusivas).

A opção -v também usa o comando sort, mas pode ser usada em combinação com as opções acima, pelo que é usada uma variável diferente das de cima para guardar o valor de argumento.

A opção -c utiliza o comando ‘grep’ e uma expressão regular para efetuar a seleção de interfaces a visualizar, sendo essa

expressão o argumento que o utilizador tem de passar para esta opção, seguido da expressão ".*:" (para filtrar apenas os nomes, que acabam todos em ":").

A opção -p tira partido do comando 'head', e de um argumento (também passado pelo utilizador) para estabelecer o nº máximo de linhas da tabela a serem apresentadas.

Na opção -l é usado um ciclo 'while', onde são calculados cumulativamente os valores TXTOT e RXTOT para cada interface a partir dos valores TX e RX obtidos a cada iteração. Uma tabela específica é impressa caso esta opção seja ativada.

As opções que regem as medidas de quantidade em que deve ser apresentada a informação (-b, -k e -m) aproveitam-se do comando 'awk', e de uma expressão muito dolorosa de se obter (e que funciona de maneira bastante diferente ao que me habituei a fazer nos guiões, visto que esta quase nunca aceita whitespace em qualquer lado em toda a expressão, pelo que tive que utilizar 2 variáveis para poder definir uma expressão supostamente única do valor de argumento) que faz a divisão de cada valor numérico da tabela (que não esteja nos nomes das interfaces) por 1024 (-k) ou 1024*1024 (-m).

Ultrapassado o getopts e vistas as funções principais acima, o código "main", se assim se pode ser referido, foi bastante mais simples, imediato e indolor de ser escrito. No final, a tabela impressa é sujeita à tal cadeia de pipes referida.

```
alienmir=$1
if [[ ! $alienmir =~ ^[0-9]+$ ]] || [[ $# -ne 1 ]]; then
        echo "Insira um (só) argumento válido (número inteiro positivo)"
        luz
        exit 1
fi


if [[ $flag_l -eq 0 ]]; then
        sono
        mau
        UFO
        echo "NETIF     TX      RX      TRATE   RRATE"
        apparate | $flag_c_arg | $flag_bytes_arg | $flag_ordem_arg | $flag_v_arg | $flag_p_arg

else
while true; do

        #dor #útil se o loop der um erro que não o pára
        sono
        mau
        UFO
        echo "NETIF     TX      RX      TRATE   RRATE   TXTOT   RXTOT"

        #TXTOT
        for ((i=0; i<${#TX[@]}; i++)); do
                TXTOT[i]=$(( ${TXTOT[i]}+${TX[i]}))
        done

        #RXTOT
        for ((i=0; i<${#TX[@]}; i++)); do
                RXTOT[i]=$(( ${RXTOT[i]}+${RX[i]} ))
        done

        for (( i=0; i<${#NETIF[@]}; i++ )); do
                echo ""${NETIF[i]}"     "${TX[i]}"      "${RX[i]}"      "${TRATE[i]}"   "${RRATE[i]}"   "${TXTOT[i]}"   "${RXTOT[i]}""
        done | $flag_c_arg | $flag_bytes_arg | $flag_ordem_arg |$flag_v_arg | $flag_p_arg

        echo ""
done
fi
```

Fig.5: Estrutura "main", a tabela que é produzida pela função "Apparate" mencionada acima é submetida a uma cadeia de pipes que conduzem a comandos como awk, sort, grep e head, conforme as opções que foram passadas, ou ao comando tee, caso não tenham sido passadas.

Após e durante a implementação da estrutura getopts e da cadeia de pipes final (sendo que foi criada uma cadeia nova no caso da opção -l ter sido ativada, visto que esta requer a obtenção de novos dados), foram feitas algumas verificações e validações, quer dos argumentos opcionais, quer do argumento obrigatório, quer de alguns aspetos importantes para assegurar o bom funcionamento do script.

Destacam-se neste sentido a função “dor” (que termina a execução do script assim que se encontre qualquer erro que de outro modo não fosse terminal, como alguns erros que pudessem acontecer ao executar comandos awk ou grep), e a função “mau” (que termina o script se houverem inconsistências entre as interfaces detetadas no início e no fim do intervalo de tempo indicado).

```
#!/bin/bash

function dor {
set -e -o pipefail
if [ $? -ne 0 ] && [ $? -ne 1 ] && [ $? -ne 130 ]; then trap 'luz' EXIT; fi
}
dor
```

```
function mau {
if [[ ${#ini[@]} -ne ${#fim[@]} ]]; then
        echo "Novos dispositivos na rede"
        exit
fi
for ((i=0; i<${#ini[@]}; i=i+3)); do
        coisa=${ini[i]}; coisa=${coisa%?}
        cena=${fim[i]}; cena=${cena%?}
        if [[ $coisa -ne $cena ]]; then
                echo "Novas interfaces na rede"
                exit
        fi
done
}
```

Fig.6: Função “dor” (que é a primeira função/comando a ser executado no script) e a função “mau” (que procura inconsistências na quantidade e qualidade de interfaces guardadas nos arrays antes e após o sleep).

Resultados:

```
joao@joao-HP-Laptop-17-cp0xxx:~/Desktop/Sistemas Operativos/Trabalho Prático 1$
 ./netifstat.sh 1
NETIF   TX      RX      TRATE   RRATE
lo:     0       0       0,0     0,0
wlp2s0: 0       42      0,0     42,0
```

Fig.7: Execução do script netifstat.sh com o parâmetro obrigatório.

```
joao@joao-HP-Laptop-17-cp0xxx:~/Desktop/Sistemas Operativos/Trabalho Prático 1$
 ./netifstat.sh -c "wl" 3
NETIF   TX      RX      TRATE   RRATE
wlp2s0: 32733   164953  10911,0 54984,3
```

Fig.8: Execução do script netifstat.sh com a opção -c (e com o respetivo argumento) ativada.

```
joao@joao-HP-Laptop-17-cp0xxx:~/Desktop/Sistemas Operativos/Trabalho Prático 1
$ ./netifstat.sh -p 1 1
NETIF   TX      RX      TRATE   RRATE
lo:     1754    1754    1754,0  1754,0
```

Fig.9: Execução do script netifstat.sh com a opção -p (e com o respetivo argumento válido) ativada.

```
joao@joao-HP-Laptop-17-cp0xxx:~/Desktop/Sistemas Operativos/Trabalho Prático 1$
 ./netifstat.sh -t -v 9
NETIF   TX      RX      TRATE   RRATE
wlp2s0: 28790   202819  3198,9  22535,4
lo:     13698   13698   1522,0  1522,0
```

Fig.10: Execução do script netifstat.sh com as opções -t e -v ativadas.

```
joao@joao-HP-Laptop-17-cp0xxx:~/Desktop/Sistemas Operativos/Trabalho Prático 1$
 ./netifstat.sh -t -k -p 2 10
NETIF   TX      RX      TRATE   RRATE
lo:     6,3     6,3     0,6     0,6
wlp2s0: 6,9     3,7     0,7     0,4
```

Fig.11: Execução do script netifstat.sh com as opções -t, -k e -p ativadas.

```
joao@joao-HP-Laptop-17-cp0xxx:~/Desktop/Sistemas Operativos/Trabalho Prático 1$
 ./netifstat.sh -v -m -p 1 14
NETIF   TX      RX      TRATE   RRATE
wlp2s0: 0,1     0,7     0,0     0,1
```

Fig.12: Execução do script netifstat.sh com as opções -v, -m e -p ativadas.

```
joao@joao-HP-Laptop-17-cp0xxx:~/Desktop/Sistemas Operativos/Trabalho Prático 1$
 ./netifstat.sh -t -c "lo" -k -p 2 10
NETIF   TX      RX      TRATE   RRATE
lo:     13,2    13,2    1,3     1,3
```

Fig.13: Execução do script netifstat.sh com as opções -t, -c, -k e -p ativadas.

```
joao@joao-HP-Laptop-17-cp0xxx:~/Desktop/Sistemas Operativos/Trabalho Prático 1$
 ./netifstat.sh -k -c "l" -R -p 1 -l 5
NETIF   TX      RX      TRATE   RRATE   TXTOT   RXTOT
wlp2s0: 0,5     0,0     0,1     0,0     0,5     0,0

NETIF   TX      RX      TRATE   RRATE   TXTOT   RXTOT
wlp2s0: 0,6     0,0     0,1     0,0     1,1     0,0

NETIF   TX      RX      TRATE   RRATE   TXTOT   RXTOT
wlp2s0: 0,9     0,0     0,2     0,0     2,0     0,0
```

Fig.14: Execução do script netifstat.sh com as opções -k, -c, -R, -p e -l ativadas.

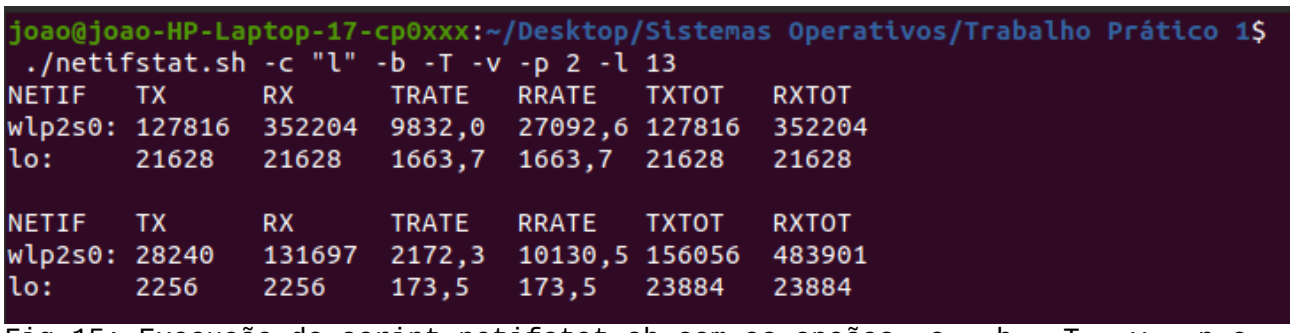

Fig.15: Execução do script netifstat.sh com as opções -c, -b, -T, -v, -p e -l ativadas.

Estes resultados provêm de uma amostra de uma série de testes, na qual examino o resultado da execução do script com cada opção ativa individualmente (e sem opções), e com duas opções ativas em simultâneo, e com mais opções ativas, e com diferentes ordens, de maneira a exaustar as opções possíveis de serem utilizadas. Vários argumentos para as opções que assim os requerem foram utilizados, o que permitiu ter uma noção aprimorada de quais verificações justificavam serem aplicadas.

Penso que o script netifstat.sh ao qual este relatório concerne é uma solução satisfatória ao problema dado (na pior das hipóteses, na forma que interpretei o mesmo). A informação referente à quantidade de dados recebidos e transmitidos que o script imprime no stdout provém da informação disponível no ficheiro virtual proc/net/dev, e aparenta ser consistente com esta (quantidades de dados recebidos/transmitidos totais impressos pelo script e mostrados no ficheiro virtual são muito semelhantes para o mesmo intervalo de tempo). As taxas de transferência também aparentam fazer sentido (13 bytes recebidos em 10 segundos traduzem-se numa taxa de transferência de 1.3 bytes/segundo). Para minimizar erros que possam ser cometidos pelo utilizador ao executar o script com argumentos e parâmetros não ideais, são feitas um conjunto de verificações e validações que me aparentam ser quantitativa e qualitativamente razoáveis. Todas as opções aparentam funcionar de acordo com o que é esperado, e mostram resultados idênticos aqueles mostrados no guião.

Raciocínio/Reflexão final:
Tendo concluído a escrita do script netifstat.sh e verificado os resultados obtidos com diversas opções e parâmetros, aproveitei para refletir sobre o trabalho que tinha feito, com o propósito de inferir e resumir alguns aspetos que, embora me pareçam ser imediatos e básicos agora, nem sequer pensei no início da escrita do script. Talvez (nem todos os raciocínios que aqui escreva sejam corretos ou lógicos ou até façam sentido, mas acredito que) a geração do mesmo fosse muito mais rápida e eficaz se eu tivesse em consideração à partida os seguintes pontos, (que também se encontram anotados em comentário no final do script):

-Se obter por defeito tudo o que é necessário, então as opções, em princípio, servem como filtros ou subtrações (retira-se, ou subtrai-se ao produto inicial o que for preciso).

-Se obter por defeito apenas algo crucial ou mínimo, e as opções requerem mais informações do que as que se tem, então estas opções requerem que seja preciso fazer outras coisas, que podem (ou não) serem mais complexas e dolorosas que obter o produto inicial.

-Opções incompatíveis devem ser sinalizadas o mais cedo que se possa, para poupar tempo e recursos em realizar uma instância errada ou não desejável do programa.

-Opções repetidas devem ser sinalizadas, para não permitir que um processo leia e interprete indefinidamente a mesma opção (se estiver repetida mesmo muitas vezes nos argumentos de execução), quando esta só precisa de ser lida uma única vez.

-Opções não incompatíveis mas que possam comprometer o resultado final se o que forem suposto fazer for essencialmente alterado uma pela outra, devem ser executadas de forma a evitar este acontecimento, ou a produzi-lo de forma consistente. Por exemplo, neste script, as opções "c" (filtro de interfaces) e "p" (número máximo de interfaces a visualizar) podem produzir um resultado indesejável (e essencialmente errado) se for feito em primeiro lugar a limitação do número de interfaces, e depois a filtragem de interfaces do resultado da limitação. Se for feita primeiro a filtragem por nomes, este resultado indesejável, em princípio, não ocorre.
-Ex. Prático:

```
Interfaces    Bytes enviados:
coisa1_TQ_burst   1
coisa2_luz_elec   9
coisa3_Swain_eye  10
coisa_cena        13
mau_cena          14
cena2             0



1a)-[limitação -p 2]:
coisa1_TQ_burst   1
coisa2_luz_elec   9

1b)[filtro -c "cena"]:



2a)[filtro -c "cena"]:
coisa_cena        13
mau_cena          14
cena2             0

2b)[limitação -p 2]:
coisa_cena        13
mau_cena          14
```

Pensamentos finais e conclusão:

Chegar a esta versão finalizada do script netifstat.sh foi complicado, algumas nuances e caraterísticas inerentes à Bash (como a forma de manipular arrays, e perceber alguns pormenores raros e ultrapassar certos obstáculos especiais próprios a algumas ferramentas, em particular o awk) resultaram em dores de cabeça ligeiras e outras experiências subjetivas sensoriais desagradáveis. O meu raciocínio e maneira de pensar foram alterados várias vezes ao longo da (re)escrita deste script, fruto da solução ao problema não me ser imediata. O uso de flags para a validação e tratamento de opções, por exemplo, foi uma novidade para mim, e revelou-se ser uma técnica eficaz que pude utilizar no processamento de opções.

Na realização deste trabalho houve (para mim) um significativo aprofundamento de conhecimentos em Bash, quer em termos de familiarização com a sintaxe e com as várias e únicas ferramentas disponibilizadas, quer com a forma de pensar e abordar (e esperançosamente, resolver) problemas como este. Acho-me mais apto e sinto-me menos tolhido neste ambiente que até agora me era extraterrestre.

Bibliografia:

https://stackoverflow.com/questions/2961635/using-awk-to-print-all-columns-from-the-nth-to-the-last

https://stackoverflow.com/questions/14718035/how-to-loop-an-awk-command-on-every-column-of-a-table-and-output-to-a-single-out

https://linuxhint.com/for_loop_awk_command/

https://stackoverflow.com/questions/38855144/how-to-divide-with-awk

https://stackoverflow.com/questions/806906/how-do-i-test-if-a-variable-is-a-number-in-bash

https://stackoverflow.com/questions/9767644/test-if-string-has-non-whitespace-characters-in-bash