# Combination of Bash and Python in Development of Wrappers used for Automation of Finite Element Analysis

Marko Topalović[*], Snežana Vulović[*], Miroslav Živković[**], Milan Bojović[**]
[*] Institute of Information Technologies Kragujevac, University of Kragujevac, Kragujevac, Serbia
[**] Faculty of Engineering, University of Kragujevac, Kragujevac, Serbia

topalovic@kg.ac.rs, vsneza@kg.ac.rs, milan.bojovic@uni.kg.ac.rs, zile@kg.ac.rs

***Abstract*—This paper presents developing wrapper scripts for automating Finite Element Method (FEM) analysis on GNU/Linux servers. The purpose of these scripts is to edit data in ASCII files that are inputs for FEM solver and to call FEM solver which performs the analysis. Input files consist of geometry model, material parameters, loads, constraints, time step definitions and other data. After the long-lasting calculations, based on the stress results, material parameters in input files are updated and the analysis is restarted. This loop is repeated until the analysis predicts structure failure and for each pass safety factor is calculated. These scripts are also used to extract certain element groups, combine file sections and adjust output file for post-processing. Although Bash is very versatile when it comes to text manipulation it was necessary to augment it with Python programing language in order to achieve required functionality, primarily for fitting material parameters needed for next calculation. Repetitive, tedious work that an engineer needs to perform is greatly reduced, utilization of server time is improved, and this solution can be used for further development, for example, an inclusion of optimization, on which will focus in the future work.**

## I. Introduction

For scientific computing many researchers use some distribution of GNU/Linux due to its free and open source nature, low hardware requirements, robustness and stability [1],[2]. In our laboratory Finite Element Method (FEM) models [3] which consist of 2164759 elements and are 698.7 MB in size are analyzed on a workstation with Intel Xeon E5-2400 processor and 64 GB of ram, running Ubuntu server 18.04. During the analysis, stiffness matrix [4] and the rest of the FEM data may require up to 19 GB of ram. The single analysis job for this model on the workstation is completed in approximately 1 hour. This long time, along with a need to perform a series of analysis jobs, motivated us to develop a Bash script which will prepare input files, and run consecutive analysis jobs. We had to keep the algorithm for fitting material parameters [5],[6] outside of the FEM solver code, so that engineers can frequently edit computation of the material parameters. Since Bash is not suited for mathematical calculations [7], we had to insert Python code [8] into the Bash script which will do some of the math. In the following sections of the paper, we will discuss the key features of Bash and Python, how to combine them, what problems may occur, and how to solve them.

## II. Methodology

### A. Bash scripting fundamentals and drawbacks

Bash (Bourne Again Shell, named after Stephen Bourne, author of the Unix shell sh) is the default shell (Command Line Interpreter CLI) for GNU/Linux, and is typically run within a terminal window. Bash scripts are invoked by entering the whole path to the script in the terminal window, or if it is opened in the current directory, by typing ./ followed by a script name [9].

Every Bash script starts with (#!/bin/Bash), where the combination (#!) is called shebang and it is used to define which interpreter to run, while hash (#) alone is used to define that the rest of the line behind (#) is a comment [7].

Bash allows using variables and arrays which do not need to be defined by type [7]. To access value of a variable a dollar sign ($) is used, while setting of a variable value is done with equal (=), however, users (engineers) must be very careful because one of the greatest issues of Bash programming is the use of whitespaces (spaces or tabs). In Bash every line is a separate command, and every line is divided by spaces into words, the first word is the name of the command, and the rest of the words are the arguments [7]. For example, (alfat=7) is the correct setting, while (alfat = 7) would mean calling unknown command (alfat) with arguments (=) and (7).

Input of values from the keyboard is done with read command, while output is done with echo [9]. The use of quotation marks is optional, we will talk about quotation marks later, for now the simple example will suffice:

```
echo ’please input alfap variable’
read alfap
echo ”you entered $alfap”.
```

In Bash, every variable is considered an array [7]. Bracket notation ([]) is used to reference an index of an array, while accessing it requires usage of braces ({}):

```
alfa[1]=8
echo ${alfa[1]}.
```

Every line in Bash is a separate command, so to write complex commands across multiple lines, we must use a backslash (\) which tells the interpreter to disregard the next character (in this case newline), so several lines of script can be considered as a single command [7].

These complex commands are composed using pipe (|) operators [7], which take output form one command (on the left) and input it to the next command (on the right).

To send output to the file we must redirect standard output [7] using greater than operator (>). If the specified file does not exist, this redirection command will create it, if the file exists, the redirection will overwrite existing data with the generated output [7]. To append output to the existing data in a text file, we use (>>) operator. For reading data from a file we can use less than operator (<), or more powerful commands which we will discuss later.

One of the most useful features of Bash is string manipulation [7], for example, to find and replace part of a string slash (/) is used, and to replace all the occurrences of a part of the string, double slash (//) is used:

matparam=”8.15e-01  1.31e-03  4.34e-03”

matparam==${matparam//e-03/e-04}.

To make the same changes in a file, sed (which stands for stream editor) command [9] is used:

$sed 's/e-03/e-04/g' input.txt.

In the above line (s) and (g) are flags, (s) stands for substitution and (g) stands for global [9]. Since previous command would make changes to the whole file (maybe even some unwanted ones), in sed command we can specify which line of text do we want to edit, for example, to edit only 7th line we would use:

$sed '7 s/e-03/e-04/g' input.txt.

If we want to replace the entire 7th line, we need to use a combination of dot and asterisk metacharacters (.*)

$sed '7 s/.*/”8.15e-01 1.31e-04 4.34e-04”/' input.txt.

Although backslash (/) is the most common delimiter for sed, other characters can be used as delimiter as well, comma (,), colon (:), at (@), etc. Beside substitute flagged with (s), sed can print (/p), delete (/d) insert (-i) string patterns from a file [7]. Another sed option is (-e), which tells interpreter that the following string is an editing instruction. If only a single instruction is passed on to sed, the flag (-e) is optional.

The strong, single quotes (' ') keep the literal values of each character, while double quotes (” ”) evaluate the expression inside [7]. Single quotes cannot be placed within another single quotes, even if they are preceded by an escaping backslash (\), while double quotes can (with escaping backslash (\) of course).

We must also emphasize the use of backquotes (` `) which are used for command substitution. The same way $variable is replaced with variable value, `command` is replaced by the output from enclosed command [7]. Another way to substitute command is to use parenthesis $(command).

The next command that we will address now is cat (short for concatenate) which is used to create, view, join (concatenate) files [9]. To open a file, one needs to type:

$ cat full/file/path/filename,

or, if the terminal window is opened in the file directory:

$ cat filename.

To create a new file, we use greater than operator:

$ cat >newfile.

To copy the content of one file to another we use:

$ cat oldfile > newfile.

In the previous two examples of cat function, one may notice the difference in using whitespaces [7]. To merge files under a new name, we use cat command like this:

$ cat material.inp load.inp > analysis.inp,

and to append one file to another, we can use (>>). When using the cat command, one can use additional options, such as (-n) which displays line numbers when the cat is used to print file content in a terminal window, or (-e) which displays dollar sigh at the end of file and paragraphs [7]. Cat command displays the entire file, to view only a part of it, one can use cat and pipe file to head and tail commands, which by default display first (head) or last (tail) 10 lines of piped file [7]. To specify the exact number of lines (–n) option is used, so to display only the last line of the file we use:

cat input.csv | tail -n 1.

To save the entire last line of a file into a variable for further processing, we use following combination of previously described Bash commands:

varInline=`cat input.csv | tail -n 1`.

The next command which was crucial for our FEM analysis is awk [10], but this is not just a simple command, it’s a whole data-driven programming language with syntax similar to C developed in 1977 by Aho, Weinberger, and Kernighan, and it is used primarily for processing text organized by rows and columns and data extraction from it [10]. Awk has the following format:

awk options 'pattern {action}' inputfile > outputfile.

Input and output files are optional, awk command is often used within a block of other commands, using pipes (|) to connect with them. As for the facultative options, we used is -F"," that tells awk that field separator (delimiter) is the comma (,) instead of default white space and –v that assigns a value to a variable [10]. Using field separator awk turns the whole line into a word array, and every word is stored as $1, $2, $3 etc. which can be later accessed [10]. The pattern can be omitted, in which case every line is considered, it can be a simple string between two slashes, for example /materials/ in which case only lines with enclosed word are considered [10]. Action part in awk invocation most often means print, usually a specific word from a line [10]. For example, to read Young’s modulus E from previously read last line of input file, we use a combination of commands:

E=`echo $varInline | awk -F"," '{print $1}'`.

The above commands send the whole line previously stored in the variable $varInline using pipe (|) to awk command, which splits the line into words separated by a comma (,), and prints the first word. Finally, using backquotes (` `) for command substitution, the variable E (Young’s modulus) is set.

To find where in the input file a certain block of data is located (which line) we could use:

awk '/Time Functions/{print NR}' analysis.inp,

where (NR) is built-in variable that stands for Number of Records [10], but we opted instead for grep command [7], which is one of the most used shell commands. The simplest form of grep command is:

$ grep 'word' filename,

but what makes the grep so powerful are its options [7], to name just a few: (-i) which makes the search case-insensitive, (-R) recursive search of all files within the current directory and its subdirectories, (-c) which counts the lines which have a matching pattern (word), (-v) reverse search that displays all the lines without a matching pattern, (-w) for matching the whole word, and (-n) that we used to find the line number of a pattern:

```
grep -n "Time Functions" analysis.inp
```

Since previous line adds the line numbers in front of a matching line, and we only need line number, we can pipe this result into awk, use a colon (:) as a field separator, and take the number which is the first word:

```
| awk -F: '{print $1}'.
```

Previously we discussed input and output using read and echo commands, but when we need a full control over format of a variable, we must use the (printf) command [9] which uses the same syntax as C language for format specifications:

```
E=$(printf "%10s\n" "$E").
```

If we need to make E variable 10 spaces wide, the above line will do the trick. We must also point out that echo command outputs a line with trailing newline character [9], while printf does not (which could be an advantage or disadvantage, depending on a use case), so we had to add (\n) at the end of the format specification.

Loops in Bash are created following pattern:

```
while [condition]
do
        #commands
done.
```

In our wrapper we used loops for reading and writing physical parameters of many materials into arrays [7].

The simplest if statement [9] has the following form:

```
if [condition]
then
        #commands
fi.
```

As in the majority of programming languages, else and else if statements are optional [7]. Note that (fi) at the end is a reverse of (if). In order to check if the specified input file exist option [ -f file ] or [ -e file ] is used:

```
if [ -f input.csv ]
then
        #open and process input file
else
        echo input.csv doesn't exist
fi.
```

Some parts of the condition semantics are somewhat similar to the old FORTRAN and some to C [9], for example to check if the variable holding number of materials is equal to 2 we would use the following:

```
if [ $matnum -eq 2 ].
```

Similar to FORTRAN, (-ne) means not equal, (-lt) is less than, (-le) is less or equal, (-gt) is greater than, (-ge) is greater or equal. Boolean operations are similar to C i.e. and is denoted as (&&), while or is denoted as (||). Operators like (-eq) perform numerical comparisons, while (==), (!=),(<=), (>=), (<), (>) compares strings [9].

As in the most programming languages, if statements can be nested, and evaluated condition can be comprised of complex mathematical expression [9]. However, one must be very careful when calculating something in Bash, because Bash supports only integer arithmetic and not floating point, causing the results to be rounded to integer values [7]. This was the main motivation behind this paper, to summarize our experience with the topic, and to discuss the ways to avoid potential pitfalls.

The simplest way to perform arithmetic operations (if we only need integer values) is to use double parentheses ( $((expression)) ):

```
variablea=7
variableb=$(( $variablea + 3 )).
```

Another option is to use expr command, but in this case, users must pay attention on white spaces and special characters. For example, (expr 2 + 2) is correct, (expr 2+2) is not, if we use quotes expression is not evaluated, but printed as is, so (expr ”2 + 2”) is also wrong. Also, if we want to multiply two numbers, we must escape asterisk (*) with backslash (\) like this (expr 2 * 2) because in Bash asterisk (*) has a special meaning [7]. Command expr prints result in the terminal window, if we want to assign the result to another variable we must use let command. When it comes to white spaces, let command behave opposite to expr command ie. (let variablea=2+2) is correct, while (let variablea = 2 + 2) is not [7]. We can use white spaces in the let command if we put the quotes around (let ”variablea = 2 + 2”).

Floating point calculations can be performed in shell using bc (Bash Calculator). This implies that the arithmetic expression must be passed to bc [7], either by a pipe (|) as a string array enclosed by the double quotes, or by so called here-string (<<<), which we will be looking back at later, when we discuss Python:

```
variablec=`echo "variablea/variableb" | bc–l`
variablec=$(bc -l <<< "variablea/variableb").
```

Note that in the first line we used back quotes (` `) for command substitution, and in the second we used parenthesis $(command) which are interchangeable and a matter of personal preferences. But more importantly, in both cases we used flag (–l) which stands for mathlib, and tells bc to use predefined math routines [7]. This flag makes bc use the maximum number of digits in floating point calculation, or one can set this value using (scale) variable, but we strongly recommend using (-l) instead.

We used bc whenever we could, but unfortunately, bc does not support scientific (exponential or e) notation. The solution for this problem was to use Python [8], which recognizes scientific notation for calculations, i.e. to insert Python code snippets into our wrappers, which we will discuss in the next section.

## B. *Python fundamentals, advantages and drawbacks*

Python [11] is high-level, object-oriented interpreted programming language, created by Guido van Rossum in 1991.

The fact that Python is the interpreted programming language means that the program code is translated into a machine code on every run-time, which leads to slower execution in comparison to the compiled programming languages, but on the other hand, interpreted nature of the Python means its code can be accessed by anyone, in our case, engineers who develop FEM models [6], and who can also change the equations for calculating material parameters directly in our wrapper script.

In order to embed Python code into a Bash script we used so called Here document [11] which are similar to here-strings that we previously used to pass on a simple math formula to Bash Calculator. Here strings are preceded with (<<<) and contain one word (we already explained that in Bash every line is divided by spaces into words, so "variablea/variableb" is one word). Here documents are marked with (<<) followed by a delimiting identifier, which can be any word, typically its (END). After the initial delimiting identifier, beginning from the next line, we can write a series of commands, in several lines, and when we are done, in the last line, we finish Here document with a closing delimiting identifier (END). The content of the Here document is passed on as standard input to the previous command, in our case Python, which is used to invoke the appropriate interpreter [11].

To reduce material parameter (Gc) by some reduction factor (SRF) we use the following Python code snippet:

```
Gcr=`python <<END
result=$Gc/$SRF
print '%10.2E' % result
END`.
```

In the above snippet we see command substitution using back quotes (` `), and accessing value of a variable with a dollar sign ($), because the math expression is nested within a Bash script. Unlike Bash, Python does not use ($) for accessing variables [11]. We can also see that the final, floating point result is printed in the scientific (exponential or e) notation, just the way we wanted it, but could not accomplish neither in Bash nor using Bash Calculator (bc). Alternative invoking Python from Bash which does not involve Here documents is done using:

```
Gcr=$(python –c “
import sys
Gc_py=float(sys.argv[1])
SRF_py=float(sys.argv[2])
result_py= Gc_py / SRF_py
print '%10.2E' % result_py
“ $Gc $SRF).
```

This invoking leaves the standard input free for other uses within the Bash script, but it can cause problems due to the fact that the single quotes cannot be nested within another single quotes [7], and double quotes perform expression evaluation. Another advantage of the Here document approach is access to the variable with the dollar sign ($Gc), while in the alternative Python invocation with ( -c command) we had to pass variables ($Gc) ($SRF) as command line parameters, which are accessed in Python via its sys module [11], using (import sys) command. Since Python treats all arguments of the (sys.argv) as strings, we also need to cast them into floats. Arrays in Python are called lists [11] and their indexes start with 0, so (sys.argv[0]) correspond to the flag (-c) which tells interpreter that the next word (text enclosed by quotations) is the command, while (sys.argv[1]) takes variable ($Gc), and (sys.argv[2]) takes ($SRF).

Now that we showed the means to embed Python code into a Bash script, we will discuss its basic features [11].

Contrary to the most popular programing languages (like C++, C#, Java) which use curly bracers ({}) to group command into blocks, Python [12] uses white spaces i.e. spaces ( ), while tabs (        ) are getting defunct [12].

Conditionals [11] in Python (if, else and elif) can check the conditions expressed similarly to C++ (==, !=, <, >, >=, <=), and can be extended [11] using logical operators (and, or, not). Colon (:) sign is used to mark the end of the conditional expression [11].  For example:

```
if variablea > variableb :
   variablec = variableb
else :
   variablec = variablea
variabled = variablec.
```

Since code block in Python is delimited by the white spaces [11], in the previous example the last command (variabled = variablec) is not part of the (else) block, and it is executed in any case. Another example of Python difference from mainstream programming languages (like C++) are loops [11], for example:

```
for i in range(5):
   print(i).
```

The loop above would print number 0,1,2,3,4 but not 5, so the implicit condition [11] is less than (<).

In the next section we will show how developed wrapper was used to run a FEM solver used to analyze Djerdap 1 and Grancarevo dams.

## III. Results and Discussion

In this section we will demonstrate how bits and pieces of Bash and Python code that we previously explained are put together in a functional wrapper script.

Our input file has 12 material models for concrete, rock, soil, so we must use Bash while loop:

```
varInline=`cat DamInput.csv | tail -n 1`
count=1
numOfMat=12
countVar=0
while [ $count -le $ numOfMat ]
do
let countVar =$countVar +1
E[$count]=$(printf "%10s\n" `echo $varInline | awk\
 -v var="$ countVar " -F"," '{print $(var)}'`)
let countVar =$ countVar +1
nu[$count]=$(printf "%10s\n" `echo $varInline | awk\
-v var="$countVar " -F"," '{print $(var)}'`).
let count=count+1
done.
```

In the example above, (count) variable is the index of the material used for parameter array, and (countVar) is the index of the parameter within the material string. In the awk command [10], we had to cast (countVar) into variable (var) using (-v) option, otherwise it would not work. In order to put the whole line i.e. command into double column paper template, we used backslash (\).

After material parameters are read from DamInput.csv file, some remain unmodified, while others are updated [6] or used for calculation of another material parameter:

```
GamR=`python <<END
result=((1-$AALFFr)*$fcPrimr-$Nred)/$ftPrim
print '%10.2E' % result;
END`.
```

Now that we have all material parameters, we can write them into the FEM solver input file. using the while loop:

```
let firstLine[1]=$firstMatLine
let secLine[1]=$firstMatLine+2
let thirdLine[1]=$firstMatLine+4
count=1
while [ $count -le $numOfMat ]
do
echo "${firstLine[$count]}""s/.*/""${E[$count]}"\
"${nu[$count]}""${D_c[$count]}"\
"${a_t[$count]}""${D_t[$count]}""/" >> script.sed
echo"${secLine[$count]}""s/.*/""${indtem[$count]}"\
"${temp0[$count]}""${s0[$count]}""/" >> script.sed
echo "${thirdLine[$count]}""s/.*/"\
"${ALF_P[$count]}""${GamR[$count"/" >> script.sed
let count=count+1
let firstLine[$count]=${firstLine[$count-1]}+10
let secLine[$count]=${secLine[$count-1]}+10
let thirdLine[$count]=${thirdLine[$count-1]}+10
done
sed -f script.sed $templateDat > $pakDat.
```

Each block for the material parameters in solver input file is defined in 10 lines, therefore, at the end of the loop, line numbers are increased by 10. Within a material block, there are 4 lines of description, followed by lines that contain names of the variables, and under them lines that contain variable values (firstLine, secLine, thirdLine).

The most important thing that we need to point out, is the fact that we could not put sed commands [7] with index substitution in while loop, so we had to create a separate script file named (script.sed) which is invoked in sed command using (-f) flag [7], and contains all lines that need to be replaced and their appropriate content (material parameters). To create this script file, we used append (>>), while to copy modified template file into solver input file we used simple redirection (>).

Wrapper algorithm is based on the bisection method [5], but, instead of a function, we have the entire FEM analysis [6], which can be completed successfully, or it can fail, but we cannot evaluate the value of the FEM analysis. Material parameters are scaled or calculated using SRF. The goal is to find the maximum value of SRF for which analysis does not fail. Initial interval is between Xa=1 and Xb=10, i.e. safety factor is between these two values. This procedure is shown in the next figure.

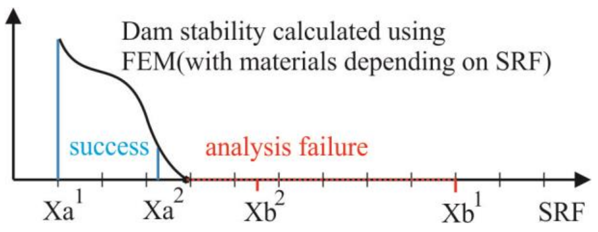

Figure 1. Bisection with FEM analysis

Wrapper can determine if the analysis was successful based on the existence of (control.sre) file using (-e) option in Bash if statement [7]. The next figure contains the entire wrapper algorithm.

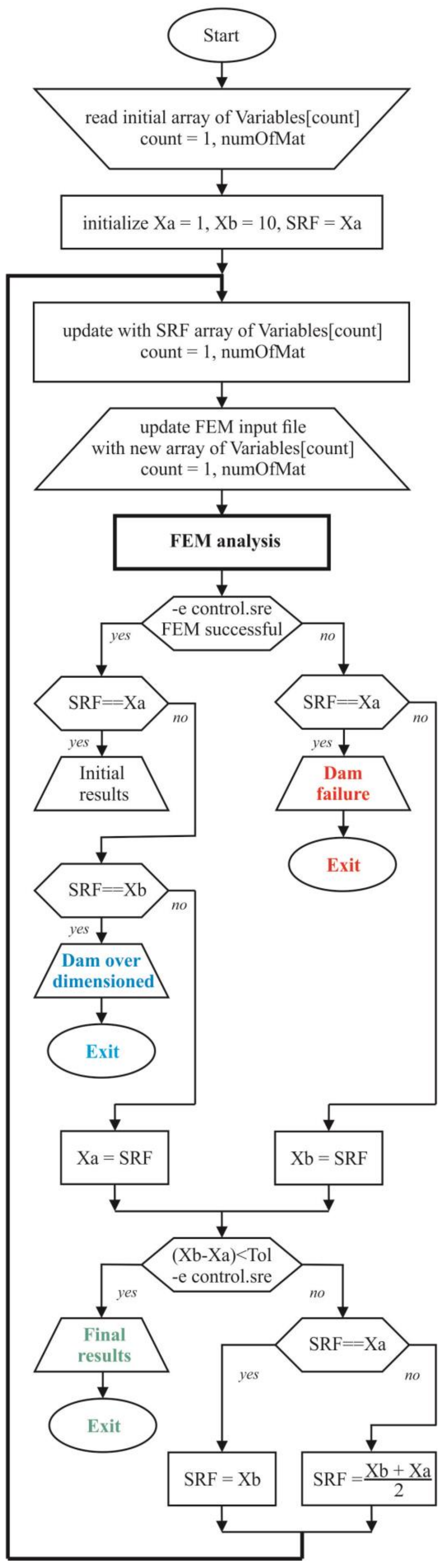

Figure 2. Wrapper algorithm

The main loop, in which material parameters (variables) are updated and FEM analysis performed, runs until the difference between Xb and Xa is less than defined tolerance and thus the safety factor SRF is determined. Another way for the loop to exit is in the case of flawed dam design, i.e. initial material parameters cause analysis failure (safety factor < 1) or over-dimensioned dam (safety factor > 10).

In the dam FEM analysis PAK Multiphysics solver was used [3]-[6], but presented wrapper could run, with modifications, with other FEM solvers as well. Every FEM solver has a specific ASCII input file, with a specific format for definition of nodes, elements, material parameters, etc. Therefore, lines for input of material parameters and update of the input file would need to change, while calculation logic would remain the same. Also, instead of (control.sre) file used by PAK, wrapper would need to look for another file depending on a solver.

This wrapper could be the base for developing simple parametric optimization solution [13], for example, instead of material parameters, shell thickness could be changed, until minimum weight is achieved. Advanced structural optimization such as shape or topology optimization [14] would require changes in node number and location, element number and defining nodes, and it would be significantly more difficult to implement within a Bash/Python script.

In the following figures, models for FEM analysis of Djerdap 1 (machine room) and Grancarevo dams are shown. Figures show great complexity of the model, with every color representing different material, and parameters of each of these materials need to be calibrated during the series of FEM analysis.

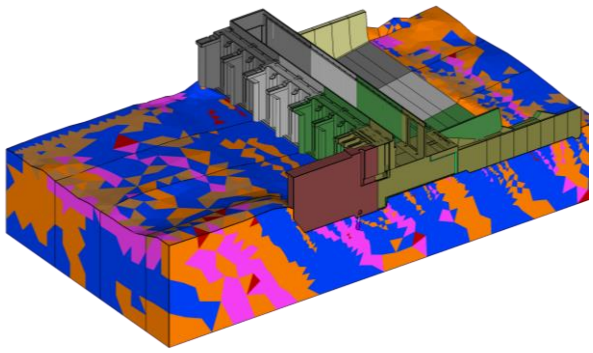

Figure 3. Djerdap 1 dam machine room

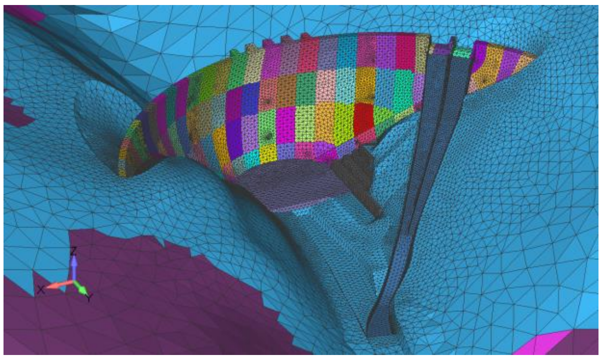

Figure 4. Arc dam Grancarevo

## IV. CONCLUSION

In this paper, we highlighted all the key features of the Bash script and Python language that we combined to accomplish our goal to develop a Bash script which will prepare input files, and run consecutive FEM analysis jobs. Advantages of Bash are its versatility and numerous powerful functions such as sed, cat, grep, awk, bc, but it has some drawbacks as well, such as floating point calculation and scientific formatting recognition. Python, on the other hand, handles scientific notation very well, but it has some disadvantages as well, for instance, it uses white spaces for the block identification, upper boundary for loops is not included, and invocation of Python code from Bash can create difficulties as well. Summarized experience in this paper could facilitate the work of other researchers, engineers and programmers faced with the similar hurdles, and will also be used for our future research in optimization of structures using FEM analysis.

## ACKNOWLEDGMENT

This research is supported by the Ministry of Education, Science and Technological Development, Republic of Serbia, Grant TR32036

## REFERENCES

[1] Y. Liguo, “An empirical study of software market share: Diversity and symbiotic relations,” *First Monday*, vol. 17, no. 8, 2012.

[2] S. H. Hong and L. Rezende, “Lock-in and unobserved preferences in server operating systems: A case of Linux vs. Windows,” Journal of Econometrics, vol. 167, no. 2. pp. 494-503, 2012.

[3] M. Kojić, R. Slavković, M. Živković and N. Grujović, *Metod konačnih elemenata I, Linearna analiza*, Kragujevac: Mašinski fakultet, Univerzitet u Kragujevcu, 1998.

[4] M. Živković, *Nelinearna Analiza Konstrukcija*, Kragujevac: Univerzitet u Kragujevcu, Mašinski fakultet, 2006.

[5] D. Rakić, M. Živković, and M. Bojović, “Implicit stress integration of the elasticplastic strain hardening model based on Mohr-Coulomb,” *6th International Congress of Serbian Society of Mechanics, Mountain Tara, Serbia*, 2017, 19-21 June, pp. S2b.1-S2b.11, ISBN 978-86-909973-6-7.

[6] D. Rakić, M. Živković, and V. Milovanović, “Stress integration of the Hoek-Brown material model using incremental plasticity theory,” *84th Annual Meeting of the International Association of Applied Mathematics and Mechanics-GAMM, Novi Sad, Serbia* 2013, 18-22 March, pp. 157-158, ISBN 10.1002/pamm.201310074.

[7] G. Speake, *Eleventh Hour Linux+, Chapter 5 - Using Bash*, Syngress, 2010.

[8] L. Prechelt, “Are Scripting Languages Any Good? A Validation of Perl, Python, Rexx, and Tcl against C, C++, and Java,” Advances in Computers, vol. 53, pp. 205-270, 2003.

[9] O. Pelz, *Fundamentals of Linux*, Packt Publishing, 2018.

[10] A. Robbins, *Effective awk Programming, 4th Edition, Universal Text Processing and Pattern Matching*, O'Reilly Media, 2015.

[11] D. Beazley and B. K. Jones, *Python Cookbook, 3rd Edition*, O'Reilly Media, 2013.

[12] G. van Rossum, B. Warsaw and N. Coghlan, *PEP 8 -- Style Guide for Python Code*, 2001

[13] M. Topalović, V. Milovanović, M. Blagojević, A. Dišić, D. Rakić and M. Živković, "Freight Wagon Mass Reduction using Parametric Optimization," in *VIII International Conference “Heavy Machinery-HM 2014”*, Zlatibor, 2014.

[14] N. Jovanović, M. Topalović, V. Milovanović, S. Vulović and M. Živković, "Topology Optimization Used to Reduce Weight of Four-Axle Bogie Freight Wagon," in *7th International Scientific and Expert Conference TEAM 2015, Technique, Education, Agriculture & Management*, Belgrade, 2015.