DEcalioc

User Information

Christopher Jelich Carolin Görnig Michael Rackl

5th November 2018

Contents

1 How to cite DEcalioc

To cite DEcalioc, please refer to source [7] of this document's bibliography:

Rackl, Michael and Hanley, Kevin J. (2017). A methodical calibration procedure for discrete element models. *Powder Technology*, 307, p. 73–83.

DOI: 10.1016/j.powtec.2016.11.048

2 Introduction

DECALIOC was developed with the purpose of simplifying the simultaneous calibration of multiple discrete element material model parameters and models. The code's name stands for "**D**iscrete **E**lement **ca**libration using **li**ggghts and **Oc**tave". As its name suggests, it is based on the discrete element code LIGGGHTS [3] and uses scripts, functions and packages written for GNU Octave [2] to control the calibration procedure.

2.1 Discrete element model calibration

Depending on the chosen contact model, a discrete element model features various input parameters which can be adjusted individually to model the behaviour of bulk materials. When particles are discretised as spheres instead of multi-sphere clumps and the particle size distribution is scaled up to decrease computational cost, these parameters need to be adjusted by means of a calibration process. Due to various interactions of the parameters, this calibration process can be very time consuming and challenging, especially when being performed by trial-and-error. Moreover, the fact that not all parameters are easily related to physical measurements of the bulk material adds further complexity to the calibration process. The interparticle friction coefficient, for instance, might be used to mimic non-spherical particle shapes, c.f. [10], and thus directly measured friction values become insignificant.

Facing a time consuming parameter calibration process at whom's end the calibrated parameters might not even be close to the optimal parameter set, the need of automated and robust calibration methods is apparent. Based on an automated workflow for discrete element material model calibration presented by Rackl et al. [8], DECALIOC is intended to meet this need. Model parameters are automatically calibrated based on the demands specified by the user. To do so, the programme uses Latin hypercube sampling (LHS), Kriging and nonlinear optimisation methods. The workflow of the calibration process and the employed methods are presented in the next section. For an in-depth understanding and a more detailed discussion, the user is referred to the corresponding publications [7, 8, 9].

2.2 Calibration procedure of DEcalioc

The core functions of DECALIOC have been tested and verified. The related studies and results were published in the literature [7, 8, 9].

The basic idea of the calibration procedure is to minimise the difference between experiment and simulation while keeping the required number of DEM simulations as small as possible. For this, DECALIOC creates a surrogate model of the underlying DEM model and performs a nonlinear optimisation based on this surrogate. Since evaluations of the surrogate model are computationally cheap when compared to evaluations of the DEM model, this process not only returns an optimal parameter set but also reduces the computational effort. In the following, each step of the workflow is discussed in separate sections.

Creating a Design of Experiments (DoE)

At the beginning, the user specifies n parameters of the DEM model and limits them by specifying their maximum and minimum values. These parameters form a n-dimensional parameter (hyper)space and will be calibrated in the following process. Based on this parameter space, a certain number of samples are generated by employing Latin hypercube sampling (LHS, [6]) which are then listed in an experimental plan. According to the experimental plan, DEM simulations are performed and their results are gathered as response data.

Creating Kriging models

With the response data and the corresponding experimental plan at hand, a black box model of the DEM model is created. Within DECALIOC, a Kriging model is used to approximate the response data in the solution space using polynomial functions and a certain correlation term. For further information about Kriging models the reader is referred to the literature and to reference [1]. Using Kriging models, an evaluation of a given parameter set only takes a fraction of CPU runtime compared to an evaluation of the DEM model. However, the accuracy of the response data is limited to a certain statistical confidence.

Nonlinear optimisation based on the Kriging models

Seeking the optimal set of parameters p_{opt}, multi-objective optimisation is applied to the Kriging models. Based on a user-specified number of target values, e.g. the bulk density, a vector-valued cost function $\boldsymbol{f}$ is defined which evaluates the quality of the response

data with regard to the user-specified target values. Exemplarily, a cost function could be defined as in Equation 2.1.

$$f_i(\boldsymbol{v}, p) = \frac{v_i - \bar{v}_i(p)}{v_i} \ , \quad i = 1, \ldots, n \tag{2.1}$$

Where n is the number of target values, $\boldsymbol{v}$ the vector containing the target values and $\bar{\boldsymbol{v}}(p)$ the vector containing the corresponding target values obtained by Kriging evaluations using the parameter set p.

With the cost function being a measurement of the quality of the response data, its minimum provides the optimal parameter choice p_{opt}. Within DECALIOC, this nonlinear problem is solved by a Levenberg-Marquardt algorithm [4, 5].

Verification of the found optimum using the DEM model

Due to the limited accuracy of the Kriging model, the obtained optimal parameter set p_{opt} is verified by repeating the optimisation procedure with the DEM models. The optimal parameter set identified by means of the Kriging models is expected to lie very close to the actual DEM models' optimum. Thus, this second optimisation run is going to be limited to few iterations, which may be required to slightly alter the parameters such that the differences between the DEM model results and the measurement data is truly optimally low.

3 Implementation of the calibration procedure

After the brief introduction to the calibration procedure in the last section, this section introduces the implementation of this procedure. In the following sections, the user interaction and workflow of the programme will be presented. Furthermore, interfaces of the main scripts will be explained, allowing users to replace the provided minimal working example with their own DEM model(s). As mentioned before, the calibration procedure is implemented in GNU Octave and makes use of several internal functions and external packages. Please see section 3.4 for prerequisites.

Since the whole calibration process and its implementation can be subdivided into four parts – LHS, Kriging model, optimisation based on the Kriging model and verification of the optimum using the DEM Model – we will stick to this pattern in the following.

3.1 User input

Several operations inside the workflow of the calibration process may be adapted by varying the input specified by the user. For this, the function `loadInput.m` is used. Herein, several variables which affect the workflow are declared.

In the first part, the input regarding the DEM models is specified. A cell structure *Input* is declared. It combines the model names, CPU-specific settings and Kriging-specific settings. Regarding the optimisation, the second part of `loadInput.m` declares the cell-structure *optim* in which information about the target values, breaking conditions and weighting factors are stored. In the third part, the model and design variables are specified. For this, the cell-structures *modelVars* and *assign* as well as the matrix *paramLims* are used. *modelVars* allows to store each model variable of the underlying DEM model to easily change the corresponding values defined in the input-file of the DEM models. In *assign* each design variable which will be calibrated during the run is specified. Based on the order of the design variables in *assign*, the parameter limits of the parameter space are specified in *paramLims*.

In the function `costFunction.m` the multi-objective cost function of the optimisation problem is specified. In every iteration, the cost function is called by the optimisation routine multiple times, with the current parameter set being passed in *params* as first

argument. The cost function then evaluates the underlying model, i.e. Kriging model or DEM model, and returns a vector of residuals to the optimisation routine. Listing 3.1 shows a code snippet of `costFunction.m` with the cost function being implemented in the way of Equation 4.1–4.2 which will be discussed in chapter 4.

Listing 3.1: `costFunction.m`

```
function residual = costFunction(params, evaluateFunc, varargin)
  % ...
  result = evaluateFunc(params, varargin);
  % ...
  % ...
  %*************************************************************
  %//   input - COST FUNCTION
  %*************************************************************
  % first part:
  residual = (result(:,1) - optim.targetVal) ./ optim.targetVal;
  % second part:
  residual(end+1) = optim.WRL * ...
       ((RLTS.max - modelVars.p) / (RLTS.max - RLTS.min));
endfunction
```

Since multi-objective optimisation is performed, the cost function must contain at least two residuals. Keep in mind that for each additional residual further function evaluations have to be performed due to the finite difference method used for gradient calculation; this will greatly affect the number of runs for the optimisation. Regarding the evaluation functions `evaluateKriging.m` and `evaluateDEM.m`, it shall be noted that both do generally not have to be modified.

3.2 Main function

The main function of DEcalioc is `DEcalioc.m`. In the beginning all necessary Octave packages are loaded into the workspace. Afterwards, the current working directory is saved as a global variable, the location of additional Octave scripts is added to the working path and the general folder structure is created. The remaining code generally follows the specified structure of the calibration process. Its workflow consists of the following five code blocks:

Input

In this block the user specified input is loaded into the workspace by calling `loadInput.m`.

Experimental plan and DEM simulation

Herein, the experimental plan is created based on LHS by calling `GenerateExperPlan.m`, which makes use of the `stk`-package. For each model specified, the experimental plan is divided into smaller stacks by `SplitToCPU.m` and `SplitExPlan.m` according to the user-specified parallelisation settings. Afterwards, these smaller stacks of the experimental plan are passed to `parcellfun.m` which calls `Start.m`. Inside `Start.m`, both the routine for starting the DEM simulation and the routine for evaluating the results of the DEM simulations are implemented. Since parcellfun is used, several DEM simulations are started in parallel without exceeding the maximum number of CPUs, specified in *Input.MaxCPU*.

At the end of this code block the results of each model are summed up and written into csv-files located in the **results** folder. One file is generated for each model and each target variable.

Kriging model

In this block, the experimental plan and the corresponding results are used to create the Kriging models. For every DEM model and each of its result variables one specific Kriging model is created. The creation of the models is done by calling `createKrigingModels.m` which makes use of several functions provided by the `stk`-package.

Optimisation using Kriging models

Based on the cost function specified in `costFunction.m`, an optimisation problem is stated and solved in this block. For solving the problem, `nonlin_residmin.m` of the `optim`-package is used. Regarding the necessary function evaluations, the evaluate function `evaluateKriging.m` is passed as function handle to `costFunction.m`, c.f. Listing 3.1. Since function evaluations of the Kriging models are computationally cheap, the maximum number of iterations and function evaluations in the optimisation process is limited at a relatively large number of about 1000 and 3000, respectively. The command line output during the optimisation process is specified in `userInt.m`.

Optimisation using DEM model

Similar to the previous block, this block states and solves an optimisation problem based on the user-specified cost function. However, this time the function evaluations require a DEM simulation. Therefore, `evaluateDEM.m` is passed to `costFunction.m` as function handle. Due to the high computational effort, the maximum number of iterations is limited by *optim.maxIter* and the maximum number of function evaluations is limited

by *optim.maxFunEvals*. An additional stopping criteria is specified in `userInt.m`: it is desired that the absolute residuals are below the user-specified tolerance *optim.tolRes*.

At the end of this block, the optimal set of parameters is written into the command line together with the vector of residuals and the results of the DEM simulation. DECALIOC ends and the calibration process is completed.

3.3 Interface specification

Only a few functions have to be adapted or replaced when introducing a new DEM-model. This section outlines where the interfaces are specified and such adaptations or replacements are necessary. Before going into detail, a general overview of the folder structure is given.

3.3.1 Overview of the folders

The general folder structure is shown in Figure 3.1. Blue folders are fixed components of the code, whereas yellow folders are generated by the user and grey folders are created during runtime.

DEcaLiOc is the main folder which includes all other folders and Octave scripts. Herein, the starting script `DEcalioc.m`, the script defining the cost function (`costFunction.m`) and the model and optimisation specific input script `loadInput.m` are stored. In general, only the latter has to be modified when changing model-specific and optimisation-specific details.

Every model has its own folder inside of **DEMmodels** which can be named arbitrarily. Herein, model specific information is stored, e.g. the input files for LIGGGHTS and a script which executes the simulation. Furthermore, a folder **OctaveFuns** is stored in every model folder providing model-specific evaluation functions. These functions are used to further process the results obtained from the DEM simulations. Additional back-end functions of DECALIOC are stored in the folder **infrastructure**.

The remaining folders are generated during runtime:

- **KrigingFuns** contains Octave scripts of the created Kriging models. Note that for each specified model and each specified target variable, one specific Kriging model is created.
- **optim** contains the results from all DEM simulations. These are located in separate folders inside the corresponding model's folder. The name of the models' folder is picked according to the naming in **DEMmodels**.

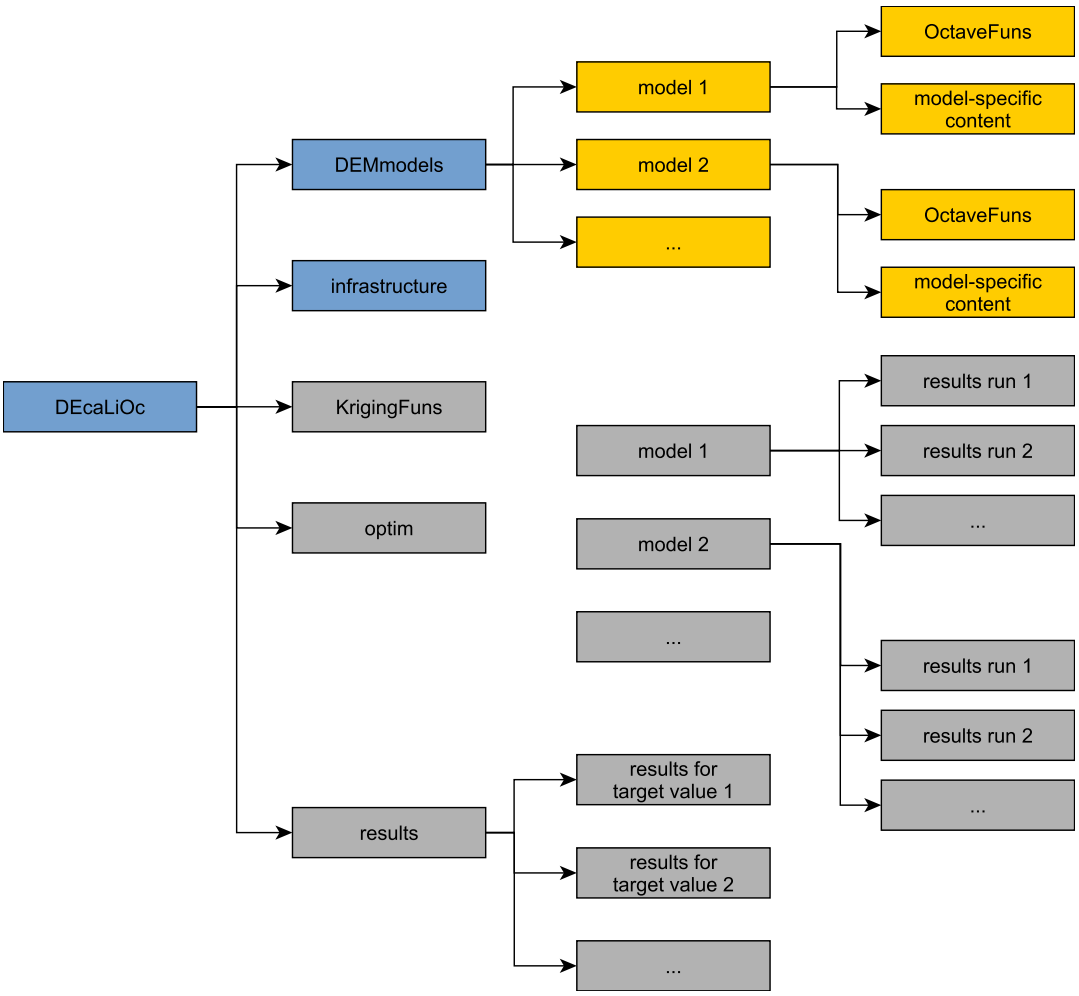

Figure 3.1: Folder structure of DECALIOC

– **results** contains simulation results which are generated when evaluating the LHS plan. The simulation results are stored in csv-files, whereby the results from each model and each target value are saved in separate files.

3.3.2 Overview of the functions

Having introduced the general folder structure, the focus is set on the provided Octave scripts. Figure 3.2 gives a brief overview of these scripts. Highlighted by a yellow background are scripts which require user input, highlighted by a light-blue background are scripts which are model-specific.

As aforementioned, the main function of DECALIOC is `DEcalioc.m` and several user specific settings can be changed by modifying the user input functions `loadInput.m` and `costFunction.m`. Besides the various back-end functions marked with a grey

background, Figure 3.2 introduces the model-specific functions which may be adapted when integrating your own DEM model. These are the bash-script `runscript` and the model-specific functions located in **DEMmodels/<modelname>/OctaveFuns/...**, e.g. `getResults.m`.

3.3.3 Interface specification for using your own DEM models

When integrating your own DEM models, modifications on the Octave functions introduced in Figure 3.2 are only necessary for those highlighted by a yellow or light-blue background. Those with a yellow background are input-specific functions and those with a light-blue background are model-specific functions. Since the user-specific functions have been addressed in section 3.1, only the model-specific functions are discussed in the following.

`runscript.sh` is a bash-script which starts the DEM simulation. This script is rather short and might look like the code-snippet in Listing 3.2. Therein, the discrete element code LIGGGHTS is called passing the input file `in.modelname` as argument. Optionally, LIGGGHTS can also be run in parallel through `mpirun`[1] specifying the number of processors used. Note that the number of processors has to be specified in `loadInput.m` to allow an efficient run of DEcalioc.

Listing 3.2: `runscript.sh`

```
1 liggghts < in.modelname
```

`getResults.m` is an Octave function which further processes the results of the DEM simulations. For this, several other model-specific Octave functions might be used. These are originally located in **DEMmodels/<modelname>/OctaveFuns/...**, together with `getResults.m` itself. To access results of the DEM simulations the corresponding data needs to be written into separate files, e.g. text files, by LIGGGHTS.

A detailed manual/instruction on how to integrate your own DEM models follows in chapter 4.

[1] Provided that `mpi` is installed on your machine.

3.4 Minimum Software Requirements

Software + version

- LIGGGHTS 3.6.0
- Octave 4.0.0

Octave packages

- struct-1.0.14
- stk-2.5.0
- parallel-3.1.1
- optim-1.5.2

The last test run was successfully completed under Ubuntu 18.04 LTS (64 bit) with LIGGGHTS 3.8.0, Octave 4.2.2, optim 1.5.2, parallel 3.1.1, stk 2.5.0 and struct 1.0.14, on 4$^{\text{th}}$ November 2018.

3.5 The first run of DEcalioc

Before including your own DEM models you should test your version of DECALIOC using the provided minimal working example. Please check the prerequisites in section 3.4 to guarantee an error-free run. Running `DEcalioc.m` in Octave starts DECALIOC. Note that the command `liggghts` should be executable from a non-interactive terminal to run the minimal example. Otherwise, please adapt the path to your `liggghts` executable in the file `runscript` in the `model` folder.

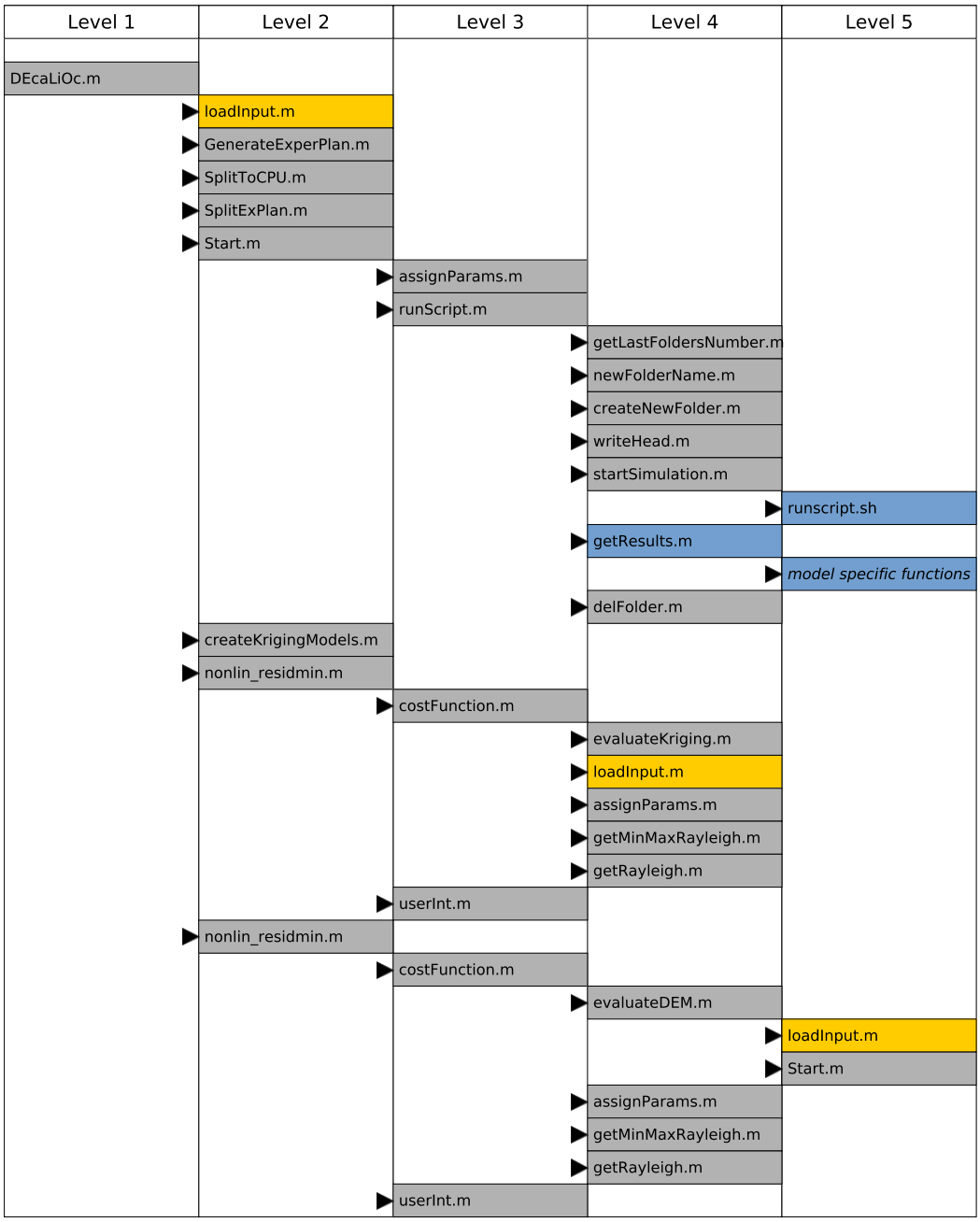

Figure 3.2: Functions of DECALIOC illustrated according to the calling hierarchy; grey: back-end function, yellow: requires user input, blue: DEM model-specific function.

4 How to integrate your own DEM models

This chapter provides an overview on how to integrate your own DEM models.

4.1 Copy your DEM models into the models folder

As first step you have to copy your DEM models into the models folder **DEMmodels**.

Each of your DEM models has to consist of at least three files. These are a LIGGGHTS input file `in.Model`, a head-file with the fixed name `data.head` and the bash script file `runscript`.

- `in.Model` is the input file used when calling LIGGGHTS. Its name as well as its structure is arbitrary and solely defined by yourself. However, there are two requirements: First, any results which are to be accessed through DEcalioc have to be written into separate files, e.g. text files. Second, your DEM model has to be parameterised such that the parameters which have to be calibrated are included in `in.Model`.
- `data.head` is the head file and defines all parameters used in `in.Model` together with their default values. This file is included into `in.Model` by stating "`include data.head`" at the beginning of `in.Model`. During the run of DEcalioc, this file will be updated with the actual parameter values of the current run.
- `runscript` is a bash script which starts LIGGGHTS and passes the input-file `in.Model` as argument (c.f. Listing 3.2).

Besides from these three files you might include an arbitrary number of additional files. Exemplarily, Figure 4.1 shows the folder of a DEM model called "Lift100".

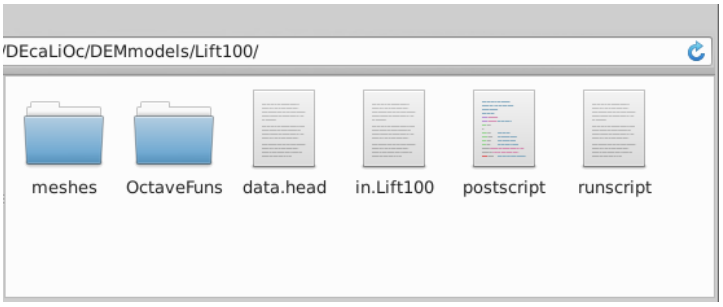

Figure 4.1: Exemplary folder of a DEM model called “Lift100”

4.2 Modify or create the evaluation functions

As mentioned before, accessing the results of the DEM simulations requires LIGGGHTS to write these results into separate files, e.g. text files. In order to be accessible by DECALIOC, the Octave script `getResults.m` is called. Since your DEM model most likely writes different output, you have to modify `getResults.m` to correctly read the written output. Furthermore, since `getResults.m` allows to further process the written results by calling model-specific functions, e.g. an image evaluation function, you might want to provide additional functions. An example is stated below.

Listing 4.1 shows a code snippet of `getResults.m`. Two arguments are passed to the function: *model*, the name of the DEM model and *folderName*, the name of the folder in which the simulation files are stored in. Together with the global variable *path*, these two arguments can be used to directly navigate to the simulation results. In line 5–6 this is shown by reading a scalar result from the csv-file `liggghtsOutput.csv`. Using the try-catch statement, DECALIOC is prevented from terminating in case of a missing or corrupted file.

Furthermore, line 12 shows an example of calling further evaluation functions. In this case images inside the results folder are processed by `imageEvaluation.m`. This result is then saved as a second result into *res{2}*. Keep in mind that these evaluation functions are model-specific, i.e. the `imageEvaluation.m` file of the DEM model specified in *model* is called.

As line 16 implies, the number of results is arbitrary. However, for each result a target value has to be specified in `loadInput.m`.

Listing 4.1: `getResults.m`

```
1  function res = getResults(model, folderName)
2    global path;
3    % first result
4    try
5      res{1} = csvread([path, 'optim/', model, '/', folderName, ...
6                        '/liggghtsOutput.csv']);
```

```
7   catch
8     res{1} = NaN;
9   end
10  % second result
11  try
12    res{2} = imageEvaluation(model, folderName);
13  catch
14    res{2} = NaN;
15  end
16  % ...
17 endfunction
```

The model-specific evaluate functions and `getResults.m` have to be stored into a folder called **OctaveFuns**, which has to be stored inside the corresponding model's folder; see the general folder structure shown in Figure 3.1 and an example of a folder with model-specific evaluation functions in Figure 4.2.

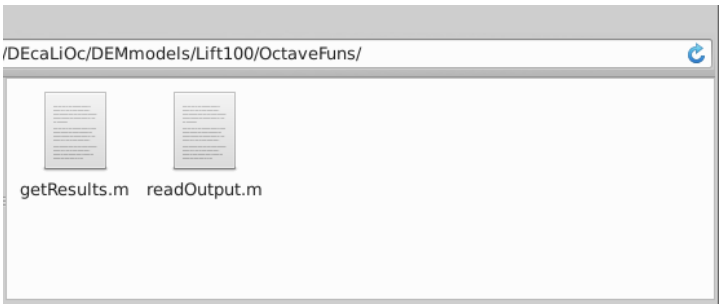

Figure 4.2: Exemplary folder with model-specific evaluation functions of a DEM model called "Lift100"

4.3 Modify the user-input and user-specified settings

After copying your DEM models into the models folder and providing evaluation functions, the user-specified input has to be modified. This is done in both `loadInput.m` and `costFunction.m`.

4.3.1 `loadInput.m`

The first part specifies model, CPU and Kriging specific input.

– *Input.model* stores the model names which correspond to the models' folder name.

- *Input.cpu* stores the number of CPUs used for a DEM run of the corresponding model. This applies if you want to run the DEM model with more than one CPU, i.e. using `mpirun`.
- *Input.maxCPU* stores the maximum number of CPUs to be used by DEcalioc.
- *Input.numOfSam* stores the number of samples to be generated by LHS

The second part covers optimisation settings.

- *optim.targetVal* stores the target values for each model. It is important, that the order of the target values matches the order of the results specified in `getResults.m`.
- *optim.tolRes* stores the tolerance used for the residual breaking condition, i.e. stop if every residual is below their corresponding tolerance in *optim.tolRes*.
- *optim.tolfun* stores the tolerance used for the breaking condition based on the overall change in cost function, i.e. stop if the sum of squares of the improvement in cost function is below *optim.tolfun*.
- *optim.maxIter* stores the maximum iteration count of the DEM-based optimisation.
- *optim.maxFunEvals* stores the maximum cost function evaluations allowed during the DEM-based optimisation.
- *optim.WRL* stores a weighting factor which scales the impact of the Rayleigh time step size on the residual, c.f. Listing 3.1. By setting *optim.WRL* to zero, the cost function does not take the Rayleigh time step into account.

The third part specifies the model variables, design variables and the parameter space.

- *modelVars* stores every model variable of the underlying DEM models which allows to easily change the corresponding values defined in the input-file of the DEM models.
- *assign* stores the design variables to be calibrated during the run. Keep in mind that the naming has to be chosen according to the naming of the corresponding parameters in the input-file of the DEM models.
- *paramLims* stores the limits of the parameter space. Each column holds the limit values of one of the design variables with the first row storing the minimal value and the second row storing the maximum value.

4.3.2 `costFunction.m`

Modifying `costFunction.m` is optional. On default, the computation of the cost function is implemented as shown in Equation 4.1–4.2.

$$f_i(\boldsymbol{v}, p) = \frac{v_i - \bar{v}_i(p)}{v_i} \ , \quad i = 1, \ldots, n \tag{4.1}$$

$$f_{n+1}(p) = w \cdot \frac{t_{\mathrm{R,max}} - \bar{t}_{\mathrm{R}}(p)}{t_{\mathrm{R,max}} - t_{\mathrm{R,max}}} \tag{4.2}$$

Herein n is the number of target values, vector $\boldsymbol{v}$ contains the target values and vector $\bar{\boldsymbol{v}}(p)$ contains the corresponding target values obtained from the Kriging evaluations using the parameter set p. $t_{R,min}$ and $t_{R,max}$ are the the minimum and maximum Rayleigh time step, respectively. $\bar{t}_{\mathrm{R}}$ is the Rayleigh time step of the current evaluation and w the weighting factor, which equals *optim.WRL*.

5 Minimal Working Example

Following is a short description of the minimal working example, which comes with DECALIOC. The minimal working example is a LIGGGHTS DEM model intended to showcase how models can be implemented within DECALIOC. The model's main component is an STL surface mesh of a cylindrical container. Its dimensions are shown in Figure 5.1; it is a simple open-top container with an orifice in the bottom.

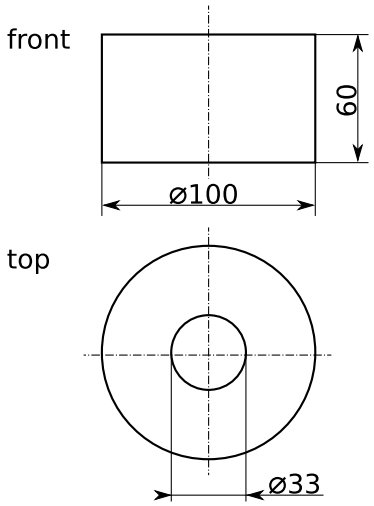

Figure 5.1: Sketch of the cylindrical container (not to scale).

The following steps are performed when the model is run.

1. Load the container into the simulation.
2. Define a surface (`ground`) to close the container's orifice on the bottom.
3. Fill the container with particles and le them settle.
4. Remove any particles above the desire filling height of 55 mm.
5. Get the remaining particles' mass and divide it by the filled volume in order to obtain the bulk density; write the result to the file `output_density`.
6. Remove the `ground` surface from step 2, such that the particles can leave the container through the orifice and wait for the next ten seconds.

7. Obtain the total mass of the particles which remained inside the container and write this result to the file `output_mass`.

The target values in `loadInput.m` are set with values of 400 kg/m^3 for the bulk density and 0.025 kg for the remaining mass. We do not claim that this setup is sensible in terms of bulk solid characterisation whatsoever. This model was built to demonstrate how DECALIOC can be applied.

With standard settings, the minimal working example runs on eight CPUs and took approximately **two hours to finish** in our test system.

Bibliography

[1] J. Bect, E. Vazquez, et al. STK: a Small (Matlab/Octave) Toolbox for Kriging. Release 2.3, 2014. URL `http://kriging.sourceforge.net`.

[2] J. W. Eaton, D. Bateman, S. Hauberg, and R. Wehbring. *GNU Octave version 4.0.0 manual: a high-level interactive language for numerical computations*. 2015. URL `http://www.gnu.org/software/octave/doc/interpreter`.

[3] C. Kloss, C. Goniva, A. Hager, S. Amberger, and S. Pirker. Models, algorithms and validation for opensource DEM and CFD-DEM. *Progress in Computational Fluid Dynamics*, 12(2/3):140–152, 2012.

[4] K. Levenberg. A method for the solution of certain non-linear problems in least squares. *Quarterly of applied mathematics*, 2(2):164–168, 1944.

[5] D. W. Marquardt. An algorithm for least-squares estimation of nonlinear parameters. *Journal of the society for Industrial and Applied Mathematics*, 11(2):431–441, 1963.

[6] M. D. McKay, R. J. Beckman, and W. J. Conover. A comparison of three methods for selecting values of input variables in the analysis of output from a computer code. *Technometrics*, 42(1):55–61, 2000.

[7] M. Rackl and K. J. Hanley. A methodical calibration procedure for discrete element models. *Powder Technology*, 307:73–83, 2017. doi: 10.1016/j.powtec.2016.11.048.

[8] M. Rackl, C. D. Görnig, K. J. Hanley, and W. A. Günthner. Efficient calibration of discrete element material model parameters using Latin hypercube sampling and Kriging. In M. Papadrakakis, V. Papadopoulos, G. Stefanou, and V. Plevris, editors, *Proceedings of ECCOMAS 2016 (VII European Congress on Computational Methods in Applied Sciences and Engineering)*, volume 2, pages 4061–4072, 2016. ISBN 978-618-82844-0-1.

[9] M. Rackl, K. J. Hanley, and W. A. Günthner. Verification of an automated work flow for discrete element material parameter calibration. In X. Li, Y. Feng, and G. Mustoe, editors, *Proceedings of the 7th International Conference on Discrete Element Methods*, Springer Proceedings in Physics, pages 201–208. Springer Science and Business Media and Springer, Singapore, 2016. ISBN 978-981-10-1926-5.

[10] C. M. Wensrich and A. Katterfeld. Rolling friction as a technique for modelling particle shape in DEM. *Powder Technology*, 217:409–417, 2012. doi: 10.1016/j.powtec.2011.10.057.