EjectorDesigner

Technical Documentation EjectorDesigner

Version 1.0.1

Knut Emil Ringstad

September 13, 2021

Contents

1 Introduction

The EjectorDesigner software is an OpenAccess software for ejector design and performance mapping. The software is based on automatic generation of large CFD databases using ANSYS Fluent []. The data is analysed using the Gaussian Process Regression code from the SciKit learn packages. Due to propriatary restrictions, the multiphase HEM model can not be shared. However, it can be replaced with minor alterations to the code. The scripts are written in Python [], using the following packages:

- Pandas - For structuring and analysing larger databases
- Numpy - For matrix and vector operations
- OS - For directory tree organizing
- Matplotlib - For plotting
- CoolProp [] - For refrigerant property evaluations
- SciKit Learn GPR [] - For machine learning predictor and other useful tools
- Threading - Used for running multiple instances of Fluent simultaneously

For details on the models or design tool, see the authors thesis and journal publication (open access):
citation
URL: Not yet published
citation
URL: Not yet published

2 Brief overview

The code is structured based on the creation of a *working case* (WR). This *working case* contains the settings that will be applied, the database, the CFD results, and CFD meshes. These are organized into their corresponding folders in each cases tree structure, see the the directory structure below.

To generate a new case, the *makeCase()* -function is available in the *main.py* script. The case settings for the case can then be set, discussed further in Section 4. When running the main program, the settings files are based on the files in the folder </*CurrentSettings*>. When a new case is loaded (*loadCase()*- function in *main.py*), the *working case*'s settings are put into the in the </*CurrentSettings*> folder.

After the settings have been specified, the *MakeDatabase()*- function can be run in *main.py* to generate a uncalculated database, ie. without any results. To fill the database with results, the *UpdateDatabase()*- function can be run in *main.py* to start the calculations. After the calculations are done, the database can be found in the *working case* tree structure under </*Results*>.

To analyse the database results, the pandas dataframes can be used directly or the data analysis tools can be used. The data analysis tools includes the GPR prediction tool. To use this copy the database into the </*DataAnalysis*>- folder and run the wanted analysis scripts.

```
EjectorGPR/
 Cases/
   your_case_name/
     Meshes_and_cases/
     Results/
       Database_and_results
     PreviousResults/
     Settings/
       settings_files
 CFDinterface/
   CFD_pre_and_post_code
 Sampling/
   sampling_code
 DatabaseInterface/
   database_interface_code
 CurrentSettings/
   settings_files
 main.py
 CFD_calculation_tools
-------------------------
 DataAnalysis/
```

3 User example

3.1 Generating a database

This example will illustrate how to generate a database with 5 ejector geometries features and N=200 samples (Lmix, α, Dmo, Dout, Dmix). The design point is set to: P_m = 85 [bar], T_m = 30[°C], Ps = 33 [bar], T_s = -2[°C], P_{lift} = 5 [bar]. The analysis of the database with the GPR tool is also presented. To start a case named 'Example_Case' is generated with the command:

```
#in main
MakeCase('Example_Case_Dmix_Lmix')
```

This generates the folder </*Example_Case*> in the folder </*Cases*>. In this folder, the following changes are made to the settings (see Section 4 for more information about the settings):

- The geometry parameters in *BaselineConditions* are changed to correspond with the ejector geometry presented in the Article [], restated in Table 1.
- The operating conditions in *BaselineConditions* are changed to match the design point.
- The settings in *SamplingSettings* are changed as shown below.
- If a different Fluent-case name is used than *EjectorSolver.cas* this needs to be specified in *CFDSettings*.
- Choose the number of cores you want to use for your CFD calculation in *CFDSettings*, according to your RAM and core count. These are chosen by defining the variables : <NumberParallel> and <NumberCoresPerParallel>, and will use NumberParallel$\times$NumberCoresPerParallel cores on your computer.

Loading this case (notice that the </*CurrentSettings*> folder is updated), the database can be generated by running:

```
#in main
LoadCase('Example_Case_Dmix_Lmix')
makeDatabase()
```

This produces the uncomputed pre-database, *TestPointMatrix.csv*, in the </*Results*> subfolder of the *working case*.

The results from this database can be calculated by running:

```
#in main
updateDatabase()
```

The program will start running and the computations will start on your local computer. If the program is stopped during calculation, it can easily be restarted by running this command again. As the program is running, the database file *Default_database.csv* (name chosen in settings), found in the *</Results>* subfolder, will be filled with data from the post processing of all completed calculations.

When all the calculations are completed, the *Default_database.csv* can be analysed, shown in the following sections.

```
#in SamplingSettings
samplingBool=true
NumberOfSamples = 200
samplingMode = 'LinearLatinHyperCube'
featureDict = {
"Pm":0,
"Ps":0,
"Po":0,
"hm":0,
"hs":0,
"LintletConst":0,
"DmotiveIn":0,
"DmotiveOut":1,
"Dthroat":0,
"alphaMotiveDiff":0,
"alphaMotiveConv":0,
"Lmch":0,
"alphaSuction":0,
"Lsuction":0,
"Dmix":1,
"Lmix":1,
"alphadiff":1,
"DdiffOut":1,
"Loutlet":0,
"Dsuc":0,
"ThicknessNozzle":0,
"ThicknessNozzleOuter":0,
}
```

```
#in SamplingSettings
SamplingMinDict = {
    "DmotiveOut":0.00141+0.00003,
    "Dmix":0.00141+0.00003,
    "Lmix":0.001,
    "alphadiff":0.1,
    "DdiffOut":1e-3,
}

SamplingMaxDict = {
    "DmotiveOut":20e-3,
    "Dmix":0.00141 * 10,
    "Lmix":0.004*50,
    "alphadiff":90,
    "DdiffOut":20e-3,
}
```

Table 1: Available features and their parameter sampling range for LHC-sampling algorithm.

Parameter	a_{min}	a_{max}
$D_{\text{m-out}}$, [mm]	D_{throat}	1.5 D_{throat}
L_{mix}, [mm]	0	50 D_{mix}
D_{mix}, [mm]	D_{throat}	10 D_{throat}
D_{diff}, [mm]	D_{mix}	10 D_{mix}
α_{d}, [$^\circ$]	1	90

3.2 Data analysis

To analyse the database, several tools are available. The database can be loaded into a Pandas dataframe to gather information about the simulations. For more information about the Pandas dataframes see the Pandas documentation. To analyse the data using GPR machine learning, the data (*Default_database.csv*) is copied into the </*DataAnalysis*> folder. This example shows how the GPR function is used for plotting heat maps of design variables. For more details on usage of the SciKit Learn GPR model and the SciKit learn toolset, see [**?**]. The primary options are set as:

```
# in GPR_function_design
trainBool=True
displayBool=True
optimizeBool=False
csvname='Default\_database.csv'
df = pd.read_csv(csvname)
features = ["DmotiveOut", "Dmix", "Lmix", "alphadiff", "DdiffOut"]
output = ["eff"]
seed = 2
```

The *trainBool* boolean variable is set to true the first time you run a model. It toggles wether the GPR model is trained using the *csvname* file as the database. It can be set to false if you do not want to retrain the GPR model, but use the previously stored GPR model.
The *trainBool* boolean variable toggles if you wish the heat maps to be plotted.
The *optimizeBool* boolean variable toggles if you wish the optimization using gradient descent to be used on the GPR response-surface mapping.
The *features* list must contain the names of the variables that you chose as features to be sampled in the database.
The *output* varible is the output you wish to predict using GPR. Here, the ejector efficiency, *"eff"*, is used, however it can be exchanged for entrainment ratio, *"ER"* other KPI's.

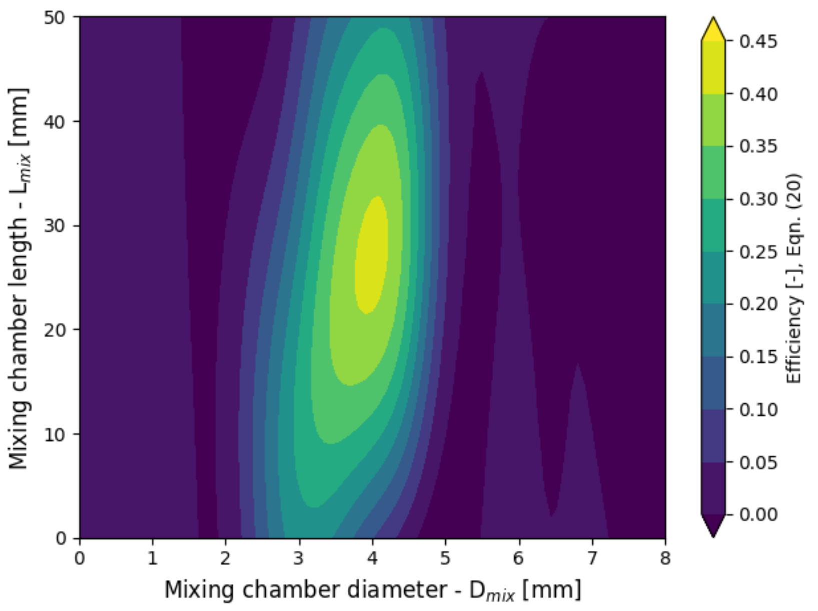

Figure 1: Output heat map from GPR model trained on the database.

The *seed* vaiable is used in randomization, and can be changed to see how different GPR realizations look like.

Running the program produces the following outputs:

```
Training GPR model
Model trained
Avg abs error: 0.031275220
Avg square error: 0.002739662
0.104**2 * RBF(length_scale=[4.01, 0.361, 1.54, 1.71, 300])
+ WhiteKernel(noise_level=0.000183)
```

Where the last line is the optimized kernel that was trained, and the output plot is shown as Figure 1.

4 Settings

The settings in the *working case* folder is organized into different files for different applications.

- The BaselineConditions settings are the settings where the geometry and operating condition variables are set. If a feature is sampled, these values are ignored.

- The CaseSettings file includes the names and locations of the folder locations for that *working case*. This is setup automatically if the *MakeCase()*- function is used.
- The CFDSettings file includes the numerical CFD calculation parameters (relaxation parameters, CFL numbers). In addition the calculation parallelization core counts are set here, see Section 3.1.
- The DatabaseSettings includes the names of the different variables and outputs in the database. Does not need to be changed for ejector calculations.
- The MeshingSettings includes the *ManualMesh* toggle boolean variables for manual meshing, where instead of the standard mesh names, the *MeshName* variable is used. The meshing algorithm includes different meshing strategies, set with *meshingMode*. If *meshingMode* is set to "Const_mix_number" a constant number of cells across the mixing section will be generated by the meshing script. This constant number of cells is set with the variable *Ncells_mix*.
- The SamplingSettings file has the settings for the database sampling. Database sampling is turned off or on with the *SamplingBool* variable. If the database is sampling, the number of samples used in the database is set by the variable *NumberOfSamples*. The sampling only applies to features specified for sampling. These features are chosen by setting the *featureDict* dictionary. If a feature is assosiated with a 1, then that feature is sampled. Otherwise, if it is a 0, then that feature is not sampled. The features are then sampled between a minimum value and a maximum value. These values are specified for each variable in the *SamplingMinDict* and *SamplingMaxDict* dictionaries. For now, only the "LinearLatinHyperCube" choice is availalbe for the *samplingMode*. The latin hypercube approach samples the feature space very efficiently, with low overlap.