Table of Contents

# Description of deliverable

After implementation and internal release of the prototype implementation of the software framework, the software has undergone testing before making a first public release.

Testing has been carried out on simple, generic examples and a baseline application for an expanding polyurethane foam which has been augmented with multi-scale capabilites using the MoDeNa software framework. The model includes a simple but realistic representation of the polymerization kinetics and accounts for the presence of a physical and chemical blowing agent.

The work has been carried out within task 5.5.

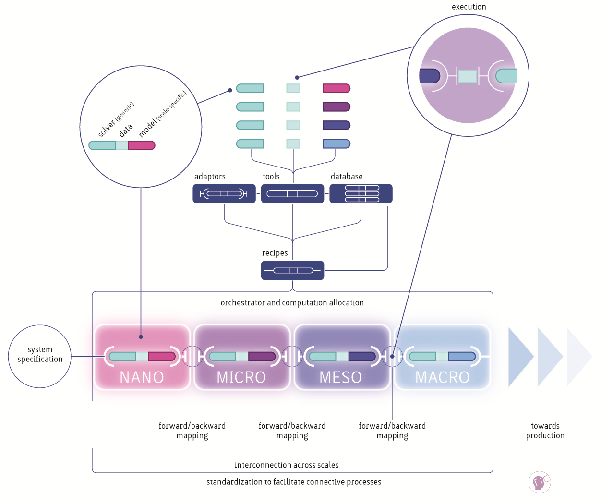
# Summary of contribution of involved partners

WIKKI carried out the implementation of the software framework and two tank example. NTNU contributed a documentation of the database structure and two tank example. POLITO contributed the 0D model and testing resuls. The deliverable forms the basis of the first release and will be used by used by POLITO, BASF, VCSHT, US and UNITS for the implementation of the recipies and adaptors for the demonstration cases.

# Introduction

The project aims at developing, demonstrating and assessing an easy-to-use multi-scale software framework application under an open-source licensing scheme that delivers models with feasible computational loads for process and product design of complex materials. The concept of is an interconnected multi-scale software framework. Four scales will be linked together by this framework namely the nano-, micro-, meso-, and macroscale (see Figure

). As application cases we consider polyurethane foams (PU), which are excellent examples of a large turnover product produced in a variety of qualities and of which the properties are the result of designing and controlling the material structure on all levels of scale, from the molecule to the final product.



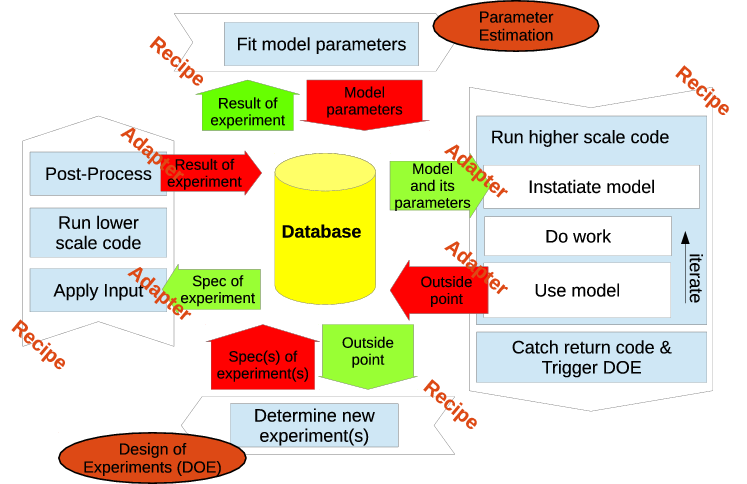
Multi-scale coupling requires the exchange of information between software instances developed for specific scales in a consistent way. In order to achieve this, generating consistent representations for models and data is necessary. The information exchange is governed by protocols and may occur in two ways, namely:

* “forward mapping” (passing information from the microscopic to the macroscopic scale in upward direction)
* “backward mapping” (passing information from the macroscopic to the microscopic scale in downward direction)

“Forward mapping” is relatively straightforward, while “backward mapping” inevitably requires iteration since changing the operating conditions at the fine level changes the feedback to the coarse level. “Backward mapping” can be realised by “two-way coupling” or by “fitting surrogate models”. The first approach usually requires exchange of large amounts of data during runtime that may be expensive either due to the complexity of the data exchange or the computational cost associated with executing the microscopic-scale simulation. In such cases, replacing the microscopic-scale simulation with a surrogate model presents the only viable alternative. This operation inherently constitutes a transfer of data across scales and is unique in that it focuses on this approach.

A typical operation sequence starts a macroscopic-scale simulation which instantiates one or more surrogate models. When the validity of a model is violated, a design of experiment operation is triggered. It creates inputs for a set of microscopic-scale simulations. When all experiments are finished, the parameter estimation component is invoked which updates the model parameters. Next, the macroscopic-scale simulation is restarted. It should be noted, that the software framework supports application and model dependencies across multiple scales.

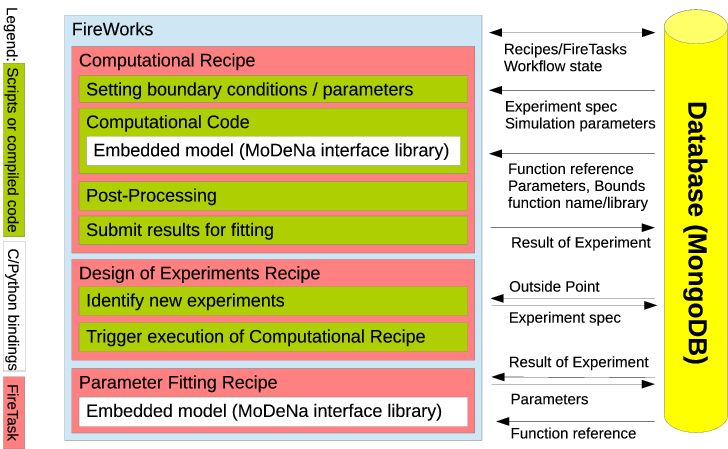
# Software Framework



The software framework handles the communication across scales through recipes and adapters ss shown in Figure

and

. Recipes perform simulations by executing applications (in-house codes or external software packages such as FOAM, Materials Studio, Predici) for a given set of inputs. Adapters handle the communication with the software framework. Both, recipes and adapters are application specific. Adapters exist as outgoing and incoming adapters. Outgoing adapters are relatively straight forward in that they perform a mapping operation (such as averaging) and communicate the results. The averaging process may have to be started and performed within the application (e.g. for time averaging). However, the results can usually be submitted in a separate process after the simulation is finished. Incoming adapters are more complicated since they usually require to embed surrogate models within the applications.



As shown in Figure

, the software framework consists of an orchestrator, a database and a interface library. The orchestrator is based on FireWorks F. project (2015) and constitutes the backbone of the software framework in that it schedules simulations as well as design of experiments and parameter estimation operations which make up the work-flow of the overall simulation. It is very much like a dynamic work-flow engine, in which the different applications are “orchestrated” to obtain information, analyse and pass it to the other operations. The NoSQL database MongoDB MongoDB project (2015) is used to store the state of the work-flow as well as the surrogate models together with associated data such as model parameters, data used for parameter estimation, and meta-data.

The interface library consists of two parts: A high-level python module providing access to the database as well as design of experiments and parameter estimation capabilities by building on MongoEngine MongoEngine project (2015) and R R. project (2015), respectively. The second part is a low-level library providing unified access to the surrogate models. This component is written in C to ensure interoperability across platforms and target applications while providing the computationally efficient model execution required by the applications. The library is loaded as a shared library by the macroscopic-scale applications or as a native python extension by the high-level python module ensuring that all components instantiate identical model implementations. Complex operations such as database access are referred back to the high-level python module using call-back mechanisms.

# Database concept

A database can be defined as an organized collection of data which enables us to handle large quantities of information by inputting, storing, retrieving and managing them. A relational database is defined as a database in which the data is organized based on the relational model of data. Codd (1970) The purpose of this model is to provide a declarative method for data and query specification. Relational databases mostly use structured query language (SQL) which can also be used as a term to describe a relational database.

Non-relational databases, dubbed as NoSQL (Not Only SQL), provide a mechanism for storage and retrieval of data which is modeled in a way different than in a relational database. Näsholm (2012) NoSQL databases are of interest in project since they allow database schemas which adapt to the users needs in a seemless manner. MongoDB is a auch a NoSQL document store database written in C++ and developed in an open-source project by the company 10gen Inc. The motivation behind its development is to close the gap between the fast and highly scalable key-value-stores and feature-rich traditional relational database management systems. Shermin (2013)

Some fundamental features of MongoDB are listed below to provide a more concrete insight and detailed information about the terminology and concepts:

* Document database: A record in MongoDB is a document which is a data structure composed of field and value, or key and value pairs. The values of fields can also include other documents, arrays, and arrays of documents. MongoDB documents are in BSON (’Binary JSON’) data format. Essentially, it is a binary form for representing objects or documents. Using document is advantageous since the documents correspond to native data types in many programming languages, embedded documents (subdocuments) and arrays reduce need for expensive joins, and dynamic schema supports fluent polymorphis. Project (2014) In the documents, the value of a field can be any of the BSON data types, including other documents, arrays, and arrays of documents. MongoDB stores all documents in collections. A collection is a group of related documents that have a set of shared common indexes. Collections are analogous to a table in relational databases.
* High performance: MongoDB provides high performance data persistence. In particular, support for embedded data models reduces I/O activity on database system, and indexes support faster queries and can include keys from embedded documents and arrays. Project (2014)
* High availability: To provide high availability, MongoDB’s replication facility, called replica sets, provide automatic failover and data redundancy. A replica set is a group of MongoDB servers that maintain the same data set, providing redundancy and increasing data availability. Project (2014)
* Automatic scaling: MongoDB provides horizontal scalability as part of its core functionality. Automatic sharding distributes data across a cluster of machines. Replica sets can provide eventually-consistent reads for low-latency high throughput deployment. Project (2014)

|  |  |
| --- | --- |
| SQL | MongoDB |
| Database | Database |
| Table | Collection |
| Row | Document or BSON Document |
| Column | Field |
| Index | Index |
| Table joins | Embedded documents and linking |
| Primary key (specify any unique column or column combinations as primary key) | Primary key (the primary key is automatically set to the \_id field in MongoDB) |
| Aggregation (e.g. by group) | Aggregation pipeline |

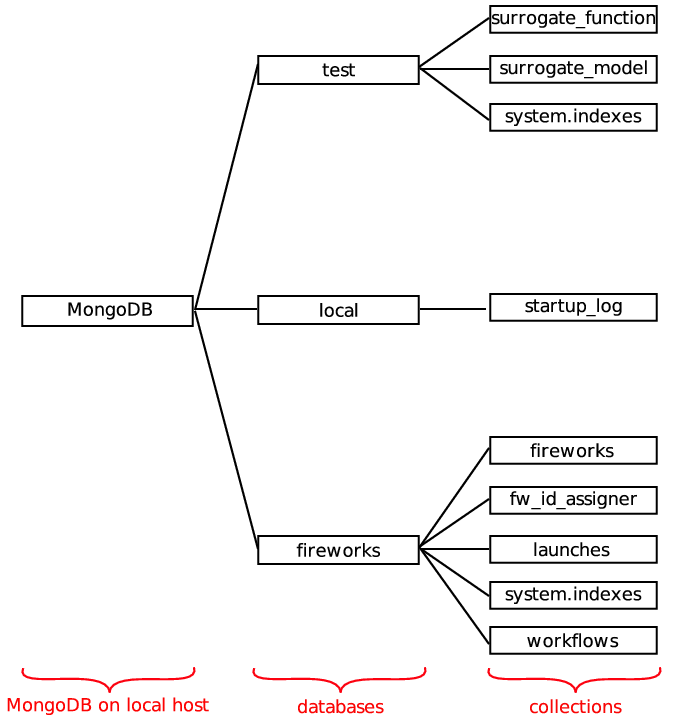
To provide insight in MongoDB in a comparative manner with more widely used relational database (SQL) approach, terminology and concepts of these two are summarized and shown in Table

.

MongoDB implements the basic functions of data storage, initially defined for SQL: (create, read, update, delete) CRUD. In MongoDB, these are represented as query which corresponds to read operations while data modification stands for create, update and delete operations. In MongoDB, read operation is a query which targets a specific collection of documents. Queries specify criteria, or conditions, that identify the documents that MongoDB returns to the clients. A query may include a projection that specifies the fields from the matching documents to return.

MongoDB queries exhibit the following behavior:

* All queries in MongoDB address a single collection.
* Queries can be modified to impose limits, skips, and sort orders.
* The order of documents returned by a query is not defined unless a sort() method is used.
* Operations that modify existing documents use the same query syntax as queries to select documents to update.
* In aggregation pipeline, the $ match pipeline stage provides access to MongoDB queries.

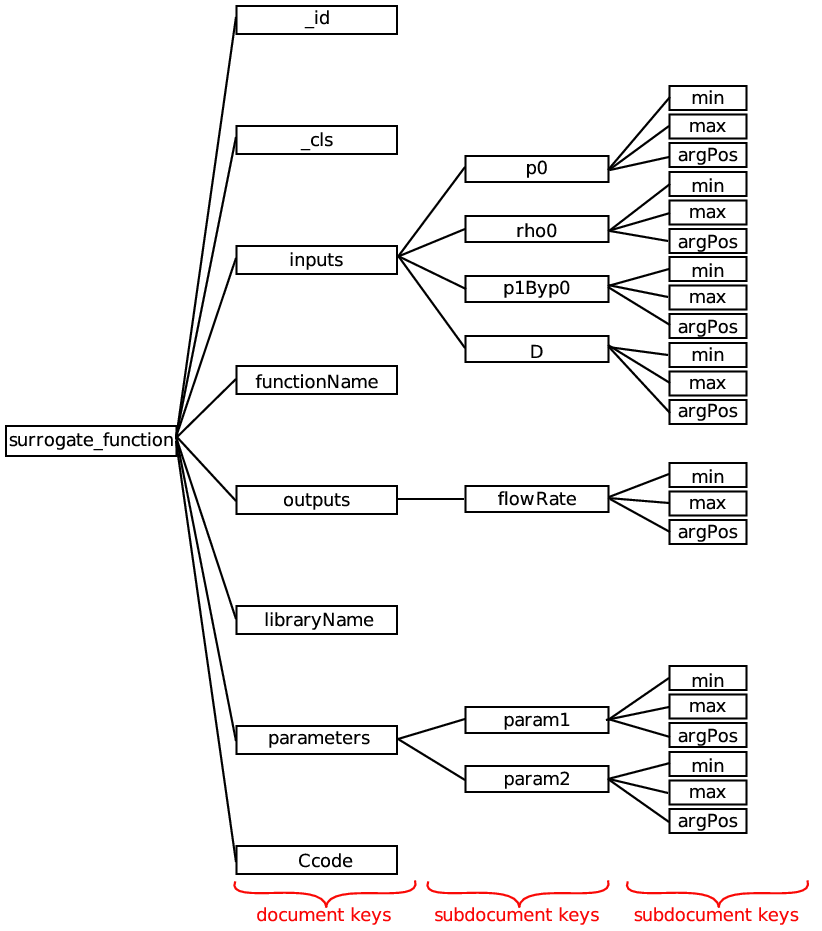


The project uses MongoDB for data storage. As shown in Figure

, the database has a crucial role as it resides in the core of the flow of information. In MongoDB, when there is data insertion, default database named as test is automatically populated unless specified otherwise. The test database in project contains information regarding *work-flow*, *work-flow state*, *models*, *experiments*, *metadata*, *simulation context*. Moreover, the software, Fireworks used in the project also employs MongoDB for *work-flow* (FireTasks) storage in a database called as fireworks. In addition to test and fireworks databases, there is another database (local) that is automatically created by MongoDB for the purpose of saving log information. Figure

shows an illustration of the MongoDB databases and corresponding collections. As can be seen in Figure

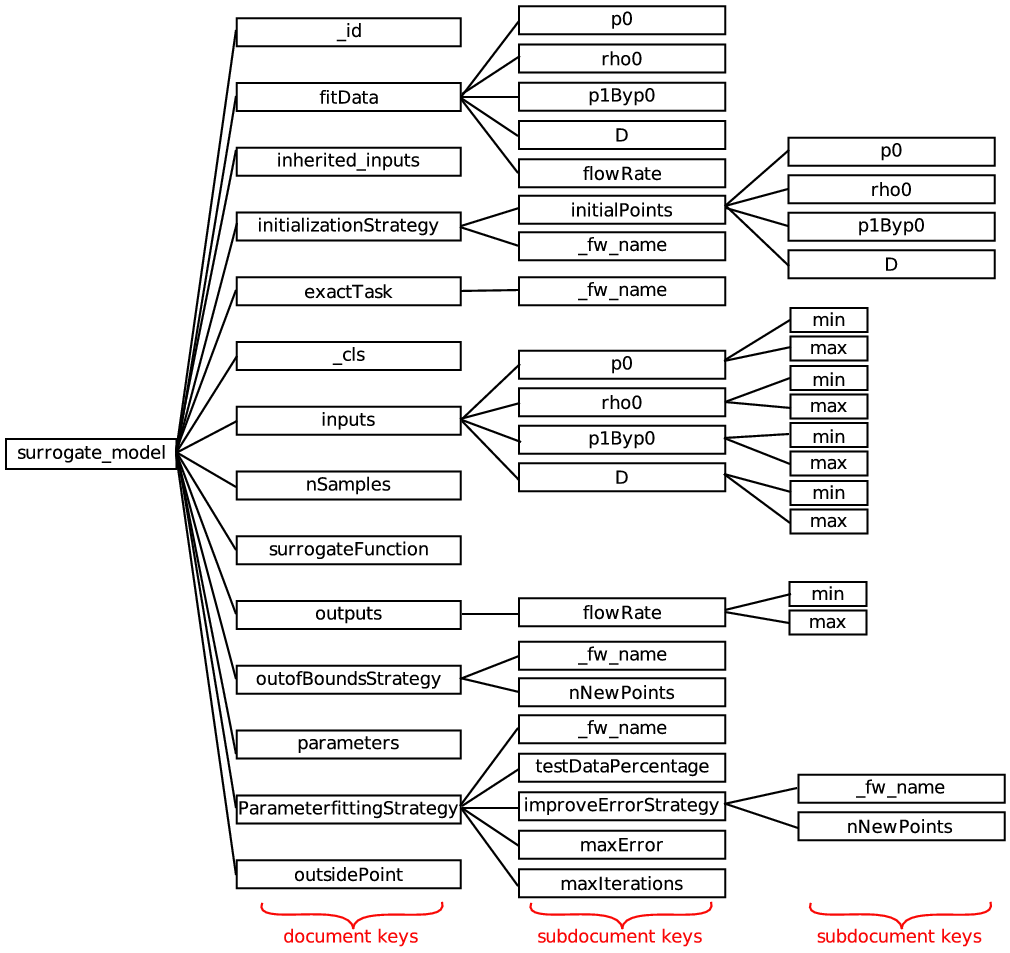
, there are three MongoDB databases (test, local, fireworks) on local host and their corresponding collections are also shown. In MongoDB, the system generates the name by concatenating the index key field and value with an underscore, e.g. cat\_1., if the user does not specify an index name Project (2014), and they are stored in a collection named as system.indexes which can be observed in the figure both for the test and fireworks databases. Each collection has at least one document which are not shown in the figure. Since the local database is generated by MongoDB and fireworks database structure is defined by the software itself, the main focus will be the test database which is developed within the scope of the project. For twoTank example in the project, data is stored in test database as well which has 3 collections: surrogate\_function, surrogate\_model and system.indexes. Each collection stores documents which consist of key and value pairs.



The collection, surrogate\_function contains one single document with the keys: id of the function \_id; class of the function, \_cls; the global bounds for the function, inputs which are p0, rho0, p1Byp0 and D, outputs which is flowRate and parameters which are param1 and param2 with global minimum value min, global maximum value max and argument position argPos; name of the function libraryName; compiled model in the library libraryName; C-code of the function for model execution and parameter estimation Ccode. All the mentioned keys and their value, and the hierarchy between them in terms of subdocuments constitute the database schema. The database schema is specified with corresponding keys and values in the initialization script except libraryName whose value is compiled in surrogatemodel script. To illustrate the data structure, data schema for surrogate\_function collection is drawn with keys and it is shown in Figure

. As can be seen in Figure

, surrogate\_function collection has one document which has 8 keys (\_id, \_cls, inputs, libraryName, outputs, libraryName, parameters, Ccode). Three of the keys store documents as values e.g. subdocuments (p0, rho0, p1Byp0, D, flowRate, param1, param2), and subdocuments also store documents (min, max and argPos). Thus, there are three levels of hierarchy in the data structure.



The collection, surrogate\_model contains also one single document with the keys: class of the model, \_cls; id of the model \_id; Firework spec exactTask which has a name of \_fw\_name; fitting data fitData which are p0, rho0, p1Byp0 and D; inherited inputs inherited\_inputs; initialization strategy initializationStrategy which has initialPoints of p0, rho0, p1Byp0 and D, and a name of \_fw\_name; the global bounds for the function, inputs which are p0, rho0, p1Byp0 and D with global minimum value min, global maximum value max; number of sample points nSamples; determination of points outside of the bounds outOfBoundsStrategy and corresponding number of points nNewPoints and a name of \_fw\_name; output of the calculation outputs which is flowRate with global minimum value min, global maximum value max; parameter fitting strategy ParameterFittingStrategy with maximum error limit of maxerror, sub-strategy for improvement improveErrorStrategy and corresponding number of points nNewPoints and a name of \_fw\_name, maximum number of iterations maxIterations, name of \_fw\_name and testDataPercentage; parameters, corresponding function of the model surrogateFunction and calculated numbers outside of bounds outsidePoint. To illustrate the data structure, data schema for surrogate\_model collection is drawn with keys and it is shown in Figure

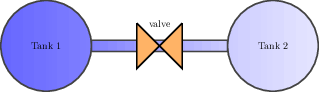
. As can be seen in Figure

, surrogate\_model collection has one document which has 14 keys (\_id, fitData, inherited\_inputs, initializationStrategy, exactTask, \_cls, inputs, nSamples, surrogateFunction, outputs, outOfBoundsStrategy, parameters, ParameterFittingStrategy, outsidePoint). As described in Figure

, keys store documents e.g. subdocuments. Thus, there are three levels of hierarchy in the data structure.

# Testing on two tank example

## Physical Description



“Two tanks” is a basic example designed to show the structure of the framework using a simple model of the discharge of air from one tank into another through a nozzle is considered as depepicted in Figure

. In order for the example to makes sense from a perspective, the flow through the nozzle must be considered as a complex problem which that has to be solved using 3D or a different detailed model. Consequently, the two tanks are macroscopic models and the rate of discharge of fluid through the nozzle represents a microscopic model. Moreover, the example also demonstrates a backward mapping problem if the range of the inputs is assumed to be unknown apriori.

In the example, the dynamic simulation, describing the contents of the two tanks, uses the interface library to embed a surrogate model describing the rate of discharge of fluid through the nozzle. In other words, the file twoTanksMacroscopicProblem.C is the macroscopic scale. The microscopic scale, a detailed model describing the dischage of fluid, is implemented in twoTanksFullProblem.C. The latter is never explicitly executed by the macroscopic code, but through the framework when the surrogate model is out of bounds.

The micro scale model, shown in equation

, was obtained from Chemieingenieurwesen (2010).

The variables of model, and describes the pressures in Tank 1 and Tank 2, respectively. is the diameter of the nozzle and is the density of the fluid (air). The model parameters shown below are respectively an empirical correlation and derived from thermodynamic assumptions.

The surrogate model approximating the detailed model in equation

is shown in equation (

).

The parameters, and , in the surrogate model are degrees of freedom used in order to approximate the detailed model in equation

.

The surrogate model is not explicitly defined in the file, twoTanksMacroscopicProblem.C, but the simulation can access the model from MongoDB using the syntax. Additionally, the file contains the initial conditions and dynamic equations describing the behaviour of the two tank system. When the simulation is startet it will therefore look for a surrogate model flowRate in the database and start solving the differential equations.

The initial conditions are defined as follows:

* Pressure in Tank 1 = 300000 Pa
* Pressure in Tank 2 = 10000 Pa
* Volume of Tank 1 = 0.1 L
* Volume of Tank 2 = 1 L
* Temperature of the system = 300 K (assumed constant)

When the surrogate model is used out of bounds, the framework steps inn and extends the range in which the surrogate can be used before the simulation is allowed to continue.

However, in order for the simulation to work the surrogate model has to be defined and put into the database. This procedure is described in Subsection

and in short consists of defining two items:

surrogate function  
the surrogate model computer code

surrogate model  
contains a surrogate function as well as strategies for events, e.g. initialization.

The nozzle surrogate model was initialised with the following values of the design variables:

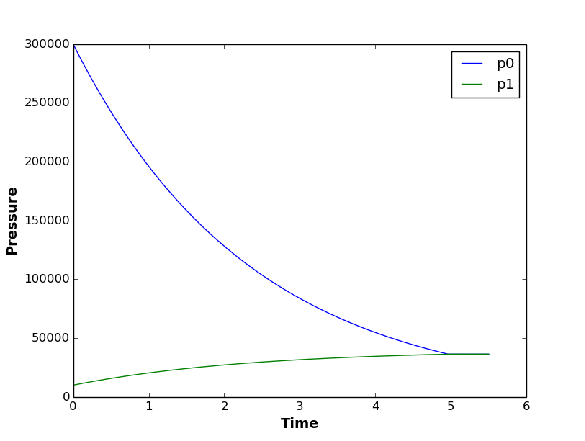
The initialisation strategy determines how the surrogate model parameters and are determined. When the initialisation is complete the surrogate model can be used by the macroscopic model.

The macroscopic simulation is started as a firetask and solves the differential equation using a timestep of seconds. It calls the surrogate model at every timestem in order to calculate how much mass is discharged through the nozzle. Afterwards, the ideal gas equation is used in order to update the pressure in the vessels due to the change in mass.

It is during the evaluation of the surrogate model that the surrogate model may be used outside the range in which the parameters where initially determined. This causes the simulation to throw an exception telling the framework that the design space of the surrogate model is extended. Consequently, the model is automatically updated by the using the strategies that was specified when the model was defined.

This the simulation is paused while the surrogate model is updated, and starts from the beginning when the new parameters are fitted. The final result in Figure

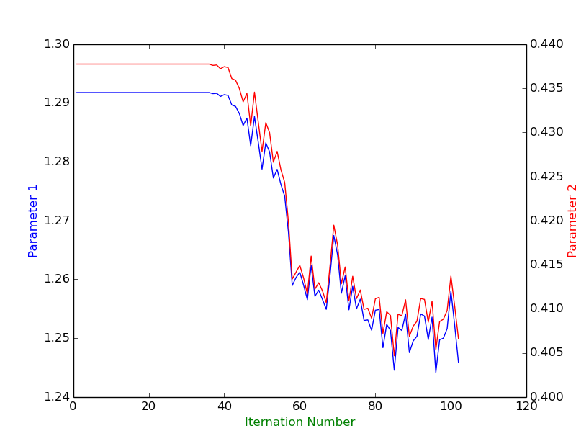
shows how the pressure of the tanks varies with respect to time. From the complete simulation results it is not possible to see that the macroscopic model used a surrogate model that was fitted and re-fitted.



In order to show that the parameters in equation

are changing dynamically throughout the simulation, Figure

shows how the value of the parameters change with respect to the iteration number. This is helpful especially in identifying parts of the design space where the surrogate model provides a good approximation.



The introduction gave an overview ot the example and how the framework facilitates the simulation to be performed in a multi-scale, -equivalent manner. The next sections aims at describing the code initiating and performing the parts of the simulation specific to the -framework.

## Initialization

The initModels script handles the initialization of the twoTank example. First, the relevant and from the interface library, as well as , are imported.

from modena import ForwardMappingModel, BackwardMappingModel, SurrogateModel, CFunction  
import modena.Strategy as Strategy  
import twoTank  
from fireworks import Firework, Workflow, LaunchPad  
from fireworks.core.rocket\_launcher import rapidfire

The code block below shows how surrogate functions contains raw C-code. The code is automatically compiled and prepared for use, the reason it can be used by the macroscopic models is that the mandatory fields *inputs*, *outputs* and *parameters* can be accessed by the macromodels through adaptors.

f = CFunction(  
 Ccode= '''  
#include "modena.h"  
#include "math.h"  
  
void two\_tank\_flowRate  
(  
 const double\* parameters,  
 const double\* inherited\_inputs,  
 const double\* inputs,  
 double \*outputs  
)  
{  
 const double D = inputs[0];  
 const double rho0 = inputs[1];  
 const double p0 = inputs[2];  
 const double p1 = p0\*inputs[3];  
  
 const double P0 = parameters[0];  
 const double P1 = parameters[1];  
  
 outputs[0] = M\_PI\*pow(D, 2.0)\*P1\*sqrt(P0\*rho0\*p0);  
}  
''',  
 # These are global bounds for the function  
 inputs={  
 'D': { 'min': 0, 'max': 9e99, 'argPos': 0 },  
 'rho0': { 'min': 0, 'max': 9e99, 'argPos': 1 },  
 'p0': { 'min': 0, 'max': 9e99, 'argPos': 2 },  
 'p1Byp0': { 'min': 0, 'max': 1.0, 'argPos': 3},  
 },  
 outputs={  
 'flowRate': { 'min': 9e99, 'max': -9e99, 'argPos': 0 },  
 },  
 parameters={  
 'param0': { 'min': 0.0, 'max': 10.0, 'argPos': 0 },  
 'param1': { 'min': 0.0, 'max': 10.0, 'argPos': 1 },  
 },  
)

The reason for the CFunction name is *syntactic sugar* making it possible to provide flexibility for the user to specify surrogate models in different programming languages. The inputs, outputs and parameter dictionaries - which contain the minimum, maximum and the argument positions of the corresponding design variables or parameters - are also stored in the database collection as key and value pairs. The creation of a surrogate function automatically updates the collection surrogate function in MongoDB, as explained in Section

.

When a surrogate function is saved in the database it is ready to be used in a surrogate model. The code block below shows how the backward mapping model in the two tanks example is defined in initModels.

m = BackwardMappingModel(  
 \_id= 'flowRate',  
 surrogateFunction= f,  
 exactTask= twoTank.FlowRateExactSim(),  
 substituteModels= [ ],  
 initialisationStrategy= Strategy.InitialPoints(  
 initialPoints=  
 {  
 'D': [0.01, 0.01, 0.01, 0.01],  
 'rho0': [3.4, 3.5, 3.4, 3.5],  
 'p0': [2.8e5, 3.2e5, 2.8e5, 3.2e5],  
 'p1Byp0': [0.03, 0.03, 0.04, 0.04],  
 },  
 ),  
 outOfBoundsStrategy= Strategy.ExtendSpaceStochasticSampling(  
 nNewPoints= 4  
 ),  
 parameterFittingStrategy= Strategy.NonLinFitWithErrorContol(  
 testDataPercentage= 0.2,  
 maxError= 0.05,  
 improveErrorStrategy= Strategy.StochasticSampling(  
 nNewPoints= 2  
 ),  
 maxIterations= 5 # Currently not used  
 ),  
)

An instance of the class is created with: the model \_id, the base surrogate function, described previously, from twoTank.py, which calls the exactTask of the twoTank example, and the various strategies. It is the surrogate model that is referenced to by the simulation in twoTanksMacroscopicProblem.C. The model is found in the database by looking up the \_id in the surrogate models collection described in Section

. Consequently, if the model has not been instantiated the simulation fails. Moreover, the framework will throw an error if the mandatory strategies, e.g. outOfBoundsStrategy and the ParameterFittingStrategy, have not been set. All the strategies are classes defined in the source file *Strategies.py*.

# set up the LaunchPad and reset it  
launchpad = LaunchPad()  
launchpad.reset('', require\_password=False)  
  
initFws = []  
for m in SurrogateModel.get\_instances():  
 initFws.append(m.initialisationFw())  
  
# store workflow and launch it locally  
launchpad.add\_wf(Workflow(initFws, {}, name="initialisation"))  
rapidfire(launchpad)

Afterwards, the surrogate models are ready to be used and the simulation can be started.

## twoTank.py

The twoTank.py file contains the two FireTasks that start the microscopic and the macroscopic codes. This file explicitly defines FireTask, this was not done for the surrogate models because they already are defined to be firetasks, but it is hidden from the user for convenience reasons. The result is that the imported modules in twoTank.py are different to those in initModels.

import os  
import modena  
from fireworks.user\_objects.firetasks.script\_task import FireTaskBase, ScriptTask  
from fireworks import Firework, Workflow, FWAction  
from fireworks.utilities.fw\_utilities import explicit\_serialize  
from blessings import Terminal

The necessary imports are the interface library, relevant objects from and a python package providing colored terminal output.

The file defines two firetasks, the first executing the microscopic model and the second executing the macroscopic, both of which are compiled C files.

@explicit\_serialize  
class FlowRateExactSim(FireTaskBase):  
 """  
 A FireTask that starts a microscopic code and updates the database.  
 """  
  
 def run\_task(self, fw\_spec):  
 print(  
 term.yellow  
 + "Performing exact simulation (microscopic code recipe)"  
 + term.normal  
 )  
  
 D = self['point']['D']  
 rho0 = self['point']['rho0']  
 p0 = self['point']['p0']  
 p1Byp0 = self['point']['p1Byp0']  
  
 # Write input  
 f = open('in.txt', 'w')  
 f.write('%g\n%g\n%g\n%g\n' % (D, rho0, p0, p1Byp0))  
 f.close()  
  
 # Execute the application  
 # In this simple example, this call stands for a complex microscopic  
 # code - such as full 3D CFD simulation.  
 # Source code in src/flowRateExact.C  
 os.system('../src/flowRateExact')  
  
 # Analyse output  
 f = open('out.txt', 'r')  
 self['point']['flowRate'] = float(f.readline())  
 f.close()  
  
 return FWAction(mod\_spec=[{'\_push': self['point']}])

The second FireTask starts the macroscopic code at the following command:  
However, unlike the microscopic firetask this can end in two ways:  
or

@explicit\_serialize  
class ModenaBackwardMappingTask(ScriptTask):  
 """  
 A FireTask that starts a macroscopic code and catches its return code.  
 @author: Henrik Rusche  
 """  
 required\_params = ['script']  
  
 def run\_task(self, fw\_spec):  
 print(  
 term.yellow  
 + "Performing backward mapping simulation (macroscopic code recipe)"  
 + term.normal  
 )  
  
 self['defuse\_bad\_rc'] = True  
  
 # Execute the macroscopic code by calling function in base class  
 ret = super(ModenaBackwardMappingTask, self).run\_task(fw\_spec)  
  
 # Analyse return code  
  
 print('return code = %i' % ret.stored\_data['returncode'])  
 if ret.stored\_data['returncode'] > 199:  
 print term.cyan + "Performing Design of Experiments" + term.normal  
 model = modena.SurrogateModel.loadFailing()  
  
 return model.outOfBoundsFwAction(  
 model,  
 self,  
 outsidePoint= model.outsidePoint  
 )  
  
 else:  
 print('We are done')  
 return ret

The reason why the FireTask has two possible outcomes is because it executes the twoTanksMacroscopicProblem code. The two outcomes represents starting design of experiments and parameter fitting, and terminating the entire workflow when the simulation is complete.

Simular to the exact simulation mentioned in the previous code bloc, the backward mapping FireTask is a ScriptTask, which helps running non-Python scripts through the command line. The required parameter for this class is script. In the twoTank example, the script is an executable, i.e. compiled C-code, representing the macroscopic twoTanksMacroscopicProblem.

However, the workflow itself is initiated in the workflow file shown below.

# set up the LaunchPad and reset it  
launchpad = LaunchPad()  
launchpad.reset('', require\_password=False)  
  
# create the individual FireWorks and Workflow  
# Source code in src/twoTanksMacroscopicProblem.C  
fw1 = Firework(  
 ModenaBackwardMappingTask(  
 script='../src/twoTanksMacroscopicProblem'  
 )  
)  
wf = Workflow([fw1], {}, name="simulation")  
  
# store workflow and launch it locally  
launchpad.add\_wf(wf)  
rapidfire(launchpad)

The LaunchPad is reset so that it can be loaded with the Fireworks. A Firework is created using the class through the macroscopic script. A Workflow is then created using this Firework, added to the and executed.

The return code of this macroscopic script is caught in the class. The return code indicates whether or not there is an outsidePoint. If there is, it returns the outOfBoundsFwAction, meaning that the Design of Experiments is performed. If not, the simulation ends.

# Testing on expanding polyurethane foam application

In this section, the work on integrating the sub-models, recipes and adaptors originating from WP1-3 is presented. The target application is an expanding polyurethane foam. A novel modelling approach, based on the solution of the population balance equation (), is used and applied for the first time to the simulation of expanding polyurethane foams. The model will be validated against experiments from the literature in WP 6.

Polyurethane foams are widely used materials, employed in many different applications, and almost always manufactured into useful objects via direct injection expanded foam molding Woods (1990; Oertel and Abele 1994; Klempner and Frisch 1991). The final characteristics (i.e. thermal and mechanical properties) of the manufactured objects highly depend on the final bubble (or cell) size distribution () of the foam Gibson and Ashby (1999). This is determined by the initial chemical recipe (i.e. isocyanate, polyol and blowing agents structure and concentration, catalyst and surfactant type and concentration), the injection dynamics and the flow history of the foam into the mold. However, the relationship between the process conditions and the final foam characteristics is highly empirical and the development of mathematical models is identified as the most efficient way to establish a more reliable and quantitative approach to the problem Niyogi, Kumar, and Gandhi (2014; Beverte 2014; Shen et al. 2014; Zhao et al. 2013; Mosanenzadeh et al. 2013; Zhou, Li, and Huang 2006; Tesser et al. 2004; Lo et al. 1994). Our objective is to develop a three-dimensional model, based on computational fluid dynamics (CFD), capable of simulating the mold filling process. The model also incorporates a population balance equation () that describes the evolution of the due to bubble growth, caused in turn by physical and chemical blowing agents, and bubble coalescence, caused by shear-induced bubble collisions. The combined use of and CFD was demonstrated to be a very useful tool for the simulation of many different multi-phase systems Marchisio and Fox (2013) and seems a very promising approach also for polyurethane foams. Before developing the full -CFD model and applying it to the three-dimensional simulation of real industrial-scale foaming processes, the model is firstly formulated, implemented in a simpler zero-dimensional framework and partially validated with experiments taken from the literature Baser and Khakhar (1994a; Baser and Khakhar 1994b; Geier, Winkler, and Piesche 2009). The model accounts for the progress of the polymerisation process, between polyol and isocyanate, by tracking the evolution of the gelling reaction. The presence of a physical blowing agent is also accounted for, together with the presence of a chemical blowing agent (i.e. water), and their effect on the is accounted for by solving the with the efficient quadrature method of moments () Marchisio and Fox (2013). The remainder of the paper is as follows: after a short description of the model equations, the operating conditions and investigated test cases are discussed. Then, results are presented and some significant conclusions are drawn.

## Governing equations

For clarity, this section starts with a list of the main simplification assumptions employed in the formulation of the baseline model. It must be noted that since this work tackles a mathematically and physically complex problem with limited information from the literature, some of the following simplifications are required. This baseline model treats the foam as perfectly mixed with a zero-dimensional “lumped” model. The model assumes that the foaming process starts after mixing of the components into a perfectly homogeneous mixture. It is also assumed that the only active bubble formation mechanism is mixing of the components that entraps some air in the mixture. Moreover, the effect of nucleation and spinodal decomposition is assumed to be negligible. The other important simplification is that the complex polymerisation reactions, often described with articulate schemes involving many species and tracking also the polymer weight distribution, is described here with the very simple kinetics scheme detailed below. Only two global reactions are considered. The reaction between polyol and isocyanate to form urethane bonds, responsible for linking the monomers, is called the gelling reaction:

The second global reaction, accounted for in this work, is the blowing reaction. This is the reaction between water and isocyanate producing carbon dioxide and forming urea bond. The reaction occurs in two steps with the thermally unstable, intermediate carbamic acid. The decomposition of carbamic acid produces carbon dioxide and amine. The amine group consequently reacts with another isocyanate molecule to form an urea bond. The global reaction reads as follows:

When the carbon dioxide produced reaches its solubility limit (often assumed constant with respect to temperature and pressure due to lack of data in the literature) gas bubbles already present in the mixture begin to grow. The gelling reaction, reported in equation

, obeys a first-order kinetic with respect to the concentrations of polyol and isocyanate, resulting in the following rate Winkler (2009; Baser and Khakhar 1994a; Baser and Khakhar 1994b; Geier, Winkler, and Piesche 2009):

where   and   are the pre-exponential factor and the activation energy,   is the gas constant,  T is the absolute temperature,   is the conversion of the gelling reaction and  ,   and are the initial molar concentrations of isocyanate, polyol and water in the reacting mixture. The last two terms between brackets represent the concentrations of polyol and isocyanate, respectively. The global blowing reaction is instead described with a pseudo-first-order model. In fact, since isocyanate is in excess with respect to water, it is assumed that its rate depends only on the concentration of water, resulting in the following expression Winkler (2009; Baser and Khakhar 1994a; Baser and Khakhar 1994b; Geier, Winkler, and Piesche 2009):

where   and   are the pre-exponential factor and the activation energy and   is the conversion of the blowing reaction.

As already mentioned, the blowing reaction produces carbon dioxide (and urea) that diffuses from the bulk of the liquid to the gas bubbles. The mass fraction of carbon dioxide in the liquid of the foam can therefore be calculated as follows:

where   is the system pressure (usually assumed constant and equal to the ambient pressure),   is the density of the polymerising liquid mixture (generally assumed constant and equal in this work to ) and   is the moment of order one of the growth rate due to carbon dioxide diffusion and will be explained in details later on.

The assumption of constant density (and therefore constant volume) for the liquid will be further discussed later on. The mixture undergoing polymerisation also contains the physical blowing agent that evaporates due to the temperature raise. The mass fraction of the physical blowing agent in the liquid of the foam can be calculated by solving the following equation:

where   is the moment of order one of the growth rate due to the diffusion of the blowing agent and will be also detailed later on. In this case the right-hand side contains only one negative source term, since the physical blowing agent is already present in the liquid and simply disappears due to its diffusion to the gas bubbles.

The temperature of the system is calculated by solving the enthalpy balance for the foam that reads as follows:

where   is the specific heat of the foam (taken in this work constant and equal to ),   is the latent heat of evaporation of the blowing agent,   and   are the heats of reaction for the blowing and gelling reactions respectively and   is the rate of change of the mass fraction of dissolved blowing agent in the liquid, as detailed in equation

.

As already mentioned, the model adopted in this work is based on a , which describes the evolution of the of the bubbles, or cells, of the foam. This function is defined in such a way that:  , represents the number of bubbles with volume in the infinitesimal size range between   and  , per unit volume of the liquid mixture undergoing polymerisation. Very often instead of tracking the evolution of the the problem is reformulated in terms of the moments of the . The generic moment of order   is defined as follows:

Each moment has a specific physical meaning:   represents the total bubble number density (i.e. total number of bubbles per unit volume of the liquid of the foam),   represents the total bubble volume per unit volume of liquid mixture whereas   is related to the variance of the . Being the polyurethane foam constituted by the liquid and the gas bubbles, in order to estimate the moments of the per unit volume of the foam the following transform is necessary:

where   is the moment of the based on the total volume of the foam. By using this notation the density of the foam can be readily calculated as follows:

where   is the density of the gas constituting the bubbles (calculated with the ideal gas law) whereas   is the density of the liquid mixture. The density of the liquid mixture undergoing polymerisation changes in time, as it is a function of composition, temperature and degree of polymerisation. Although a detailed model for describing these dependencies is under development, as already mentioned, for the moment this density is assumed constant. This is justified by the fact that while the foam expands, its behavior is dominated by the produced gas, rather than by the liquid phase. Therefore the error associated with this assumption is considered acceptable for this baseline model. The mean bubble (or cell) size is instead calculated as follows:

The evolution of the is dictated by the that reads as follows (time dependencies are omitted for brevity):

where   is the so-called bubble growth rate and   is the bubble coalescence kernel. The first term on the left-hand side represents accumulation, the second bubble growth, whereas the two terms on the right-hand side represent the formation and disappearance of bubbles due to coalescence.

The bubble growth rate is due to the diffusion of the physical blowing agent and of the carbon dioxide molecules to the gas bubbles of the foam. Although several detailed models can be formulated, in this work it is assumed that the growth of the bubbles is diffusion-controlled. The rate of diffusion-controlled growth processes is generally assumed to be proportional to the driving force around the bubble (i.e. concentration gradient). The proportionality constant can be theoretically calculated *a-priori* if the diffusion coefficients of the physical blowing agent and of carbon dioxide in the liquid mixture undergoing polymerisation are known. The overall growth rate is therefore calculated as the summation of the growth rate due to the blowing agent:

and due to carbon dioxide:

where   and   are the mass fractions of blowing agent and carbon dioxide in the liquid phase of the foam, whereas   and   are the corresponding equilibrium concentrations (which depend on the system temperature,  T). The proportionality constants   and   are treated in this work as model parameters and are varied between and , as these values correspond to reasonable diffusion coefficient values. Bubble coalescence is caused by the relative motion of the bubbles, due to shear and velocity gradients, that inevitably are imposed during the expansion of the foam. For shear-induced coalescence the resulting rate is expressed by the following sum kernelMarchisio and Fox (2013):

where   is proportional to the local shear rate and therefore depends on the flow conditions. As in this work the foam expansion is described via a simple zero-dimensional model this value is assumed constant and varied between and , corresponding to reasonable values for foam expansion flows.

By applying the moment transform the becomes:

where the moment of order   of the growth rates are defined as follows:

whereas the moment of order   of the source term due to coalescence reads as follows:

These latter source terms are calculated by using , which transforms these complex integrals into simple summations Marchisio and Fox (2013):

and:

where  ,   and   ,   are the   weights and nodes of the quadrature approximation.

For standard applications involving growth and coalescence usually  , requires the solution of the first four moments of the (i.e.  ,  ,   and  ).

## Status of recipes and adaptors

Figure

describes the recipes and adaptors (links) that are being included into the expanding polyurethane foam application described in the previous section. Boxes and ellipses stand for tools and surrogate models, respectively. Arrows indicate the flow of data forwards and backwards across scales. It should be noted that this only represents the first step which will be enhanced and augmented.

|  |  |
| --- | --- |
| Layouts of links for diffusivity, Henry coefficient, kinetics, liquid density, bubble growth and rheology. Boxes and ellipses stand for tools and surrogate models, respectively. | Layouts of links for diffusivity, Henry coefficient, kinetics, liquid density, bubble growth and rheology. Boxes and ellipses stand for tools and surrogate models, respectively. |
| Layouts of links for diffusivity, Henry coefficient, kinetics, liquid density, bubble growth and rheology. Boxes and ellipses stand for tools and surrogate models, respectively. | Layouts of links for diffusivity, Henry coefficient, kinetics, liquid density, bubble growth and rheology. Boxes and ellipses stand for tools and surrogate models, respectively. |
| Layouts of links for diffusivity, Henry coefficient, kinetics, liquid density, bubble growth and rheology. Boxes and ellipses stand for tools and surrogate models, respectively. | Layouts of links for diffusivity, Henry coefficient, kinetics, liquid density, bubble growth and rheology. Boxes and ellipses stand for tools and surrogate models, respectively. |

The status of the recipes and adaptors (links) is summarised in Figure

. There are 6 tools (computational fluid dynamics, bubble growth tool, kinetics, equation of statem nano and rheology) that are to be orchestrated in this application. 5 of the 6 tools and 3 of the 6 links are operational and have undergone testing. It is envisaged that the remaining ones will be operational for the mid-term.

The majority tools and links scheduled for implementation in the first application development phase are operational and have undergone testing. The final model predictions will validated against experiments in WP6.

|  |  |
| --- | --- |
| Status of recipes and adaptors. | Status of recipes and adaptors. |

Baser, S. A., and D. V. Khakhar. 1994a. “Modeling of the Dynamics of R-11 Blown Polyurethane Foam Formation.” *Polymer Engineering & Science* 34 (8): 632–641. <doi:[10.1002/pen.760340804](http://dx.doi.org/10.1002/pen.760340804)>. <http://dx.doi.org/10.1002/pen.760340804>.

———. 1994b. “Modeling of the Dynamics of R-11 Blown Polyurethane Foam Formation.” *Polymer Engineering & Science* 34 (8): 642–649. <doi:[10.1002/pen.760340804](http://dx.doi.org/10.1002/pen.760340804)>. <http://dx.doi.org/10.1002/pen.760340804>.

Beverte, Ilze. 2014. “Determination of Highly Porous Plastic Foam Structural Characteristics by Processing Light Microscopy Images Data.” *Journal of Applied Polymer Science* 131 (4): n/a–n/a. <doi:[10.1002/app.39477](http://dx.doi.org/10.1002/app.39477)>. <http://dx.doi.org/10.1002/app.39477>.

Chemieingenieurwesen, VDI-Gesellschaft Verfahrenstechnik und. 2010. *VDI Heat Atlas; 2nd Ed.* Berlin: Springer.

Codd, Edgar F. 1970. “A Relational Model of Data for Large Shared Data Banks.” *Communications of the ACM* 13 (6): 377–387.

Geier, S., C. Winkler, and M. Piesche. 2009. “Numerical Simulation of Mold Filling Processes with Polyurethane Foams.” *Chemical Engineering & Technology* 32 (9): 1438–1447. <doi:[10.1002/ceat.200900202](http://dx.doi.org/10.1002/ceat.200900202)>. <http://dx.doi.org/10.1002/ceat.200900202>.

Gibson, L.J., and M.F. Ashby. 1999. *Cellular Solids: Structure and Properties*. Cambridge Solid State Science Series. Cambridge University Press. <https://books.google.no/books?id=IySUr5sn4N8C>.

Klempner, D., and K.C. Frisch. 1991. *Handbook of Polymeric Foams and Foam Technology*. Hanser Gardner Publications. <http://books.google.no/books?id=QbcZHAAACAAJ>.

Lo, Yu-Wen, Danny D. Reible, John R. Collier, and Cheng-Ho Chen. 1994. “Three-Dimensional Modeling of Reaction Injection Molding. I.” *Polymer Engineering & Science* 34 (18): 1393–1400. <doi:[10.1002/pen.760341805](http://dx.doi.org/10.1002/pen.760341805)>. <http://dx.doi.org/10.1002/pen.760341805>.

Marchisio, D.L., and R.O. Fox. 2013. *Computational Models for Polydisperse Particulate and Multiphase Systems*. Cambridge Series in Chemical Engineering. Cambridge University Press.

Mosanenzadeh, Shahrzad Ghaffari, Hani E. Naguib, Chul B. Park, and Noureddine Atalla. 2013. “Development, Characterization, and Modeling of Environmentally Friendly Open-Cell Acoustic Foams.” *Polymer Engineering & Science* 53 (9): 1979–1989. <doi:[10.1002/pen.23443](http://dx.doi.org/10.1002/pen.23443)>. <http://dx.doi.org/10.1002/pen.23443>.

Näsholm, Petter. 2012. “Extracting Data from NoSQL Databases-a Step Towards Interactive Visual Analysis of NoSQL Data.” Master’s thesis, Chalmers University of Technology.

Niyogi, Debdarsan, Rajinder Kumar, and Kandukuri S. Gandhi. 2014. “Modeling of Bubble-Size Distribution in Water and Freon Co-Blown Free Rise Polyurethane Foams.” *Journal of Applied Polymer Science* 131 (18): n/a–n/a. <doi:[10.1002/app.40745](http://dx.doi.org/10.1002/app.40745)>. <http://dx.doi.org/10.1002/app.40745>.

Oertel, G., and L. Abele. 1994. *Polyurethane Handbook: Chemistry, Raw Materials, Processing, Application, Properties*. Hanser.

project, Fireworks. 2015. “Project Web-Site.” march. <http://pythonhosted.org/FireWorks/>.

project, MongoDB. 2015. “Project Web-Site.” march. <http://www.mongodb.org/>.

Project, MongoDB Documentation. 2014. *MongoDB Documentation*. <http://docs.mongodb.org/master/MongoDB-manual.pdf>.

project, MongoEngine. 2015. “Project Web-Site.” march. <http://www.mongoengine.org/>.

project, R. 2015. “Project Web-Site.” march. <http://www.r-project.org/>.

Shen, Lu, Yusheng Zhao, Ali Tekeei, Fu-Hung Hsieh, and Galen J. Suppes. 2014. “Density Modeling of Polyurethane Box Foam.” *Polymer Engineering & Science* 54 (7): 1503–1511. <doi:[10.1002/pen.23694](http://dx.doi.org/10.1002/pen.23694)>. <http://dx.doi.org/10.1002/pen.23694>.

Shermin, Motahera. 2013. “An Access Control Model for NoSQL Databases.” Master’s thesis, University of Western Ontario.

Tesser, R., M. Di Serio, A. Sclafani, and E. Santacesaria. 2004. “Modeling of Polyurethane Foam Formation.” *Journal of Applied Polymer Science* 92 (3): 1875–1886. <doi:[10.1002/app.20170](http://dx.doi.org/10.1002/app.20170)>. <http://dx.doi.org/10.1002/app.20170>.

Winkler, Christian A. 2009. “PhD Thesis.” PhD thesis, University of Stuttgart.

Woods, George. 1990. *Polyurethane Handbook: Chemistry, Raw Materials, Processing, Application, Properties*. ICI Polyurethanes; Wiley in Chichester, New York .

Zhao, Yusheng, Michael J. Gordon, Ali Tekeei, Fu-Hung Hsieh, and Galen J. Suppes. 2013. “Modeling Reaction Kinetics of Rigid Polyurethane Foaming Process.” *Journal of Applied Polymer Science* 130 (2): 1131–1138. <doi:[10.1002/app.39287](http://dx.doi.org/10.1002/app.39287)>. <http://dx.doi.org/10.1002/app.39287>.

Zhou, Hong, Bo Li, and Guangsu Huang. 2006. “Sound Absorption Characteristics of Polymer Microparticles.” *Journal of Applied Polymer Science* 101 (4): 2675–2679. <doi:[10.1002/app.23911](http://dx.doi.org/10.1002/app.23911)>. <http://dx.doi.org/10.1002/app.23911>.