Delivery date:

31-08-2015

Authors:

Sigve Karolius

MoDeNa

Deliverable D4.3

Backward Mapping Tool

Principal investigators:

Sigve Karolius
Heinz A Preisig
Henrik Rusche
Dominik Christ
Hrvoje Jasak

Collaborators:

Cansu Birgen

Project coordinator:

Heinz A Preisig

heinz.preisig@chemeng.ntnu.no

Abstract:

MoDeNa implements the concept of surrogate models that represent approximation of complex models. The surrogate model is fitted to the behaviour of the complex models using standard techniques. Once identified, the surrogate can be used to replace the complex model with the objective to reduce the computational load. Fitting is done on demand, i.e. the model requesting the surrogate is controlling the domain in which the surrogate is fitted and th strategy on how the domain is being extended and what error criterion is used. The implementation works recursively over the scales.

© 2015
MoDeNa

Contents

1 Introduction **1**

2 Backward Mapping Models in the MoDeNa Framework **1**

2.1 User-level Backward Mapping Model 2

2.2 The Backward Mapping FireTask 3

2.2.1 Framework-level backward mapping surrogate model definition 4

2.3 Backward Mapping Workflow 9

3 Strategies **11**

3.1 Fundamental Framework Implementation 12

3.2 Specific Strategy Implementation Example 14

4 Backward mapping in practice **14**

1 Introduction

The term backward mapping refers to comparing the output from a surrogate model to that of the corresponding exact model. However, in order to clarify the syntax of the computer code in the MoDeNa framework, the code employs a terminology for the surrogate model that is somewhat different: From a programming perspective the implementation of a surrogate model must include a callable function; therefore, the framework distinguishes between the surrogate function, i.e. the callable function, and the surrogate model.

The reason for the design choice was to facilitate the implementation of surrogate models with different properties, e.g. backward mapping capabilities. Consequently, backward mapping is not a standalone module in the MoDeNa framework, but rather implemented as a property of the surrogate model itself. There are two kinds of surrogate models in the MoDeNa framework: backward mapping and forward mapping models. Only the backward mapping model can compare the output from the surrogate function to that of the exact model, thus validating the parameters of the surrogate function.

Since the backward mapping task itself is simply to compare the input-output from the surrogate function to that of the exact simulation, i.e. after it the surrogate function parameters have been fitted, the procedure itself is dependent on the design of experiments and parameter fitting. The standalone features of the framework is therefore the design of experiments and parameter fitting modules, which are independent from one another; therefore, in order to make the framework more modular, the concept of strategies was introduced. This allows the user to choose the strategies for design of experiments and parameter fitting. Moreover, the R statistical language is used as a backend for statistical methods of design of experiments and nonlinear regression, thus making it quicker to implement space filling designs, regression and statistical methods.

The report will explain how the backward mapping procedure is integrated into the MoDeNa framework, and elaborate on the strategies that allow for the flexible instantiation. Further it describes the integration of the computational recipes written in the R statistical language.

2 Backward Mapping Models in the MoDeNa Framework

From the users perspective the backward mapping capabilities of the MoDeNa framework are facilitated by adding strategies specific to: initialisation, design of experiments, and parameter fitting as shown in Listing 1. This design choice has been made primarily for two reasons:

- Easy to use and understand
- Quick implementation and integration of new modules

This section will introduce the big picture of the technical aspects behind the implementation of the "Backward Mapping Model", i.e. the backward mapping tool, in order to avoid getting caught up in coding-specific details related to the implementation. The report will therefore specifically focus on explaining how the MoDeNa framework implementation integrates into:

1. Fireworks
    - FireTask
    - Workflow
2. MongoDB

- Looking up surrogate models
- Updating the database

3. The low-level C-Framework

2.1 User-level Backward Mapping Model

The ”backward mapping model” class in the MoDeNa framework is responsible for handling surrogate models that require backward mapping. For the purpose of completeness, note that the framework also defines a ”forward mapping model”; which, as the name dictates, does not facilitate any backward mapping capabilities.

The code block in Listing 1 shows an example of how the user will define a backward mapping model. In addition to specify strategies for: initialisation, design of experiments and parameter fitting, the user has to suply a name, surrogate function and exact task. The latter is the simulation whose input-output data the surrogate models parameter will be fitted to.

```
m = BackwardMappingModel(
    _id= 'flowRate',
    surrogateFunction= f,
    exactTask= 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
    ),
)
```

Listing 1: The code block shows an example of an instantiation of a backward mapping surrogate ”flowRate”. The surrogate model is instantiated using notation equivalent to a JSON document where every key has an associated value. The model defines a ”surrogate function” in line 3, whose parameters will be fitted to the input-output data of the ”exactTask” in line 4. The strategies in lines: 6, 15, 18 and 21 are triggered by events during the simulation.

Note that from the user perspective, just by looking at the definition in Listing 1, it is not at all obvious how the underlying framework works. From this section it should be noted that the behaviour of the backward mapping model, i.e. the strategies for initialisation etc., is specified by the user using strategies. Consequently, it can be concluded that there are ”slots” in the underlying framework, i.e. ”missing” blocks of code that are designed to be completed by the user, called strategies.

2.2 The Backward Mapping FireTask

In order to perform the actual backward mapping, i.e. compare output from the surrogate function, e.g. line 3 from Listing 1, to that of the exact model, such as the one defined in line 4, it is necessary to start a FireTask that can override the main workflow of the simulation. The reason for this is that if the backward mapping is being performed by a surrogate model it is because the parameters of the model are no longer valid and are in the process of being updated. It is therefore logical that the simulation using the surrogate model cannot continue.

The MoDeNa framework therefore introduces two classes: `Backward Mapping Task` and `Backward Mapping Script Task`. The latter is the FireTask that Fireworks executes in order to handle the overall simulation, and the first is a wrapper which analyses the output from the simulation when an error occurs, and modifies the workflow accordingly.

The `Backward Mapping Task` in Listing 2 only contains one method, `handleReturnCode`, which catches the output from the main simulation and takes action accordingly. There are currently two ways the main simulation can exit and be caught by the framework using error codes 201 and 200 for referring to instantiation or a surrogate model being out of bounds in lines 7 and 23 respectively. Any other error code will, according to line 40, terminate the the simulation with an error, finally the simulation continues until it has succeeded in line 44.

```
1  class BackwardMappingTask:
2
3      def handleReturnCode(self, returnCode):
4
5          # Analyse return code
6          print('return code = %i' % returnCode)
7          if returnCode == 201:
8              print term.cyan + ''Performing Initialisation'' + term.normal
9              model = modena.SurrogateModel.loadFromModule()
10
11             # Continue with exact tasks, parameter estimation,
12             # and (finally) this
13             # task in order to resume normal operation
14
15             print model.initialisationStrategy()
16             wf = model.initialisationStrategy().workflow(model)
17             wf.addAfterAll(
18                 Workflow2([Firework(self)], name='original task')
19             )
20
21             return FWAction(detours=wf)
22
23         elif returnCode == 200:
24             print term.cyan + ''Performing Design of Experiments'' + term.normal
25             model = modena.SurrogateModel.loadFailing()
26
27             # Continue with exact tasks, parameter estimation,
28             # and (finally) this
29             # task in order to resume normal operation
30             wf = model.outOfBoundsStrategy().workflow(
31                 model,
32                 outsidePoint= model.outsidePoint
33             )
34             wf.addAfterAll(
35                 Workflow2([Firework(self)], name='original task')
36             )
37
38             return FWAction(detours=wf)
39
40         elif returnCode > 0:
```

```
                print('An␣error␣occurred')
                sys.exit(returnCode)

            else:
                print('Success␣-␣We␣are␣done')
                return FWAction()
```
Listing 2: The script contains the framework implementation of a backward mapping task which keeps track of the error codes from the simulations in the framework.

The class `BackwardMappingScriptTask` in Listing 4 represents the main simulation that the user wants to perform. From the user perspective the simulation is an executable file, e.g. line 2 in Listing 3, with a function returning the appropriate error codes if one of the surrogate models that are used is out of bounds.

```
m = BackwardMappingScriptTask(
    script='nameOfExecutableFile'
)
```
Listing 3: The code block shows an example shows how the user would specify a script, which contains calls to various surrogate models, which the framework should simulate.

The underlying `Backward Mapping Script Task` in the framework uses the script parameter in line 4, which is automatically executed by Fireworks due to the `run_task` method in line 6. The error code from the simulation is passed to the `handleReturnCode` function when the simulation terminates; which, is available because of the inheritance of the `Backward Mapping Task` class from line 2 in Listing 2.

```
@explicit_serialize
class BackwardMappingScriptTask(ScriptTask, BackwardMappingTask):

    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(BackwardMappingScriptTask, self).run_task(fw_spec)

        return self.handleReturnCode(ret.stored_data['returncode'])
```
Listing 4: The code block shows the implementation of a backward mapping script task which performs a simulation.

This section summarised how the framework base classes facilitate the backward mapping by allowing the simulation, i.e. the `Backward Mapping Script Task`, to ”fail” in order to perform design of experiments and parameter fitting according to the `Backward Mapping Script` before re-starting the simulation.

2.2.1 Framework-level backward mapping surrogate model definition

Section 2.2 explained how the framework facilitates the main simulation to be halted in order to perform design of experiments and parameter fitting. Moreover, Listing 1 it was introduced how the concept of

strategies is used by the MoDeNa framework to allow the user to specify how the backward mapping is performed without writing code, but simply by ”filling in the blanks”.

However, the section did not explain how the `Backward Mapping Model` uses the strategies and how the database is being kept up-to-date. The `Backward Mapping Model` class, shown in Listing 5, is substantial, and this section will outline how the class is used in the framework. Even though the inner workings of the code will not be explained in detail it will be emphasised which part of the code, i.e. the methods of the class, is responsible for instantiation, parameter fitting etc.

There are a total of nine methods defined in the `Backward Mapping Model`:

`__init__` (line 11): Instantiate a new surrogate model object, use a model from database if possible, otherwise create a new model based on the input e.g. from Listing 1.

`exactTasks` (line 98): Build a workflow to excute an `exactTask` for each point.

`initiationStrategy` (line 125): Load the initialisation strategy

`parameterFittingStrategy` (line 131): Load the parameter fitting strategy

`outOfBoundsStrategy` (line 137): Load the out of bounds strategy

`updateFitDataFromFwSpec` (line 143): Update database with new parameters

`updateMinMax` (line 160): Update database with new surrogate model min/max boundaries

`error` (line 179): Calculate the error between the exact simulation and the surrogate function

`extendedRange` (line 196): Expand the domain of the surrogate model

The reader is left to consider the code for the `Backward Mapping Model` in Listing 5 without further adue. However, instead of reading the code line-by-line locate the methods listed above above; moreover, pay special attention to how the methods loading a particular strategy refers directly to the keys that are used in the user-level example in Listing 1.

```python
class BackwardMappingModel(SurrogateModel):

    # Database definition
    inputs = MapField(EmbeddedDocumentField(MinMaxArgPosOpt))
    outputs = MapField(EmbeddedDocumentField(MinMaxArgPosOpt))
    fitData = MapField(ListField(FloatField(required=True)))
    substituteModels = ListField(ReferenceField(SurrogateModel))
    outsidePoint = EmbeddedDocumentField(EmbDoc)
    meta = {'allow_inheritance': True}

    def __init__(self, *args, **kwargs):
        SurrogateModel.__init__(self, *args, **kwargs)

        if kwargs.has_key('_cls'):
            DynamicDocument.__init__(self, *args, **kwargs)

        else:
            if not kwargs.has_key('_id'):
                raise Exception('Need␣_id')
            if not kwargs.has_key('surrogateFunction'):
                raise Exception('Need␣surrogateFunction')
            if not isinstance(kwargs['surrogateFunction'], SurrogateFunction):
                raise TypeError('Need␣surrogateFunction')

```

```
25              kwargs['inherited_inputs'] = 0
26
27              kwargs['fitData'] = {}
28              kwargs['inputs'] = {}
29              for k, v in kwargs['surrogateFunction'].inputs.iteritems():
30                  kwargs['inputs'][k] = v.to_mongo()
31
32              kwargs['outputs'] = {}
33              for k, v in kwargs['surrogateFunction'].outputs.iteritems():
34                  kwargs['fitData'][k] = []
35                  kwargs['outputs'][k] = MinMaxArgPosOpt(**{})
36
37              subOutputs = {}
38              for m in kwargs['substituteModels']:
39                  if not isinstance(m, SurrogateModel):
40                      raise TypeError(
41                          'Elements of substituteModels '
42                          'must be derived from SurrogateModel'
43                      )
44                  subOutputs.update(m.outputsToModels())
45
46              nInp = len(kwargs['inputs'])
47              replaced = {}
48              while True:
49                  found = None
50                  for o in subOutputs:
51                      if o in kwargs['inputs']:
52                          found = o
53                          break
54
55                  if found == None:
56                      break
57
58                  del kwargs['inputs'][o]
59                  for k, v in subOutputs[o].inputs.iteritems():
60                      if not k in kwargs['inputs']:
61                          kwargs['inputs'][k] = { 'argPos': nInp }
62                          nInp += 1
63
64              nInputs = 0
65              for k, v in kwargs['inputs'].iteritems():
66                  kwargs['fitData'][k] = []
67                  kwargs['inputs'][k] = MinMaxArgPosOpt(**v)
68
69              checkAndConvertType(kwargs, 'exactTask', FireTaskBase);
70
71              checkAndConvertType(
72                  kwargs,
73                  'initialisationStrategy',
74                  InitialisationStrategy
75              );
76
77              checkAndConvertType(
78                  kwargs,
79                  'outOfBoundsStrategy',
80                  OutOfBoundsStrategy
81              );
82
83              checkAndConvertType(
84                  kwargs,
85                  'parameterFittingStrategy',
86                  ParameterFittingStrategy
87              );
88
89              DynamicDocument.__init__(self, *args, **kwargs)
```

```
            indices = self.parseIndices()
            for k,v in indices.iteritems():
                kwargs['surrogateFunction'].indices[k].get_index(v)

            self.save()


    def exactTasks(self, points):

        indices = self.parseIndices()

        # De-serialise the exact task from dict
        et = load_object(self.meth_exactTask)

        tl = []
        e = six.next(six.itervalues(points))
        for i in xrange(len(e)):
            p = {}
            for k in points:
                p[k] = points[k][i]

            for m in self.substituteModels:
                p.update(m.callModel(p))

            t = et
            t['point'] = p
            t['indices'] = indices
            fw = Firework(t)

            tl.append(fw)

        return Workflow2(tl, name='exact tasks for new points')


    def initialisationStrategy(self):
        return loadType(self,
            'initialisationStrategy',
            InitialisationStrategy)


    def parameterFittingStrategy(self):
        return loadType(self,
            'parameterFittingStrategy',
            ParameterFittingStrategy)


    def outOfBoundsStrategy(self):
        return loadType(self,
            'outOfBoundsStrategy',
            OutOfBoundsStrategy)


    def updateFitDataFromFwSpec(self, fw_spec):
        for k in self.inputs: # Load the fitting data
            if fw_spec[k][0].__class__ == list:
                self.fitData[k].extend(fw_spec[k][0])
            else:
                self.fitData[k].extend(fw_spec[k])

        for k in self.outputs:
            if fw_spec[k][0].__class__ == list:
                self.fitData[k].extend(fw_spec[k][0])
            else:
                self.fitData[k].extend(fw_spec[k])
```

```
155
156         firstSet = six.next(six.itervalues(self.fitData)) # Get first set
157         self.nSamples = len(firstSet)
158
159
160     def updateMinMax(self):
161         if not self.nSamples:
162             for v in self.inputs.values():
163                 v.min = 9e99
164                 v.max = -9e99
165
166             for v in self.outputs.values():
167                 v.min = 9e99
168                 v.max = -9e99
169
170         for k, v in self.inputs.iteritems():
171             v.min = min(self.fitData[k])
172             v.max = max(self.fitData[k])
173
174         for k, v in self.outputs.iteritems():
175             v.min = min(self.fitData[k])
176             v.max = max(self.fitData[k])
177
178
179     def error(self, cModel, **kwargs):
180         idxGenerator = kwargs.pop('idxGenerator', xrange(self.nSamples))
181
182         in_i = list()
183         i = [0] * (1 + self.inputs_max_argPos())
184
185         output = self.fitData[six.next(six.iterkeys(self.outputs))]
186
187         for j in idxGenerator:
188             for k, v in self.inputs.iteritems():
189                  i[v.argPos] = self.fitData[k][j] # Load inputs
190
191             out = cModel.call(in_i, i)             # Call the surrogate model
192
193             yield out[0] - output[j]
194
195
196     def extendedRange(self, outsidePoint, expansion_factor=1.2):
197
198         sampleRange = {}
199
200         for k, v in self.inputs.iteritems():
201             sampleRange[k] = {}
202             outsideValue = outsidePoint[k]
203
204             # If the value outside point is outside the range, set the
205             # ''localdict'' max to the outside point value
206
207             if outsideValue > v['max']:
208                 sampleRange[k]['min'] = v['max']
209                 sampleRange[k]['max'] = min(
210                     outsideValue*expansion_factor,
211                     self.surrogateFunction.inputs[k].max
212                 )
213
214             elif outsideValue < v['min']:
215                 sampleRange[k]['min'] = max(
216                     outsideValue/expansion_factor,
217                     self.surrogateFunction.inputs[k].min
218                 )
219                 sampleRange[k]['max'] = v['min']
```

```
220
221                 else:
222                     sampleRange[k]['min'] = v['min']
223                     sampleRange[k]['max'] = v['max']
224
225             return sampleRange
```

Listing 5: The code block shows the implementation of the backward mapping surrogate model in the MoDeNa framework. The class is made up of a total of nine methods: `__init__` (line 11), `exactTasks` (line 98), `initiationStrategy` (line 125), `parameterFittingStrategy` (line 131), `outOfBoundsStrategy` (line 137), `updateFitDataFromFwSpec` (line 143), `updateMinMax` (line 160), `error` (line 179) and `extendedRange` (line 196). The backward mapping model connects the database, MoDeNa framework and Fireworks by providing strategies to the Fireworks, saving and retrieving data from the database and calling the low-level C-function implemented in the surrogate function.

Not even the `Backward Mapping Model` represents ”the bottom of the barrel” of the MoDeNa framework, there is yet another layer of abstraction indicated by the inheritance of `Surrogate Model` in line 1. Instead of explaining the details of the code block the next section will visualise how and when the different methods are called during different stages of the backward mapping procedures.

2.3 Backward Mapping Workflow

The model-based design of experiments (MBDoD) workflow, outlined in Figure 1, is central to the backward mapping strategies of the MoDeNa framework. Imagine that a simulation of a `Backward`

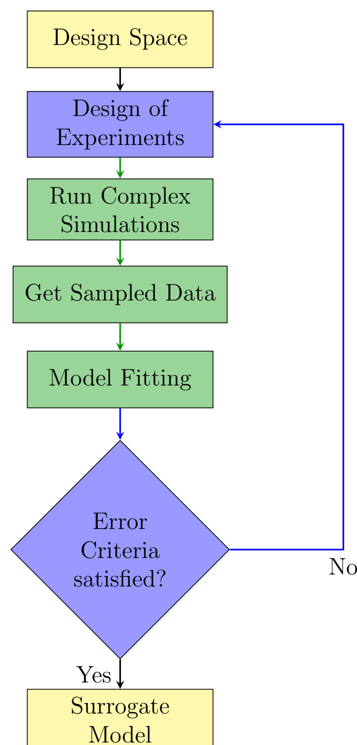

Figure 1: Illustration of model-based design of experiments computational workflow.

`Mapping Script Task` akin to the example in Listing 3, which uses a backward mapping surrogate

model similar to that of Listings 1. Firstly, according to line 16 in Listing 4, the `Backward Mapping Script Task` is started. However, if a surrogate model has not yet been instantiated the simulation fail immediately, throwing error code 201 in the process. The error code is passed to the `return code handler` in the `Backward Mapping Task` in Listing 2, where the boolean check in line 7 is triggered.

This will cause the framework to instantiate all necessary models, loading the `initialistation strategy` using the method in line 125. The initialisation in Listing 1 specifies which points the initialisation should be performed in, shown as black points in Figure 2. The initialisation strategy calls the `exactTasks` in line 98 of the `Backward Mapping Model` (Listing 4), which creates a workflow which performs all the exact simulations and saves the data, thus turning the points blue. After the

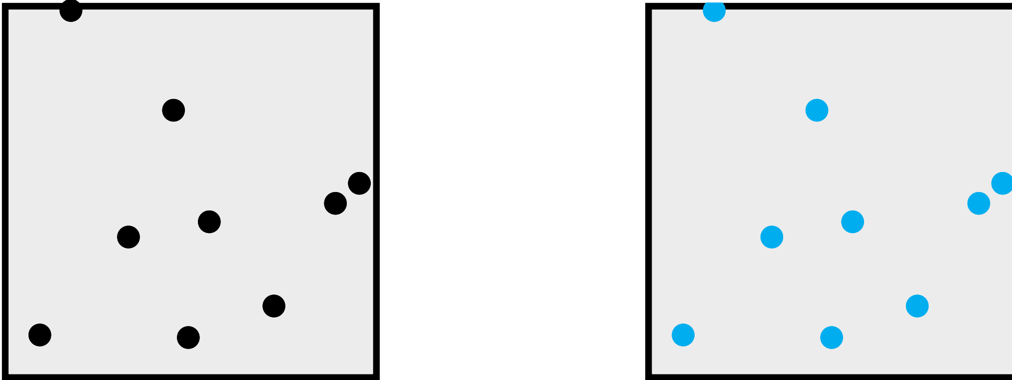

Figure 2: The figure illustrates the initialisation of a surrogate model by filling the design space of the model with **points** that should be explored further. The framework simulates the points and performs parameter fitting of the surrogate function using a subset of the simulations. The rest of the points are used in order to validate the surrogate model.

exact simulations have been performed, the surrogate function is fitted to the input-output data. The user-specified parameter fitting strategy from Listing 4 is loaded using the method in line 131 from Listing 4, and the nonlinear regression results in a set of parameters that is verified against a subset of the simulated points that where not used in the fitting procedure.

The validation procedure uses the `error` method in line 179 in Listing 5 in order to evaluate the error between the surrogate function and the exact simulation. In the case where all the points are accepted (green) the model is ready to be used. However, if a point is rejected it is necessary to perform additional simulations. This situation is illustrated in Figure 3, which shows that when the parameters are rejected, the framework adds more points to the design space, keeping the points that have already been simulated, and performs the exact simulation only for the new points. This procedure will be repeated, adding points, fitting the surrogate function and performing validation until the error criterion is satisfied. The simulation in the `Backward Mapping Script Task` will automatically start and the MoDeNa framework can now evaluate the surrogate function inside domain, this is illustrated using by the brown trajectory in Figure 4. When the surrogate model reaches the boundary of the validated design space the simulation will terminate, throwing error 200, which triggers any surrogate model that is ”out of bounds” to expand the domain by loading the `out of bounds strategy` using the method from Listing 5 in line 137.

The framework can for instance expand the domain by 20%, and try to fit the surrogate function by adding simulations to a new domain and fit the surrogate function globally over both the old and new domains as illustrated in figure 5. When the surrogate model has been validated the simulation is automatically re-started, and the process continues to the simulation is complete. The introduction

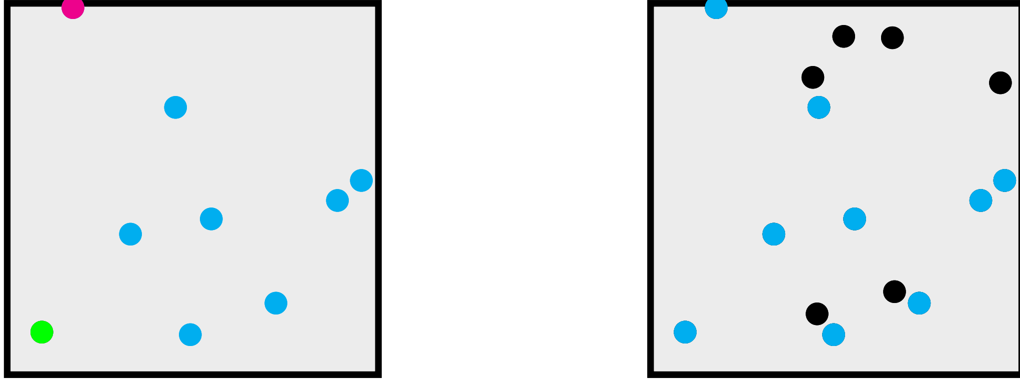

Figure 3: The figure illustrates how a subset of the simulated points is used for the validation of the surrogate model, note that the subset that is used for the validation is **not** used for fitting the parameters of the surrogate function. The input-output of the surrogate function and the simulation is compared and checked against an error criterion, each point is either rejected or validated. In the case that any of the points are rejected, the framework will employ a strategy for improving the error of the surrogate model, adding **new points** to the domain in an attempt that increasing the resolution will generate a better fit for the parameters of the surrogate function.

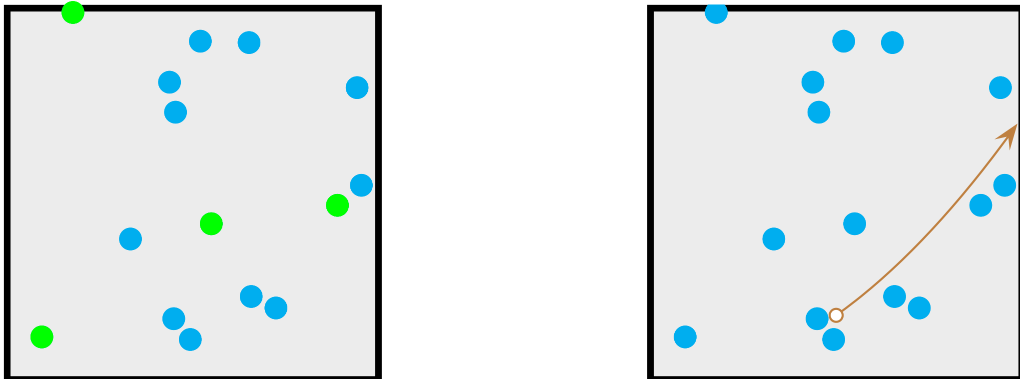

Figure 4: The figure illustrates that only a subset of the simulated points, not used for the parameter fitting of the surrogate function, is used for validation. When the surrogate model has been validated for a domain it can be evaluated, in this example allong a trajectory, untill it reaches the bounds of the domain.

of strategies into the MoDeNa framework is a result of cooperation between the contributors in work package 4 and 5. Even though it is a more abstract implementation than a standalone backward mapping module it has ultimately lead to a much more flexible implementation that can easily be modified and extended depending on the needs of the user. Moreover, it has intertwined MoDeNa, MongoDB and Fireworks into a more complete modelling tool which can be said to ”speak the language of multi-scale modelling”.

3 Strategies

A strategy in the MoDeNa framework is a way of introducing event handling into the simulation. Instead of having one centralised event-handler, every surrogate model is allowed to influence the

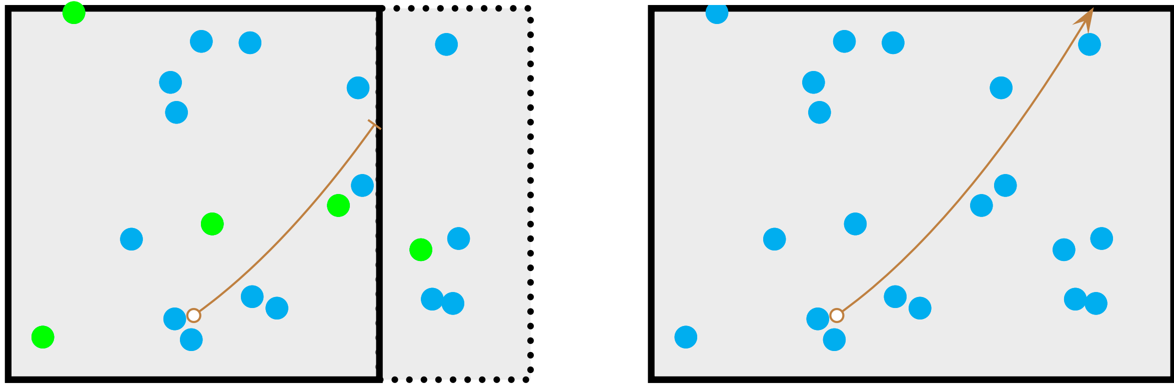

Figure 5: The figure illustrates the process of a simulation, where the surrogate function has been used along the trajectory, has been stopped, and the domain for the surrogate model is expanded to the dotted lines. New points are added to the domain and exact simulations are performed at every point before the surrogate function is fitted and validation globally across the entire domain. Afterwards, the simulation is re-started, and the process continues until the simulation is complete.

computational workflow according to their needs; thus the event handling is implicitly handled by Fireworks, which happily executes the computational workflows as they are introduced.

From a framework perspective a strategy represents a exchangeable block of code. The backward mapping model in Listings 1 could for instance change the initialisation strategy in line 6 with a different one, e.g. "Initial Data" which initialises a model given the input-output data without executing the exact task. Moreover, the strategy for the parameter fitting, which is where the backward mapping is performed, can also be changed.

All-in-all the introduction of strategies has allowed the framework to become more modularised than with an external backward mapping module, especially since new strategies for design of experiments and parameter fitting can easily be formulated as necessary because the framework uses the R-statistical language.

3.1 Fundamental Framework Implementation

The framework-level implementation of strategies is divided into two-pieces, e.g. for initialisation: `Initialisation Strategy` in Listing 6, i.e. the base-class for the different initialisation methods, and `Initialisation` in Listing 7 which creates a FireTask for the initialisation strategy.

The `Initialisation Strategy` in Listing 6 is intended to serve as a base-class for specific initialisation strategies, i.e. it should be inherited by other classes. This can be seen in the method `new Points` in line 8, which must be overwritten by the child-class as shown in the example implementation in Listing 8. However, the main functionality of the `Initialisation Strategy` is provided by the `workflow` method in line 12, which generates a workflow for the initialisation that can be executed by Fireworks. Note that the link between the `Initialisation Strategy` and the `Initialisation` FireTask is that the purpose of the latter is to call the `workflow` method of the first. This happens due to the `run_task` on line 13 in Listing 7, which is automatically executed by Fireworks.

```
1  class InitialisationStrategy(defaultdict, FWSerializable):
2
3      def __init__(self, *args, **kwargs):
4          dict.__init__(self, *args, **kwargs)
5
```

```
6
7     @abc.abstractmethod
8     def newPoints(self):
9         raise NotImplementedError(''newPoints not implemented!'')
10
11
12    def workflow(self, model):
13        p = self.newPoints()
14        if len(p):
15            wf = model.exactTasks(p)
16            wf.addAfterAll(model.parameterFittingStrategy().workflow(model))
17            return wf
18        else:
19            return Workflow2([])
20
21
22    @serialize_fw
23    @recursive_serialize
24    def to_dict(self):
25        return dict(self)
26
27
28    @classmethod
29    @recursive_deserialize
30    def from_dict(cls, m_dict):
31        return cls(m_dict)
32
33
34    def __repr__(self):
35        return '<{}>:{}'.format(self.fw_name, dict(self))
```

Listing 6: The code block shows the generic base-class (mother-class) for the different initialisation strategies in the MoDeNa framework. The reason why it should be considered a base-class is that it will not work stand-alone. Instead the class must be inherited by a child-class, which should overwrite the `newPoints` method in Line 8; thus, the difference between the different initialisation strategies is largely their implementation of `newPoints`. The purpose of the class is to return the necessary `workflow` in order to initialise a surrogate model, this is done when the `workflow` in line 12 is called. The approach of using base-classes that returns the necessary workflow in order to perform initialisation, parameter fitting etc. is carried out for all the different types of strategies in the MoDeNa framework, which makes it easy to implement new strategies. Generally, it is only necessary to implement one method in order to create a new strategy when the respective base-class is inherited. The "magic" of integrating the strategy into the MoDeNa workflow will thereby be handled by the base class.

The `Initialisation` FireTask in Listing 7 is acts as a buffer between the `Initialisation Strategy` and the Fireworks workflow; thus allowing the workflow to be modified as necessary instead of always executing the same procedure.

```
1  @explicit_serialize
2  class Initialisation(FireTaskBase):
3
4      def __init__(self, *args, **kwargs):
5          FireTaskBase.__init__(self, *args, **kwargs)
6
7          if kwargs.has_key('surrogateModel'):
8              if isinstance(kwargs['surrogateModel'], modena.SurrogateModel):
9                  self['surrogateModelId'] = kwargs['surrogateModel']['_id']
10                 del self['surrogateModel']
11
12
13     def run_task(self, fw_spec):
14         print term.cyan + ''Performing initialisation'' + term.normal
```

```
15
16              model=modena.SurrogateModel.load(self['surrogateModelId'])
17
18              return FWAction(
19                  detours=model.initialisationStrategy().workflow(model)
20              )
```

Listing 7: The code block shows the implementation of the `Initialisation` FireTask, whose purpose is to call the `workwlow` method of the `initialisation strategy` that belongs to the surrogate model; thereby, the class tells Fireworks to take a detour in order to execute the workflow that is necessary in order to instantiate the model. The parameter fitting, out of bounds and improve error strategies employs a similar implementation, albeit with subtile diferences.

The parameter fitting, out of bounds and improve error strategies all follow a similar generic framework implementation in that there is one FireTask similar to that of Listing 7, and a base-class, such as the one in Lsiting 6, that is responsible for generating the workflow for the strategy. The specific implementation of the strategies themselves are naturally very different, but all inherit their respective base-class as shown in line 2 in Listing 8.

3.2 Specific Strategy Implementation Example

The initialisation strategy used in Listing 1 was called `Initial Points`, the implementation of that strategy is shown in Listing 8. Line 8 in Listing 6 implies that any new initialisation strategy is required to implement the method `new Points`, which is used by the method that generates the `workflow` for the initialisation in line 12.

```
1  @explicit_serialize
2  class InitialPoints(InitialisationStrategy):
3
4      def __init__(self, *args, **kwargs):
5          InitialisationStrategy.__init__(self, *args, **kwargs)
6
7      def newPoints(self):
8          return self['initialPoints']
```

Listing 8: The code block shows the implementation of the initialisation strategy corresponding to the user providing the framework with specific input points, which the framework then will perform simulations before fitting the surrogate function to the input-output data.

The framework currently have two generic initialisation strategies implemented in addition to `Initial Points`: `Initial Data`, where the user provides both input and output, and `Empty Initialisation Strategy` which does nothing. The latter is primarily used for forward mapping models.

4 Backward mapping in practice

The two-tank example that was presented in deliverable 5.5 exemplifies how the backward mapping is automatically performed as a part of the workflow. The example is designed to show the structure of the framework using a simple model of air, which is being discharged through a valve from one reservoir to another. From a MoDeNa perspective the flow through the nozzle must be considered as a complex problem, such as a 3D Computational FluidDynamics or a different detailed model. Consequently, the two reservoirs are macroscopic models and the rate of discharge of fluid through the nozzle represents a microscopic model. It demonstrated backward mapping because the domain of the inputs where

assumed to be unknown a priori; therefore the simulation had to expand the design space for the "backward mapping model" throughout the simulation,

The final result of the simulation was the plot in Figure 6, which shows how the pressure of the reservoirs 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

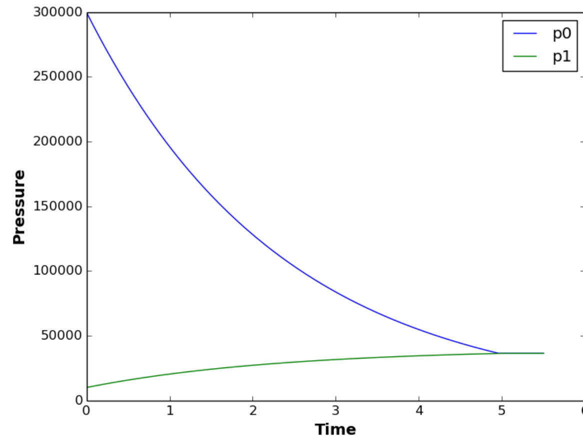

Figure 6: Pressures in each of the two reservoirs as a function of time.

that the parameters of the surrogate function where changing dynamically throughout the simulation, Figure 7 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.

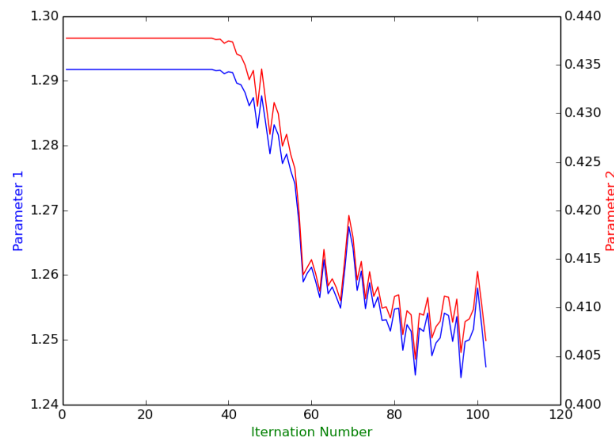

Figure 7: Model parameters as function of the number of backward mapping iterations.