# easi Documentation

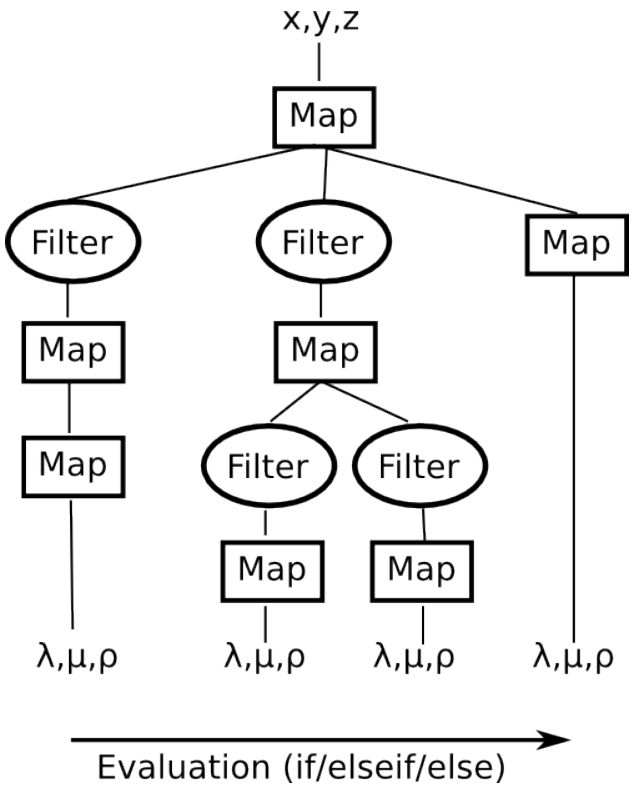
**Carsten Uphoff**

**Jan 24, 2022**

# CONTENTS

easi is a library for the **E**asy **I**nitialization of models in three (or less or more) dimensional domains. The purpose of easi is to evaluate functions $f : \mathbb{R}^m \to \mathbb{R}^n$, which are described in a YAML configuration file. In grid-based simulation software, such as SeisSol, easi may be used to define models. In SeisSol, the function f maps every point in space (say x,y,z) to a vector of parameters (e.g. , , and ), which define a rheological model. Here, over 5000 lines of model-specific Fortran code could be replaced with YAML files.

An easi model consists of only two components: Map and Filter. These components may be wired as a tree, e.g. as in the following figure:




The procedure is as follows: A point x,y,z enters the tree at the root. The first Map takes a 3-dimensional vector as input and returns a n-dimensional vector. The following Filters decide if they accept this n-dimensional vector or reject it (e.g. accept if it lies in a hypercube and reject otherwise). Then, the branch which accepts a vector is taken (a Map accepts everything by default). The final parameter vector is, in this example, always 3-dimensional as it contains the density and the Lamé parameters.

CHAPTER

ONE

# GETTING STARTED WITH EASI

easi is a library written in C++14. It needs to be compiled with Cmake.

## 1.1 Dependencies

Easi depends on the following three projects:

- yaml-cpp
- ASAGI
- ImpalaJIT

Only yaml-cpp (version 0.6.x) is a required dependency. It can be obtained, for example, from the package repositories of most Linux distributions or as a module on SuperMUC-NG.

## 1.2 Compilation

Before installing easi, you first need to install all dependencies. Next, clone the easi repository and create a separate build directory. The directory can be outside of the repository. This makes the compilation cleaner. Finally, execute the following lines:

This installs easi into your home directory.

## 1.3 Usage example

easi is configured via YAML configuration files. For example, such a configuration file could look like the following:

```
!Any
components:
  - !AxisAlignedCuboidalDomainFilter
    limits:
      x: [-100000, 100000]
      y: [-5000, 5000]
      z: [-100000, 10000]
    components:
      - !ConstantMap
        map:
          lambda: 1e10
```

(continues on next page)

(continued from previous page)

```
        mu: 2e10
        rho: 5000
- !LayeredModel
  map: !AffineMap
    matrix:
      z: [0, 0, 1]
    translation:
      z: 0
  interpolation: linear
  parameters: [rho, mu, lambda]
  nodes:
    -100.0:   [2300.0, 0.1766e10, 0.4999e10]
    -300.0:   [2300.0, 0.6936e10, 1.3872e10]
    -1000.0:  [2600.0, 1.3717e10, 1.8962e10]
    -3000.0:  [2700.0, 2.1168e10, 2.7891e10]
    -6000.0:  [2870.0, 3.1041e10, 3.8591e10]
    -31000.0: [3500.0, 3.9847e10, 4.3525e10]
    -50000.0: [3200.0, 6.4800e10, 6.5088e10]
```

Here, all points with y-coordinate inbetween -5 km and +5 km would be assigned constant model parameters. For all other points, a linear interpolation, depending on the z-coordinate is used.

## 1.4 Application example

The first step is always to create a model. Here, we may use the YAMLParser class which creates models from YAML configuration files.

```
1 easi::YAMLParser parser(3);
2 easi::Component* model = parser.parse("test.yaml");
```

The argument in YAMLParser's constructor is the dimension of the input vectors. Here, we take 3 as we want to query our model in a 3-dimensional space.

As a next step, we need to define a query, which defines the input vectors for which we want to evaluate our model. In the following example we add the points (1,2,3) and (2,3,4). Each point may have an additional group parameter, which may be used to distinguish points in an easi file.

```
easi::Query query(2,3);
query.x(0,0) = 1.0;
query.x(0,1) = 2.0;
query.x(0,2) = 3.0;
query.group(0) = 1;
query.x(1,0) = 2.0;
query.x(1,1) = 3.0;
query.x(1,2) = -4.0;
query.group(1) = 1;
```

We need to store the output vectors somewhere. For this purpose, we always need to supply an adapter, which connects the output vector with locations in memory. In our sample application, the output vector shall be stored as array of structs, and hence we use an ArrayOfStructsAdapter. (Note that additional adapters can be implemented by overriding the class ResultAdapter.)

```
struct ElasticMaterial {
  double lambda, mu, rho;
};

ElasticMaterial material[2];
easi::ArrayOfStructsAdapter<ElasticMaterial> adapter(material);
adapter.addBindingPoint("lambda", &ElasticMaterial::lambda);
adapter.addBindingPoint("mu",     &ElasticMaterial::mu);
adapter.addBindingPoint("rho",    &ElasticMaterial::rho);
```

Finally, a simple call to evaluate is sufficient, and the model should be deleted if is not required anymore.

```
model->evaluate(query, adapter);
delete model;
```

The whole sample code is listed in the following:

```
#include <iostream>
#include "easi/YAMLParser.h"
#include "easi/ResultAdapter.h"

struct ElasticMaterial {
  double lambda, mu, rho;
};

int main(int argc, char** argv)
{
  easi::Query query(2,3);
  query.x(0,0) = 1.0;
  query.x(0,1) = 2.0;
  query.x(0,2) = 3.0;
  query.group(0) = 1;
  query.x(1,0) = 2.0;
  query.x(1,1) = 3.0;
  query.x(1,2) = -4.0;
  query.group(1) = 1;

  easi::YAMLParser parser(3);
  easi::Component* model = parser.parse("test.yaml");

  ElasticMaterial material[2];
  easi::ArrayOfStructsAdapter<ElasticMaterial> adapter(material);
  adapter.addBindingPoint("lambda", &ElasticMaterial::lambda);
  adapter.addBindingPoint("mu",     &ElasticMaterial::mu);
  adapter.addBindingPoint("rho",    &ElasticMaterial::rho);

  model->evaluate(query, adapter);

  delete model;

  for (unsigned j = 0; j < 2; ++j) {
    std::cout << material[j].lambda << " " << material[j].mu << " " << material[j].rho <
↪< std::endl;
```

(continues on next page)

(continued from previous page)

```
  }

  return 0;
}
```

CHAPTER

TWO

# COMPONENTS

In the following is a list of all components currently available in easi.

Every component has a domain m and a codomain n which can be thought of as a function $f : \mathbb{R}^m \to \mathbb{R}^n$. That is, a component accepts vectors in $\mathbb{R}^m$ and passes vectors in $\mathbb{R}^n$ to its child components (or as a result). The dimensions are labeled and a childs input dimensions must match its parent's output dimensions.

## 2.1 Composite components

Each composite may have a sequence of child components. Composite itself is abstract and may not be instantiated. Maps and Filters are always composite, builders are not.

```
!ABSTRACT
components:
  - <Component>
  - <Component>
  - ...
```

example

Alternative for composites with a single child:

```
!ABSTRACT
components: <Component>
```

Remark: Composites must of at least one child component.

CHAPTER

# THREE

# MAPS

A map allows to map vectors from $\mathbb{R}^m$ to $\mathbb{R}^n$.

## 3.1 ConstantMap

Assigns a constant value, independent of position.

```
!ConstantMap
map:
  <dimension>: <double>
  <dimension>: <double>
...
```

**Domain** *inherited*

**Codomain** keys in map

**Example** 0_constant

## 3.2 IdentityMap

Does nothing in particular (same as !Any).

```
!IdentityMap
```

**Domain** *inherited*

**Codomain** *domain*

## 3.3 AffineMap

Implements the affine mapping $y = Ax + t$.

```
!AffineMap
matrix:
  <dimension>: [<double>, <double>, ...]
  <dimension>: [<double>, <double>, ...]
  ...
```

(continues on next page)

(continued from previous page)

```
translation:
  <dimension>: <double>
  <dimension>: <double>
  ...
```

**Domain** *inherited*

**Codomain** keys of matrix / translation

**Example** Say we have a row in matrix which reads "p: [a,b,c]", and a corresponding row in translation "p: d". Furthermore, assume rho, mu, and lambda are the input dimensions. Then $p = a \cdot \lambda + b \cdot \mu + c \cdot \rho + d$.

3_layered_linear

By convention, the input dimensions are ordered lexicographically (according to the ASCII code, i.e. first 0-9, then A-Z, and then a-z), hence the first entry in a matrix row corresponds to the the first input dimension in lexicographical order.

## 3.4 PolynomialMap

Assigns a value using a polynomial for every parameter.

```
!PolynomialMap
map:
  #              x^n,       ...,         x,         1
  <dimension>: [<double>, ..., <double>, <double>]
  <dimension>: [<double>, ..., <double>, <double>]
  ...
```

**Domain** *inherited* but may only be a one dimension

**Codomain** keys in map

**Example** 62_landers

## 3.5 FunctionMap

Implements a mapping described by an ImpalaJIT function.

```
!FunctionMap
map:
  <dimension>: <function_body>
  <dimension>: |
    <long_function_body>
  ...
```

**Domain** *inherited*

**Codomain** keys in map

**Example** Input dimensions are x,y,z. Then "p: return x * y * z;" yields $p = x \cdot y \cdot z$. (Note: Don't forget the return statement.)

5_function

The <function_body> must an be ImpalaJIT function (without surrounding curly braces). The function gets passed all input dimensions automatically.

**Known limitations:**

- No comments (// or /* */)
- No exponential notation (use pow(10.,3.) instead of 1e3)
- No 'else if' (use else { if () {}}).

## 3.6 ASAGI

Looks up values using ASAGI (with trilinear interpolation).

```
!ASAGI
file: <string>
parameters: [<parameter>,<parameter>,...]
var: <string>
interpolation: (nearest|linear)
```

**Domain** *inherited*

**Codomain** keys in parameters

**Example** 101_asagi

**file** Path to a NetCDF file that is compatible with ASAGI

**parameters** Parameters supplied by ASAGI in order of appearance in the NetCDF file

**var** The NetCDF variable which holds the data (default: data)

**interpolation** Choose between nearest neighbour and linear interpolation (default: linear)

## 3.7 SCECFile

Looks up fault parameters in SCEC stress file (as in http://scecdata.usc.edu/cvws/download/tpv16/TPV16_17_Description_v03.pdf).

```
!SCECFile
file: <string>
interpolation: (nearest|linear)
```

**Domain** *inherited*, must be 2D

**Codomain** cohesion, d_c, forced_rupture_time, mu_d, mu_s, s_dip, s_normal, s_strike

**Example** example

**file** Path to a SCEC stress file

**interpolation** Choose between nearest neighbour and linear interpolation (default: linear)

## 3.8 EvalModel

Provides values by evaluating another easi tree.

```
!EvalModel
parameters: [<parameter>,<parameter>,...]
model: <component>
... # specify easi tree
components: <component>
... # components receive output of model as input
```

**Domain** *inherited*

**Codomain** keys of parameters

**Example** 120_initial_stress: [b_xx, b_yy, b_zz, b_xy, b_yz, b_xz] are defined through the component "!STRESS_STR_DIP_SLIP_AM", which depends itself on several parameters (mu_d, mu_s, etc). One of these parameter "strike" is set to vary spatially through an "!AffineMap". "!EvalModel" allows to evaluate this intermediate variable before executing the "!STRESS_STR_DIP_SLIP_AM" component.

## 3.9 OptimalStress

This function allows computing the stress which would result in faulting in the rake direction on the optimally oriented plane defined by strike and dip angles (this can be only a virtual plane if such optimal orientation does not correspond to any segment of the fault system). The principal stress magnitudes are prescribed by the relative prestress ratio R (where $R = 1/(1 + S)$), the effective confining stress (effectiveConfiningStress $= Tr(sii)/3$) and the stress shape ratio $s2ratio = (s_2 - s_3)/(s_1 - s_3)$, where $s_1 > s_2 > s_3$ are the principal stress magnitudes, following the procedure described in Ulrich et al. (2019), methods section 'Initial Stress'. To prescribe R, static and dynamic friction (mu_s and mu_d) as well as cohesion are required.

```
components: !OptimalStress
  constants:
    mu_d:      <double>
    mu_s:      <double>
    strike:    <double>
    dip:       <double>
    rake:      <double>
    cohesion:  <double>
    s2ratio:   <double>
    R:         <double>
    effectiveConfiningStress: <double>
```

**Domain** *inherited*

**Codomain** stress components (s_xx, s_yy, s_zz, s_xy, s_yz, and s_xz)

## 3.10 AndersonianStress

This function allows computing Andersonian stresses (for which one principal axis of the stress tensor is vertical).

The principal stress orientations are defined by SH_max (measured from North, positive eastwards), the direction of maximum horizontal compressive stress.

S_v defines which of the principal stresses $s_i$ is vertical where $s_1 > s_2 > s_3$. S_v = 1, 2 or 3 should be used if the vertical principal stress is the maximum, intermediate or minimum compressive stress. Assuming mu_d=0.6, S_v = 1 favours normal faulting on a 60° dipping fault plane striking SH_max, S_v = 2 favours strike-slip faulting on a vertical fault plane making an angle of 30° with SH_max and S_v = 3 favours reverse faulting on a 30° dipping fault plane striking SH_max.

The principal stress magnitudes are prescribed by the relative fault strength S (related to the relative prestress ratio R by $R = 1/(1 + S)$), the vertical stress sig_zz and the stress shape ratio $s2ratio = (s_2 - s_3)/(s_1 - s_3)$, where $s_1 > s_2 > s_3$ are the principal stress magnitudes, following the procedure described in Ulrich et al. (2019), methods section 'Initial Stress'. To prescribe S, static and dynamic friction (mu_s and mu_d) as well as cohesion are required.

```
components: !AndersonianStress
  constants:
    mu_d:      <double>
    mu_s:      <double>
    SH_max:    <double>
    S_v:       <int (1,2 or 3)>
    cohesion:  <double>
    s2ratio:   <double>
    S:         <double>
    sig_zz:    <double>
```

**Domain** *inherited*

**Codomain** stress components (s_xx, s_yy, s_zz, s_xy, s_yz, and s_xz)

## 3.11 STRESS_STR_DIP_SLIP_AM (deprecated)

This routine is now replaced by the more complete and exact 'OptimalStress' routine. It is nevertheless preserved in the code for being able to run the exact setup we use for the Sumatra SC paper (Uphoff et al., 2017). It is mostly similar with the 'OptimalStress' routine, but instead of a rake parameter, the direction of slip can only be pure strike-slip and pure dip-slip faulting (depending on the parameter DipSlipFaulting). In this routine the s_zz component of the stress tensor is prescribed (and not the confining stress tr(sii)/3) as in 'OptimalStress'.

```
components: !STRESS_STR_DIP_SLIP_AM
  constants:
    mu_d:      <double>
    mu_s:      <double>
    strike:    <double>
    dip:       <double>
    DipSlipFaulting: <double> (0 or 1)
    cohesion:  <double>
    s2ratio:   <double>
```

**Domain** *inherited*

**Codomain** stress components (s_xx, s_yy, s_zz, s_xy, s_yz, and s_xz)

**Example** 120_initial_stress

## 3.12 SpecialMap

Evaluates application-defined functions.

```
!<registered-name>
constants:
  <parameter>: <double>
  <parameter>: <double>
  ...
```

**Domain** *inherited* without constant parameters

**Codomain** user-defined

**Example** We want to create a function which takes three input parameters and supplies two output parameters:

```
#include "easi/util/MagicStruct.h"

struct Special {
  struct in {
    double i1, i2, i3;
  };
  in i;

  struct out {
    double o1, o2;
  };
  out o;

  inline void evaluate() {
      o.o1 = exp(i.i1) + i.i2;
      o.o2 = i.i3 * o.o1;
  }
};

SELF_AWARE_STRUCT(Special::in, i1, i2, i3)
SELF_AWARE_STRUCT(Special::out, o1, o2)
```

Register this file with the parser:

```
easi::YAMLParser parser(3);
parser.registerSpecial<Special>("!Special");
```

And use it in the following way, e.g.:

```
!Special
constants:
  i2: 3.0
```

The domain of !Special is now i1, i3 and the codomain is o1, o2. i2 is constant and has the value 3.

CHAPTER

FOUR

# FILTERS

Filters accept only a subsets of points and allows for the spatial partitioning of models.

## 4.1 Any

Any mostly serves as a root node and accepts every point and every group.

```
!Any
```

**Domain** *inherited*

**Codomain** *domain*

**Example** 1_groups

## 4.2 AxisAlignedCuboidalDomainFilter

Accepts only points inside an axis-aligned bounding box, i.e. when $l_x \leq x \leq u_x$ and $l_y \leq y \leq u_y$ and ...

```
!AxisAlignedCuboidalDomainFilter
limits:
  <dimension>: [<double>, <double>] # [l_x, u_x]
  <dimension>: [<double>, <double>] # [l_y, u_y]
  ...
```

**Domain** *codomain*

**Codomain** keys of limits

**Example** f_103_scec

## 4.3 SphericalDomainFilter

Accepts only points inside a sphere, i.e. when $\|x - c\| \leq r$.

```
!SphericalDomainFilter
radius: <double>
centre:
  <dimension>: <double>
  <dimension>: <double>
  ...
```

**Domain** *codomain*

**Codomain** keys of centre

## 4.4 GroupFilter

Accepts only points belonging to a set of groups.

```
!GroupFilter
groups: [<int>,<int>,...]
```

**Domain** *inherited*

**Codomain** *domain*

**Example** 120_sumatra

## 4.5 Switch

Can be used to use select a component based on the requested parameters.

```
!Switch
[<parameter>,<parameter>,...]: <component>
[<parameter>,<parameter>,...]: <component>
...
```

**Domain** *inherited*

**Codomain** *domain*

**Example** 120_sumatra: [mu_d, mu_s, d_c] are defined with a !ConstantMap and [cohesion, forced_rupture_time] are defined with a !FunctionMap.

The component on the right-hand side of the colon specifies a sub-model for the parameters on the left-hand side of the colon. The parameter lists must not intersect, as otherwise the sub-model, which shall be evaluated for a parameter, would not be uniquely defined.

CHAPTER

FIVE

# BUILDERS

Builders are not components by itself, but build a subtree using available components.

## 5.1 LayeredModel

Defines parameters at nodes, which are interpolated inbetween nodes.

```
!LayeredModel
map: <map> # Mapping to 1D (root component)
interpolation: (lower|upper|linear)
parameters: [<dimension>, <dimension>, ...] # order of dimension
nodes:
  <double>: [<double>, <double>, ...] # key: node position, value: dimension values
  <double>: [<double>, <double>, ...]
  ...
```

**Domain** *inherited*, must be 1D

**Codomain** length of coefficients sequence

**Interpolation methods**

**lower** Take the value of the lower (smaller) node

**upper** Take the value of the upper (larger) node

**linear** Linear interpolation

**Example** 3_layered_linear

## 5.2 Include

Includes another configuration file.

```
!Include <filename>
```

**Example** f_120_sumatra

CHAPTER

SIX

# GLOSSARY

**Domain and codomain** The set of all permitted inputs to a given function is called the domain of the function, while the set of permissible outputs is called the codomain.

**Map** A map transforms a m-dimensional input vector into an n-dimensional output vector.

**Filter** A filter is a boolean function which either accepts or rejects input vectors.

**Composite components** A composite component is a component with attached components (that is, a composite is the root of a tree). Basically, it means that several components can be plugged to process a chain of operations. Maps and Filters are composite components, Builders are not.