# Aetna Documentation

***Release 0.1.0***

**Igor Lobanov**

March 26, 2012

# Contents


**1 Introduction to Aetna project** **3**
- 1.1 About . . . 3
- 1.2 Features . . . 3
- 1.3 Limitations . . . 4
- 1.4 Further goals . . . 4
- 1.5 Disclaimer . . . 4
- 1.6 Developers . . . 4

**2 Other software implementing finite differences** **5**

**3 Installation manual** **7**
- 3.1 Getting Aetna . . . 7
- 3.2 Requirements . . . 7
- 3.3 Installation . . . 7
- 3.4 Running Aetna . . . 7

**4 First steps using Aetna project** **9**
- 4.1 Solution of Stokes and continuity equations . . . 9

**5 Reference manual** **13**
- 5.1 List of modules . . . 13

**6 Indices and tables** **67**

**Python Module Index** **69**

**Index** **71**

Contents:

CHAPTER 1

# Introduction to Aetna project

## 1.1 About

The Aetna project is a numerical library, written in Python, which ultimate goal is to solve PDE's. In contrast to most libraries of such kind, it uses finite difference discretization. As auxiliary tools it contains, in particular:

- Algebra of scalar fields enhanced with various differentiation approximations, boundary conditions support and more.
- A powerful tree object capable of storing direct sums of spaces, tensors and implementing operations on them.
- Classes for linear operators on scalar and tensor fields, which allows to define them and manipulate with them almost in mathematical notation.
- Classes for solution of linear systems of equations by Jacobi, multigrid and other methods.
- A unique solver based on Woodbury formula for solution of PDE with high contrasts in coefficients.
- Modules for visualization and VTK export.

## 1.2 Features

- Automatic discretization of PDE using finite differences.
- Almost mathematical syntax for operations on tensor fields and operators on them.
- Support of arbitrary dimension and wide range of boundary conditions.

## 1.3 Limitations

- Only rectangular or box grids are supported. Moreover, at the moment only regular grids are implemented.
- The code is completely unoptimized, hence the performance sometimes can be very bad. Low level operations are done by NumPy, hence are fast enough. However, it seems that calculations cam be several times faster.

## 1.4 Further goals

- Implement wider range of operators and solvers.
- Optimize code and implement GPU calculations.
- Implement marker-in-cell method.
- Make a GUI.

## 1.5 Disclaimer

The project is in early development stage, and everything should be considered as work in progress and subject to change.

## 1.6 Developers

- Igor Lobanov lobanov.igor@gmail.com

CHAPTER 2

# Other software implementing finite differences

http://code.google.com/p/fdtl/

CHAPTER 3

# Installation manual

## 3.1 Getting Aetna

The sources are on BitBucket. The repository is managed by Mercurial. To clone the repository execute:

```
hg clone https://dhwty@bitbucket.org/dhwty/aetna
```

## 3.2 Requirements

The Aetna projects is written in Python 2.x. NumPy package is used for linear algebra on N-dimensional arrays. Data is stored in VTK v2.0 files using PyVTK library. The visualization is done on top of matplotlib.pyplot.

The code was tested on an Intel Core machine running Ubuntu 11.10 with Python 2.7.2+ and NumPy 1.5.1 installed.

## 3.3 Installation

At the current state there are no special steps of installation. Just copy/unpack/clone the sources to a directory.

## 3.4 Running Aetna

Change working directory to the directory, where the sources were unpacked (the directory ends on <install_dir>/aetna/src/). For an interactive session run IPython:

```
ipython --pylab
```

Import all modules executing:

```
from aetna import *
```

See *First steps using Aetna project* for basic usage.

CHAPTER 4

# First steps using Aetna project

Change working directory to the directory, where the sources were unpacked (the directory ends on <install_dir>/aetna/src/). For an interactive session run IPython:

```
ipython --pylab
```

Import all modules executing:

```
from aetna import *
```

## 4.1 Solution of Stokes and continuity equations

There are several functions defined in module `main` demonstrating usage of the library for solution of Stokes and continuity equations. The simplest function `main.runCase()` solves the equations using Arrow-Hurwitz (a variant of Jacobi method). The first argument of `main.runCase()` is a description of the system including density, viscosity, external force, boundary conditions and so on. The description is returned by functions which name starts from prefix case*. Namely,

- `main.caseAnalytic()` describes Solcx problem,
- `main.casePixel()` describes problem with viscosity jump at one cell, and so on.

E.g. to solve Solcx problem on the grid 34x34 (32x32 internal points and boundary) with viscosity contrast 10 and desired residual 0.01 execute:

```
main.runCase(main.caseAnalytic(2**5+2, eta_B=10), eps=0.01)
```

During the execution you will see graph of residue with respect to iteration number and plot of current approximation of solution. After the execution you will see the following strings, describing parameters of the case and of the solver, number of floating point operations and number of Arrow-Hurwitz iterations made to obtain the solution:

```
solcx34etaA1.0etaB10c0.5nx2nz1eps0.01relax0.1.ArrowHurwitz.vtk
fpo : 1119872159
ArrowHurwitz : 17837
```

The VTK v2.0 file with the obtained solution will be stored in the current directory. To see early obained solution execute:

```
main.show('solcx34etaA1.0etaB10c0.5nx2nz1eps0.01relax0.1.ArrowHurwitz.vtk')
```

To compare the obtained solution with e.g. exact one, execute:

```
main.compare('exact.vtk', 'solcx34etaA1.0etaB10c0.5nx2nz1eps0.01relax0.1.ArrowHu
```

If you want, you can set relaxation parameters manualy as well as control other parameters, see `main.runCase()`:

```
main.runCase(main.caseAnalytic(2**5+2, eta_B=10), relax=.9, eps=0.01)

solcx34etaA1.0etaB10c0.5nx2nz1eps0.01relax0.7.ArrowHurwitz.vtk
fpo : 160089895
ArrowHurwitz : 2548
```

To use more advanced multigrid solver, use `main.runMulti()`:

```
main.runMulti(main.caseAnalytic(2**5+2, eta_B=10), relax=tree.fromList([.6,.2]),

solcx34etaA1.0etaB10c0.5nx2nz1eps0.01relax{0.6, 0.2}depth3gran3.Multi.vtk
fpo : 32225945
Multi : 69
ArrowHurwitz : 600
```

Here we used different relaxation parameters for velocity (0.6) and pressure (0.2). We used 3 grids (`depth`) and made 3 iterations on every grid (`gran`). New line in output shows number of V-cycles (Multi).

The library includes rarely used method, which can be preferable for localized jumps of viscosity. The method is based on Woodbery formula and corresponding function called for historical reasons `main.runKrein()`:

```
main.runKrein(main.casePixel(2**5+2, eta_B=10), eps=1e-2)

pixel34etaA1.0etaB10nx2nz1eps0.01relax0.7.Krein.vtk
Perturbation rank  6
fpo : 379368523
ArrowHurwitz : 6247
```

Now combining all the methods, we use Woodbury formula together with multigrid solver:

```
rlx = [tree.fromList([.7,.3])]*3
main.runKreinMulti(main.casePixel(2**5+2, eta_B=10), relax=rlx, depth=3, gran=3,

pixel34etaA1.0etaB10nx2nz1eps0.01relax[{0.7, 0.3}, {0.7, 0.3}, {0.7, 0.3}]depth3
Perturbation rank  6
fpo : 57555457
Multi : 126
ArrowHurwitz : 1134
```

Here `rlx` is a list of relaxation parameters for every grid. Other parameters are identical to `main.runMulti()`.

Mentioned above functions are just examples, and there are lot of things can be tweaked. Please see `main` for further examples, and feel free to modify them.

CHAPTER 5

# Reference manual

The package Aetna consists of the following parts. The module `field` provides class `Grid` describing regular grids in N-dimensional spaces, class `F` doing algebra on scalar fields on regular grids, and class `BC` defining boundary conditions.

## 5.1 List of modules

### 5.1.1 field

The module contains definitions of grids, scalar fields on grids, and boundary conditions.

At the moment there is only definition of regular grid `Grid`.

Scalar fields are represented by only class `F`, which stores values of a field at a grid vertices.

Boundary conditions are defined by ancestors of class `BC`. At the moment there are two such conditions: `Dirichlet` and `Neumann`. Free-slip conditions for vector fields are generated by function `toper.freeSlip()`.

**class** `field.`**`BC`**

The class represents boundary conditions on a grid. Normally class `F` is used to store values in internal points of a domain, to obtain values on whole domain including boundary one should call `get()`.

The class `BC` does not extend a field, and corresponds to missing boundary conditions. Practically `Dirichlet` and `Neumann` are useful.

There is no need to create new objects of this type, an exemplar of the object is stored in `field.ubc`.

**`get`** $(f, a, k)$

Returns continuation of a scalar field on larger grid according to boundary conditions.

**Note:** The return is only values at points lying outside grid of input argument. To get field on the whole expanded domain, use `F.extend()`.

---

**Parameters**

- **f** (*F*) – Scalar field to be continued
- **a** (*integer*) – Axis to continue along
- **k** – If positive, the continuation is to boundary with larger coordinate (right boundary), otherwise, with smaller one (left boundary)

**Return type** numpy.array

**matrix**(*a*, *k*, *before*, *after*)

Compute matrix elements of the continuation. The method is used by `foper.FExt`.

**Parameters**

- **a** (*integer*) – Axis to continue along
- **k** – If positive, the continuation is to boundary with larger coordinate (right boundary), otherwise, with smaller one (left boundary)
- **before** (*Grid*) – Grid to continue from
- **after** (*Grid*) – Grid to continue onto

**Return type** The same format as used by `foper.FO.matrix()`.

**order**

Difference between size of extended and original grid, e.g. Dirichlet and Nuemann bounday conditions have order 1.

**class** `field.`**Dirichlet**

Dirichlet boundary conditions f(x)=0. See `BC` for class methods.

There is no need to create new objects of this type, an exemplar of the object is stored in `field.dbc`.

**class** `field.`**F**(*X*, *G*)

Storage for scalar fields on regular grids. The field is represented by its values at the grid vertexes. The values are stored in numpy.Array accessed by `F.data`. The grid is available as `F.grid`.

Common arithmetic operations are defined, in particular, addition, subtraction, unary minus, multiplication, division and power are defined pointwise. If one of the operands is scalar, the operation is applied to every element of the array.

Fields can be compared precisely using == operator, or compared with given precision using `almast_eq()`.

All floating operations are counted and appended to the log using `log.add()` with key 'fpo'.

The field can be differentiated using `df()`, `db()`.

The field can be restricted to a smaller grid using `restrict()` or continued by boundary conditions to large one using `extend()`.

Auxiliary functions for common field constructions are provided: `zeros()`, `ones()`, `fromFunction()`, `fromArray()`.

```
>>> G = Grid((4,5), H=1, O=(0, 0))
>>> A = random(G)
>>> A
<F: Size 4x5 Origin (0, 0) Spacing (1, 1)>
>>> A+A==2*A
True
>>> Z = fromArray(np.zeros((4,5)))
>>> A-A==Z
True
>>> B = fromFunction(lambda x,y: x*y, G)
>>> B
<F: Size 4x5 Origin (0, 0) Spacing (1, 1)>
>>> print B
[[ 0  0  0  0  0]
 [ 0  1  2  3  4]
 [ 0  2  4  6  8]
 [ 0  3  6  9 12]]
>>> print B.restrict(Grid((2,2)))
[[0 0]
 [0 1]]
>>> C = fromFunction(lambda x,y: x+y, G)
>>> (B+C)**2==B**2+2*B*C+C**2
True
>>> print B+B.translate([1,0])
[[ 0  1  2  3  4]
 [ 0  3  6  9 12]
 [ 0  5 10 15 20]]
>>> (B+1e-11).almost_eq(B)
True
>>> G = Grid((4,4), H=(.2, .5))
>>> X = random(G)
>>> Y = random(G)
>>> XY1 = (X*Y).df(1)
>>> XY2 = X.df(1)*Y.shift((0,-1))+X*Y.df(1)
>>> XY3 = X.df(1)*Y+X.shift((0,-1))*Y.df(1)
>>> XY1.almost_eq(XY2)
True
>>> XY1.almost_eq(XY3)
True
```

**__init__**(*X*, *G*)

Create scalar field on a given grid with given values.

**Parameters**

- **X** (*numpy.array*) – Values of the field at vertexes
- **G** (*Grid*) – A grid where the field defined.

**__eq__**(*other*)
Two fields are equals, iff # their grids coincides, # they have same values at all vertexes.

**__getitem__**(*key*)
Elements of the grid can be accessed using square brackets, where indexes are indexes in the array `F.data`, not vertexes coordinates. In fact, the following are equivalent:

```
Grid.data[i]
Grid[i]
```

**Parameters** **key** (*Any index that numpy.array understand*) – Indexes of vertexes

**Return type** number

**__setitem__**(*key*, *val*)
The following are equivalent:

Grid[i] = v Grid.data[i] = v

**almost_eq**(*other*, *eps=1e-10*)
Compares grids up to desired precision.

**Parameters**

- **other** (*field.Grid*) – A grid.
- **eps** (*a positive number*) – Desired precision

**Return type** bool

**data**
Values at vertexes.

**Return type** numpy.array

**db**(*a*)
Return backward finite difference along given axis.

**Parameters** **a** (*integer*) – Axis to differentiate along

**Return type** field.F

**df**(*a*)
Return forward finite difference along given axis.

**Parameters** **a** (*integer*) – Axis to differentiate along

**Return type** field.F

**extend**(*bc*)
Return scalar field on larger domain where new values are deduced using boundary conditions.

**Parameters** **bc** (*boundary conditions for the grid in compact or complete form*) – Boundary conditions

**Seealso** `parseBC()`, `BC.get()`

**grid**
Grid of the domain of the field.

**Return type** field.Grid

**imap**(*other*, *fun*, *fpo=1*, *throw=True*)
Return scalar field obtained by application of given function of two arguments to elements of given fields with equal coordinates. Grid of the answer is intersection of arguments' grids.

Functions should accept numpy.array as argument.

**Parameters**

- **other** (*field.F*) – One of the arguments
- **fun** (*callable*) – A function of two arguments
- **fpo** (*integer*) – number of floating point operations to compute the function at one point
- **throw** (*bool*) – If True, throw `NotImplementedError`, otherwise return `NotImplemented`

**Return type** field.F

**map**(*fun*, *fpo=1*)
Return scalar field obtained by application of given function to the field at every vertex. Function should accept numpy.array as argument.

**Parameters**

- **fun** (*callable*) – Function to apply
- **fpo** (*integer*) – number of floating point operations to compute the function at one point

**restrict**(*grid*)
Construct restriction of the field onto the given grid. On restrictions onto the grid, see `Grid.common()`.

**Parameters** **grid** (*field.Grid*) – A grid.

**Return type** field.F

**shift**(*vec*)
Return the field with the origin shifted to given number of vertexes.

**Seealso** `Grid.shift()`

**Parameters** **vec** (*array-like object of integers*) – A vector to translate the field

**Return type** field.F

**sqnorm**()

**Returns** Square of l-squared norm

**translate**(*vec*)

Return the field with the origin translated by a vector.

**Seealso** `Grid.translate()`

**Parameters** **vec** (*array-like object of real numbers*) – A vector to translate the field

**Return type** field.F

**class** field.**Grid**(*shape=()*, *H=1.0*, *O=0.0*)

This class describes regular N-dimensional grid. The grid is defined by its origin `Grid.O`, size `shape` and spacing `shape`.

```
>>> G = Grid((4,5), H=1, O=(1, 0))
>>> G
<Grid: Size 4x5 Origin (1, 0) Spacing (1, 1)>
>>> G.dim
2
>>> G.low
array([1, 0])
>>> G.high
array([4, 4])
>>> G.box([2,2], [3,3])
(slice(1, 3, None), slice(2, 4, None))
>>> G.common(G.translate([0,1]))
(array([1, 1]), array([4, 4]))
>>> G.translate([0,1]).translate([1,0])==G.translate([1,1])
True
>>> G.mesh()
(array([[1, 1, 1, 1, 1],
       [2, 2, 2, 2, 2],
       [3, 3, 3, 3, 3],
       [4, 4, 4, 4, 4]]), array([[0, 1, 2, 3, 4],
       [0, 1, 2, 3, 4],
       [0, 1, 2, 3, 4],
       [0, 1, 2, 3, 4]]))
>>> G*G
<Grid: Size 4x5 Origin (1, 0) Spacing (1, 1)>
```

**__init__**(*shape=()*, *H=1.0*, *O=0.0*)

Constructor of the regular grid. :param shape: size of the grid :param H: spacing :param O: origin :type shape: tuple of positive integers :type H: array-like object of positive numbers :type O: array-like object of real numbers

**__mul__**(*other*)
Return intersection of the grids.

---

**Note:** Grids must have same spacing.

---

**Parameters** **other** (*field.Grid*) – A grid.

**Return type** field.Grid

**box**(*low*, *high*)
Return indexes of the grid vertexes lying in the given box.

**Parameters**

- **low** (*array-like object of real numbers*) – Bottom-left corner
- **high** (*array-like object of real numbers*) – Top-right corner

**Return type** Tuple of indexes suitable for numpy.array indexing.

**common**(*other*)
Return tuple of the bottom-left and upper-right corners of the intersection of the grids.

---

**Note:** Grids must have same spacing.

---

**Parameters** **other** (*field.Grid*) – A grid

**Return type** 2-tuple of numpy.array

**copy**()
Create a copy of the grid. Since Grid object is mutable, you should use `copy()` instead of assignment, if you are going to change the copy.

**Return type** field.Grid

**dim**
Return dimension of the grid. Equivalent to:

```
len(self.shape)
```

**Return type** integer

**high**
Return top-right corner of the bounding box.

**Return type** numpy.Array

**low**
Return bottom-left corner of the bounding box.

**Return type** numpy.Array

**mesh**()
Return coordinates of all vertexes. An analog of numpy.meshgrid.

**Return type** tuple of numpy.array

**shift**(*vec*)
Return the grid translated to the given number of vertexes. A grid with origin at (0,0) and spacing (.5,.5) shifted to (-2,-2) has origin at (-1,-1).

**Parameters** **vec** (*array-like object of integers*) – Number of vertexes to translate the grid

```
>>> G = Grid((10,10), H=.5, O=1.)
>>> G.shift((-2, -2))==G.translate([-1,-1])
True
```

**translate**(*vec*)
Return the grid translated to a given vector. E.g. a grid with origin in (0,0) translated to (1,0) has origin (1,0).

**Parameters** **vec** (*array-line object of real values*) – Vector with dimension of the grid

**view**(*grid*)
Return indexes of vertexes lying on the the intersection of the current grid a given one.

---

**Note:** Grids may have distinct spacing, since only bounding box of the given grid is taken into account.

---

**Parameters** **grid** (*field.Grid*) – A grid.

**Return type** Tuple of indexes suitable for numpy.array indexing.

**class** field.**Neumann**
Dirichlet boundary conditions f(x)=0. See BC for class methods.

There is no need to create new objects of this type, an exemplar of the object is stored in field.nbc.

field.**fromArray**(*X*)
Convert given numpy.array to the scalar field with origin at zero and spacing equal to one.

**Parameters** **X** (*array-like object*) – An array of values

**Return type** field.F

field.**fromFunction**(*fun*, *G*)
Compute given function ob given grid and return obtained scalar field.

**Parameters**

- **fun** (*callable*) – Function to compute

- **grid** (*field.Grid*) – A grid to compute on.

---

**Note:** Given function must accept numpy.array.

---

`field.`**`ones`**(*grid*)
Return constant field on given grid having all values equals to one.

**Parameters** **grid** (*field.Grid*) – A grid

`field.`**`parseBC`**(*bc*, *grid*)
The function expands short description of boundary conditions for a scalar field. The complete boundary conditions has the form

<boundary conditions> = [<boundary along axis>] <boundary along axis> = (<bottom boundary>, <top boundary>) <bottom boundary> = <top boundary> = `BC`

Short description can be of the form

<boundary conditions> = `BC`

meaning that all conditions along all boundaries are the same, or of the form

<boundary conditions> = [<boundary along axis>] <boundary along axis> = `BC`

meaning, that left and right (bottom and top, and so on) boundary conditions coincide.

**Parameters**

- **bc** – A short description of boundary conditions
- **grid** (*Grid*) – Grid of the domain. Only `Grid.dim()` is taken into account.

**Return type** Boundary conditions in complete form.

`field.`**`random`**(*G*)
Return random scalar field on given grid.

**Parameters** **G** (*field.Grid*) – Underlying

**Return type** field.F

`field.`**`zeros`**(*grid*)
Return constant field on given grid having all values equals to zero.

**Parameters** **grid** (*field.Grid*) – A grid

### 5.1.2 foper

The module defines classes of linear operators on scalar fields `field.F`. Provided classes are best suited for finite differences. The base class is `FO`. The identity operator is defined by `FId`. The sum, difference and composition of operator are given by `FSum`, `FSub`, `FComp`. `FMulS` defines multiplication by a scalar. `FExt` defines extension over boundary. `FShift` is

translation operator. FD and FB define forward and backward finite differences. laplace() defines Laplace operator on a box. All classes can be used to calculate matrix elements of the operators.

**class** foper.**FB**(*axis*)

Backward finite difference. Descendant of FO. The operator is defined by:

```
FB(axis)(X) == X.db(axis)
```

```
>>> N = 5
>>> C = fi.fromArray(np.ones((N,N)))
>>> A = FB(0)
>>> O = fi.fromArray(np.zeros((N-1,N))).shift([1,0])
>>> A(C)==O
True
>>> B = FD(0)
>>> (A*B)(C)==(B*A)(C)
True
>>> (A*B).diag(C.grid)
-2.0
>>> A.diag(fi.Grid((N,N)))
1.0
>>> A(C) == FO.__call__(A, C)
True
```

**__init__**(*axis*)

Create backward finite difference along a given axis. :param int axis: Axis to differentiate along.

**image**(*grid*)

See FO.image(). Image of the finite difference is defined on the grid smaller than grid of a preimage. Namely, if shape of the grid of a preimage is (n0, n1, ...), then the grid of the image of backward finite difference along axis a is (n0, n1,.. na-1,..). The origin of the grid is advanced one step forward along the given axis.

**class** foper.**FD**(*axis*)

Forward finite difference. Descendant of FO. The operator is defined by:

```
FD(axis)(X) == X.fd(axis)
```

```
>>> N = 3
>>> X = fi.fromArray(np.ones((N,N)))
>>> A = FD(0)
>>> O = fi.fromArray(np.zeros((N-1,N)))
>>> A(X)==O
True
>>> A.diag(fi.Grid((N,N)))
-1.0
>>> A(X) == FO.__call__(A, X)
True
```

**__init__**(*axis*)

Create forward finite difference along a given axis. :param int axis: Axis to differ-

entiate along.

**image**(*grid*)
See FO.image(). Image of the finite difference is defined on the grid smaller than grid of a preimage. Namely, if shape of the grid of a preimage is (n0, n1, ...), then the grid of the image of forward finite difference along axis a is (n0, n1,.. na-1,..). The origin of the grid is not changed.

**class** foper.**FExt**(*bc*)
Extension operator. Descendant of FO. The operator is defined by:

```
FExt(bc)(X) == X.extend(bc)
```

where bc are boundary conditions.

```
>>> G = fi.Grid((2,2))
>>> bc = [fi.dbc, (fi.nbc, fi.dbc)]
>>> A = FExt(bc)
>>> A
<FExt[<Dirichlet>, (<Neumann>, <Dirichlet>)]>
>>> A.image(G)
<Grid: Size 4x4 Origin (-1.0, -1.0) Spacing (1.0, 1.0)>
>>> X = fi.fromArray(np.ones(G.shape))
>>> print A(X)
[[ 0.  0.  0.  0.]
 [ 1.  1.  1.  0.]
 [ 1.  1.  1.  0.]
 [ 0.  0.  0.  0.]]
>>> U = fi.ones(fi.Grid((4,4), O=(-1,-1)))
>>> B = (FD(1)*U*FB(0)+FD(0)*U*FB(1))*A
>>> B.image(G)
<Grid: Size 2x2 Origin (0.0, 0.0) Spacing (1.0, 1.0)>
>>> B.diag(G)
<F: Size 2x2 Origin (0.0, 0.0) Spacing (1.0, 1.0)>
>>> G = fi.Grid((3,))
>>> U = fi.ones(fi.Grid((5,), O=(-1,)))
>>> (FD(0)*U*FB(0)*FExt(fi.dbc)).diag(G)
<F: Size 3 Origin (0.0) Spacing (1.0)>
>>> (FD(0)*U*FB(0)*FExt(fi.dbc)).image(G)
<Grid: Size 3 Origin (0.0) Spacing (1.0)>
```

**__init__**(*bc*)
Extension over boundary operator.

**Parameters bc** (field.BC or an argument accepted by field.parseBC() or field.F.extend().) – Boundary conditions

**class** foper.**FId**
Identity operator. Descendant of FO. The operator preserves its argument.

```
>>> A = FId()
>>> G = fi.Grid((3, 3))
```

```
>>> X = fi.random(G)
>>> A(X)==X
True
>>> A(X)==FO.__call__(A, X)
True
>>> print A.diag(G)
1.0
```

**__init__**()
Create identity operator. Requires no arguments.

**class** foper.**FMatrix**(*matrix*)
A linear operator defined by its matrix.

```
>>> X = np.ones((4,2,3))
>>> A = FMatrix(X)
>>> G = fi.Grid((2,3))
>>> Y = fi.ones(G)
>>> print A.image(G)
Size 4 Origin (0.0) Spacing (1.0)
>>> print A(Y)
[ 6.  6.  6.  6.]
```

**__init__**(*matrix*)
Creates linear operator FO defined by its matrix.

**Parameters matrix** (The same dictionary as returned by FO.matrix()) – Matrix of the operator

**__call__**(*X*)
See FO.__call__().

---

**Note:** The operator can be applied only to fields having the same grid as the matrix passed to the constructor.

---

**image**(*grid*)
See FO.image().

---

**Note:** The grid for arguments of the operator is not adjustable, hence the grid should correspond to the matrix of the operator.

---

**class** foper.**FO**
The class implements a linear operator on a scalar field given by its discretization field.F.

Instances of the class are callable. Application of the instance to a scalar field returns value of the operator on the scalar field. The matrix of the operator is calculated by matrix(). The diagonal elements of the matrix are returned by diag().

The following algebra is provided for two instances of the class. Addition and subtraction are defined element-wise. Multiplication is defined as composition. For arithmetic op-

erations with other types see description of `__add__()`, `__sub__()`, `__mul__()`, `__neg__()`.

---

**Note:** Abstract class.

---

**`__call__`**(*X*)
Calculate value of the operator on a given scalar field.

**Parameters** **X** (`field.F`) – Argument of the operator

**Returns** Value of the operator

**Return type** `field.F`

---

**Note:** This class implements `FO.__call__()` using only matrix elements of the opeartor returned by `FO.matrix()`. The following invariant should be valid for all descendances A of `FO`:

```
A(X)==FO.__call__(A, X)
```

---

**`__add__`**(*s*, *o*)
If both arguments are of the type `FO`, then addition is defined element-wise. If one of the arguments is a scalar or a scalar field, the argument is treated as multiplication operator by this constant.

**`__sub__`**(*s*, *o*)
Return difference of two operators. Arguments are treaded as by `__add__()`.

**`__neg__`**()
Multiply the operator by -1.

**`arrowDiag`**(*grid*)
Do the same thing as `diag()` by substitute ones for zeros.

**`diag`**(*grid*)
Return diagonal matrix elements of the operator, if argument is defined on a given grid.

**Parameters** **grid** (`field.Grid`) – Grid for the argument

**Returns** Diagonal coefficients

**Return type** `field.F`

The method is equivalent to:

```
O.diag(G) == O.matrix(G)[0,.. 0]
```

**`image`**(*grid*)
Return grid for the image, if grid of preimage is given.

**Parameters** **grid** (`field.Grid`) – Grid for preimage

**Return type** `field.Grid`

**Returns** Grid for image

**matrix**(*grid*)

Return matrix elements of the operator, if argument is defined on a given grid.

**Parameters** **grid** (field.Grid) – Grid for the argument

**Returns** Dictionary of matrix elements.

**Return type** Dictionary indexed by tuples of shift vectors, containing matrix elements.

If the operator is the composition of the translation operator to `vec` elements of the grid and of the multiplication by `mul`, then:

```
A.matrix(G) == {vec: mul}
A(X) == mul*X.shift(vec)
```

**class** foper.**FOComp**(*A*, *B*)

Operators composition. Descendant of FO. The composition is defined by:

```
FOComp(A, B)(X) == A(B(X))
```

The following shortcut is defined in FO for two objects `A`, `B` of the class FO:

```
A*B == FOComp(A, B)
```

```
>>> G = fi.Grid((4,4))
>>> A = FD(1)
>>> X = fi.random(G)
>>> (A*A)(X)==A(A(X))
True
```

**__init__**(*A*, *B*)

Create composition of given operators:

```
(AB)(x)=A(B(x)).
```

**Parameters**

- **A** (FO) – An operator to apply second
- **B** (FO) – A operator to apply first

**class** foper.**FOMulS**(*C*)

Multiplication by scalar. Descendant of FO. The multiplication is defined by:

```
FOMulS(C)(X) == C*X
```

The following shortcut is defined in FO for a scalar `C`:

```
C*X == X*C == FOMulS(C)*X
```

```
>>> G = fi.Grid((3,3))
>>> A = FId()
>>> X = fi.ones(G)
```

```
>>> (2*A)(X)==2*A(X)
True
>>> (X*A)(X)==X*A(X)
True
>>> -A
((-1*)*<FId>)
>>> (-A).matrix(G)
{(0, 0): -1.0}
>>> (A*2)(X)==A(X)*2
True
>>> (A*X)(X)==A(X)*X
True
>>> (X*A).diag(G)
<F: Size 3x3 Origin (0.0, 0.0) Spacing (1.0, 1.0)>
>>> B = FOMulS(X)
>>> A(X) == FO.__call__(A, X)
True
>>> C = B*FShift((1,0))*B
>>> C.matrix(G)
{(1, 0): <F: Size 2x3 Origin (1.0, 0.0) Spacing (1.0, 1.0)>}
```

**__init__**(*C*)

Create opeartor of multiplication by a give scalar. :param C: A multiplier :type C: Either scalar or scalar field `field.F`

**image**(*grid*)

See `FO.grid()`. The image of the multiplication is defined on intersection of the domains of multipliers.

**class** `foper.`**FOSub**(*A*, *B*)

Difference of operators. Descendant of `FO`. The difference is defined by:

```
(A-B)(X) == A(X) - B(X)
```

The following shortcut is defined in `FO`:

```
A-B == FOSub(A, B)
```

```
>>> G = fi.Grid((3,3))
>>> A = FId()
>>> B = A-A
>>> X = fi.random(G)
>>> (A-A)(X)==A(X)-A(X)
True
>>> B(X)==FO.__call__(B, X)
True
```

**__init__**(*A*, *B*)

Create an operator defined as the difference of given operators. :param A: Minuend :param B: Subtrahend :type A: `FO` :type B: `FO`

**image**(*grid*)

See `FO.image()`. The image of the difference of operators is defined on the

intersection of the images of addenda.

**class** foper.**FOSum**(*A*, *B*)

Sum of the operators. Descendant of FO. The sum is defined by:

```
(A+B)(X) == A(X) + B(X)
```

The following shortcut is defined in FO:

```
A+B == FOSum(A, B)
```

```
>>> A = FId()
>>> X = fi.random(fi.Grid((3,3)))
>>> B = A+A
>>> B(X)==A(X)+A(X)
True
>>> B(X)==FO.__call__(B, X)
True
```

**__init__**(*A*, *B*)

Create an operator defined as the sum of the given operators. :param A: First addendum :param B: Second addendum :type A: FO :type B: FO

**image**(*grid*)

See FO.image(). The image of the sum of operators is defined on the intersection of the images of addenda.

**class** foper.**FShift**(*vec*)

Translation operator. Descendant of FO. The operator translates operand to a given number of vertexes of the lattice. The following invariant is valid:

```
FShift(vec)(X) == X.shift(vec)
```

```
>>> G = fi.Grid((3,3))
>>> A = FShift((-1,0))
>>> A.image(G) == G.shift((-1,0))
True
>>> X = fi.random(G)
>>> A(X) == FO.__call__(A, X)
True
>>> A(X) == X.shift((-1,0))
True
>>> (A*A).image(G) == G.shift((-2,0))
True
>>> A = FShift((-1,0))
>>> B = FD(0)
>>> N = fi.random(G) #fi.fromFunction(lambda x,y: x+y, G)
>>> C = B*N*B
>>> D = (N*B+N.df(0)*A)*B
>>> C(N).almost_eq(D(N))
True
>>> C.diag(N.grid).almost_eq(D.diag(N.grid))
True
```

```
>>> A = FShift((-1,0))-FId()
>>> B = FD(0)
>>> A(N).almost_eq(B(N))
True
>>> A.matrix(N.grid)
{(0, 0): -1.0, (-1, 0): 1}
>>> B.matrix(N.grid)
{(0, 0): -1.0, (-1, 0): 1.0}
```

**__init__**(*vec*)

Create operator translating the argument to given number of vertexes. :param vec: Number of vertexes to shift along each direction :type vec: Tuple of integer

foper.**addDict**(*x*, *y*)

Add two dictionary. If a key belongs to both addenda, then corresponding values are added. If akey belongs to only one addenda, the corresponding value is copied to the result. The result contains only keys belonging to either of the addenda.

```
>>> addDict({1: 1, 2: 3}, {2: -1, 3: 3})
{1: 1, 2: 2, 3: 3}
```

foper.**laplace**(*dim*, *bc=<Dirichlet>*)

Return Laplace operator on the box of a given dimension with a given boundary conditions.

**Parameters**

- **dim** (*int*) – Dimension of the box
- **bc** (must be accepted by `field.parseBC()`) – Boundary conditions

```
>>> G = fi.Grid((3,3))
>>> A = laplace(2, fi.nbc)
>>> A.image(G)
<Grid: Size 3x3 Origin (0.0, 0.0) Spacing (1.0, 1.0)>
>>> print A.diag(G)
[[-2. -3. -2.]
 [-3. -4. -3.]
 [-2. -3. -2.]]
```

### 5.1.3 graphics

Graphics output layer. Here vizualisation tools are collected.

At the moment, there are several functions repeating functionality of `matplotlib.pyplot`, namely `figure()`, `plot()`, `close()`, `show()`. The function `drawSV()` allows to plot two dimensional scalar and vector fields on the same picture.

The module set interactive mode on, that is pictures will be updated instantly making call of `show()` superfluous.

graphics.**close**(*num*)
Close figure created with figure().

**Parameters** **num** (*integer*) – id of figure

graphics.**drawSV**(*scalar=None*, *vector=None*, *title=''*)
Plot on current figure given scalar and vector fields, Only two dimensional fields are supported.

**Parameters**

- **scalar** (*field.F*) – A scalar field
- **vector** (*tree.T of field.F*) – A vector field in the form {{x-component}, {x-component}}
- **title** (*string*) – Title of the picture

graphics.**figure**(*num=None*)
Create new figure or switch to previously created figure.

**Seealso** matplotlib.pyplot.figure()

**Parameters** **num** (*integer or None*) – None to create new figure, or id of figure to switch to

**Returns** id of created or choosen figure

**Return type** integer

graphics.**plot**(*x*, *y*, *title=''*)
Clear current figure and plot given curves.

**Seealso** matplotlib.pyplot.plot()

**Parameters**

- **x** (*array-like*) – x-coordinates of curves
- **y** (*array-like*) – y-coordinates of curves
- **title** (*string*) – Title of the picture

graphics.**show**()
Show all figures, and stop execution until all figure will be closed.

### 5.1.4 log

This module provides counters. The counters are named bu strings. In particular, 'fpo' counter is used to count floating-point operations.

log.**add**(*key*, *value=1*)
Increment counter.

**Parameters**

- **key** (*string*) – Counter name

- **value** (*integer*) – Value to append to the counter

**Return type** None

`log.`**`let`**`(`*key*, *value*`)`
Set desired counter to the given value.

**Parameters**

- **key** (*string*) – Counter name
- **value** (*integer*) – Value to append to the counter

**Return type** None

`log.`**`reset`**`()`
Reset all statistics. Delet all counters. :rtype: None

`log.`**`show`**`()`
Print all collected statistics to stdout.

**Return type** None

### 5.1.5 main

This module contains examples of solutions of PDE using Aetna library. There two main examples: Poisson equation and system of Stokes and continuity equation.

Poisson equation is solved by function `testLaplace()` using multigird method.

The Stokes and continuity equations solved by functions run*. The following methods are used by the functions: `runCase()` - Jacobi method, `runMulti()` - multigrid method doing Jacobi iterations on each step, `runKrein()` - Woodbury formula with Jacobi method for intermediate problems, `runKreinMulti()` - Woodbury formula sing multigraid to solve intermediate problems.

Functions case* define test cases for the Stokes and continuity equations. The functions returns tuple (G, eta, bc, relax, name) where

G is product of density and external force, eta is viscosity, bc is boundary conditions, relax are relaxation coefficients for Jacobi method, name is a string describing configuration.

Auxiliary classes `Viz` and `VizS` are used for visualization of the solution progress.

Functions `writeVTK()` and `readVTK()` are used for input/output of Stokes-continuity solutions.

Function `show()` visualize previously stored solution.

Function `compare()` shows difference between two previously obtained solutions.

**class** `main.`**`Viz`**`(`*E=<class toper.TId at 0x42f1a10>*`)`
Visualize pair (V, P), where V is a vector field, P is a scalar fields. The class can be passed to solver.Solver.register to show current approximation of the solution.

**class** main.**VizS**(*E=<class toper.TId at 0x42f1a10>*)
Visualize scalar fields. The class can be passed to solver.Solver.register to show current approximation of the solution.

main.**caseAnalytic**(*n=31*, *eta_A=1.0*, *eta_B=100.0*, *c=0.5*, *nx=2*, *nz=1*)
Two dimensional test configuration on grid n x n (SolCx problem). viscosity eta = eta_A if x <= x_c, eta = eta_B if x > x_c, density rho = cos( nx.pi.x ).sin( nz.pi.y), external field f = (0,-1).

main.**casePixel**(*n=31*, *eta_A=1.0*, *eta_B=100.0*, *nx=2*, *nz=1*)
Two dimensional test configuration on grid n x n. viscosity eta = eta_B if x=y=1/2, eta = eta_A otherwise, density rho = cos( nx.pi.x ).sin( nz.pi.y), external field f = (0,-1).

main.**caseVortex**(*n=32*)
Two dimensional test configuration on grid (n+2)x(n+2). Density rho=1, viscosity eta = 1, and external force G(x,y) = (-y, x), Dirichlet boundary conditions.

main.**caseVortexFS**(*n=32*)
Two dimensional test configuration on grid (n+2)x(n+2). Density rho=1, viscosity eta = 1, and external force G(x,y) = (-y, x), free-slip boundary conditions.

main.**caseVortexPie**(*n=16*, *power=10*)
Two dimensional test configuration on grid (n+2)x(n+2). Density rho=1, viscosity eta = 1 for x<=1/2 and eta = power for x>1/2, external force G(x,y) = (-y, x), Dirichlet boundary conditions.

main.**compare**(*first*, *second*)
Visualize two solutions of the Stokes and continuity equations stored in the files first and second, and display mean square error.

main.**laplaceMultiSolver**(*grid*, *depth=3*, *eps=0.01*, *relax=0.9*, *gran=4*, *bc=<Dirichlet>*)
Returns solver object, which solves Poisson equation with bc boundary conditions using multigrid method on depth grids doing gran Jacobi iterations on every grid with relax relaxation parameters. The stop conditions are residual less than eps. VizS object is used to show progress of calculations.

main.**pie**(*x*, *x0*, *a*, *b*)
Returns array of the same dimension as x, containing a for x<=x0, and b otherwise.

main.**readVTK**(*filename*, *prefix='../output/'*)
Read pair (V, P) saved in the file filename by the function writeVTK. The result is stored in the tree{{{V0}, {V1}}, {P}}, where V0 and V1 are components of the vector field V, and P is a scalar field.

main.**reset**()
Reload all modules. Useful in ipython interactive session

main.**runCase**(*case*, *eps=0.01*, *relax=None*)
The function solves Stokes and contietaity equations for a test problem case using Jacobi method with relaxation parameters relax. The stop condition is residual less than eps.

main.**runKrein**(*case*, *eps=0.01*, *relax=0.7*)
The function solves Stokes and continuity equations for a test problem case using Wood-

bury formula (Krein solver) using Jacobi method with relax relaxation parameters to solve auxiliary problems. The stop condition is residual less than eps.

main.**runKreinMulti**(*case*, *eps=0.01*, *relax=None*, *depth=3*, *gran=4*)
The function solves Stokes and continuity equations for a test problem case using Woodbury formula (Krein solver) using multigrid method for auxiliary problems. The multigird uses depth grids, doing gran Jacobi iterations with relax relaxation parameters on each of them. The stop condition is residual less than eps.

main.**runMulti**(*case*, *eps=0.01*, *relax=None*, *depth=1*, *gran=4*)
The function solves Stokes and contietaity equations for a test problem case using the solver returned by stokesContMultiSolver. The meaning of parameters see in description of stokesContMultiSolver.

main.**show**(*filename*)
Visualize the solution of Stokes and continuity equations stored in the file filename.

main.**stokesCont1MultiSolver**(*grid*, *eta*, *depth=3*, *eps=0.01*, *relax=0.5*, *gran=4*, *bc=<Dirichlet>*)
Returns solver object, which solves the equation -eta*Delta_i(Nabla_i V_j+Nabla_j V_i)+Delta_jP=G_j, Nabla_i V_i = 0, on a grid grid with bc boundary conditions using multigrid method on depth grids doing gran Jacobi iterations on every grid with relax relaxation parameters. The stop conditions are residual less than eps. Viz object is used to show progress of calculations.

---

**Note:** The size of the grid must be a power of 2.

---

main.**stokesContMultiSolver**(*grid*, *eta*, *depth=3*, *eps=0.01*, *relax=0.5*, *gran=4*, *bc=<Dirichlet>*)
Returns solver object, which solves Stokes and contietaity equation with viscosity eta on a grid grid with bc boundary conditions using multigrid method on depth grids doing gran Jacobi iterations on every grid with relax relaxation parameters. The stop conditions are residual less than eps. Viz object is used to show progress of calculations.

---

**Note:** The size of the grid must be a power of 2.

---

main.**testLaplace**(*size=128*, *depth=5*, *eps=0.0001*, *relax=0.9*, *gran=20*)
Solve Poisson equation Laplace X = 1, on the grid size x size with Dirichlet boundary conditions using solver returned by function laplaceMultiSolver.

---

**Note:** The size of the grid must be a power of 2.

---

main.**writeVTK**(*filename*, *V*, *P*, *prefix='../output/'*)
Save vector field V and scalar field P to the VTK v2.0 file named filename. Only two-dimensional space is supported. Grid of P has to coincide with grid of V with dropped cells having smallest first or second coordinates.

### 5.1.6 ravel

The module contains definitons of mapping of vector fields to vector in C^N. Classes `FRavel` is an abstract class for the mapping. Class `TRavel` defines mappings from tensor fields using mappings from its scalar components. Class `FFlatten` maps scalar field as whole to vector in C^N. Class `FSelect` maps only selected vertices of scalar field to vectors in C^N. Auxilliary function `selectNonzero()` is used to obtain list of vertices where given scalar field does not vanish.

**class** `ravel.`**`FFlatten`**(*grid*)

Bijection between a scalar field on a given grid and C^n. Descendant of `FRavel`.

```
>>> G = fi.Grid((3,3))
>>> R = FFlatten(G)
>>> X = np.random.rand(R.rank)
>>> np.all(X == R.restrict(R.embed(X)))
True
```

**`__init__`**(*grid*)

**Parameters** **grid** (`field.Grid`) – A grid

**`embed`**(*X*)

See `FRavel.embed()`.

**`grid`**

See `FRavel.grid()`. The grid coincides with the one given to the constructor.

**`rank`**

See `FRavel.rank()`. The rank is exactly number of vertexes of the grid `grid()`.

**`restrict`**(*X*)

See `FRavel.restrict()`.

**class** `ravel.`**`FRavel`**

Bijection between a domain in `field.F` and C^n.

---

**Note:** Abstract class.

---

**`embed`**(*X*)

Injective mapping from C^n to the functions defined on the grid `grid()`.

**Parameters** **X** (`numpy.array`) – An element of C^n, n=:meth:*rank*

**Returns** Function on the grid `grid()`

**Return type** `field.F`

**`grid`**

**Returns** grid for the scalar field returned by `embed()`

**Return type** `field.Grid`

**rank**
Return rank of the mapping.

**Returns** Size of the array returned by `restrict()`

**Return type** int

**restrict**(*X*)
Inverse mapping to `embed()`. The following invariant is valid for every object `A` of type `FRavel`:

```
A.restrict(A.embed(X)) == X
```

**Parameters** **X** (`field.F`) – A function on the grid `grid()`

**Returns** A vector from C^n, n=:meth:*rank*

**Return type** one dimensional `numpy.array`

**submatrix**(*A*)
Calculate matrix of composition restrict*A*embed.

**Parameters** **A** (`foper.FO`) – An operator defined on `field.F` over the grid `grid()`

**Returns** Matrix of the composition restrict*A*embed

**Return type** two dimmensional `numpy.array`

The following variant should be valid for every mapping R of `FRavel` and every operator A of `foper.FO` defined on R.grid:

```
R.restrict(A(R.embed(X))) = numpy.dot(R.submatrix(A), X)
```

**class** `ravel.`**FSelect**(*grid*, *idx*)
Bijection between C^n and functions on fixed vertexes of a given grid. Descendant of `FRavel`. The object is returned by `selectNonzero()`.

```
>>> G = fi.Grid((2,3))
>>> R = FSelect(G,[(0,1),(1,2)])
>>> X = np.array([1,2])
>>> print R.embed(X)
[[ 0.  1.  0.]
 [ 0.  0.  2.]]
>>> np.all(X == R.restrict(R.embed(X)))
True
>>> R.submatrix(lambda g: 2*g)
array([[ 2.,  0.],
       [ 0.,  2.]])
>>> R = FSelect(G,[(0,0)])
>>> print R.embed(1)
[[ 1.  0.  0.]
 [ 0.  0.  0.]]
>>> R = FSelect(G, [])
>>> print R.embed([])
```

```
[[ 0.  0.  0.]
 [ 0.  0.  0.]]
```

**__init__**(*grid*, *idx*)

Create bijection between functions on given vertexes of given grid and C^n.

**Parameters**

- **grid** (`field.Grid`) – A grid
- **idx** (any `numpy` indexes) – Indexes of the vertexes as indexes of the array `field.F.data()`

---

**Note:** If `idx` is a list, then its elements are treated as indexes of elements of `field.F.data()` (differs from default `numpy` behavior).

---

**class** `ravel.`**TRavel**(*tree*)

Bijection between a tensor field and C^n. The mapping is a direct sum of mappings `FRavel` over all components of the tensor.

```
>>> G = fi.Grid((2, 2))
>>> R1 = FSelect(G, ([0],[1]))
>>> R2 = FSelect(G, ([1],[0]))
>>> T = tr.fromList([R1, [R2]])
>>> R = TRavel(T)
>>> R.offsets
{0, {1}}
>>> X = np.ones(R.rank)
>>> print R.embed(X)
{[[ 0.  1.]
 [ 0.  0.]], {[[ 0.  0.]
 [ 1.  0.]]}}
>>> np.all(X == R.restrict(R.embed(X)))
True
```

**__init__**(*tree*)

Create the mapping from a tensor field to C^n. The mapping is defined as the direct sum of operators `FRavel` passed in `tree` in the order specified by iterators `tree.T.map()` and `tree.T.mapReduce()`.

**Parameters** **tree** (`tree.T` of `FRavel`) – Tree of embedings of scalar fields to C^n

**embed**(*vec*)

Injective mapping from C^n to the tree of fields defined on the grids `grid()`.

**Parameters** **vec** (`numpy.array`) – An element of C^n, n=:meth:*rank*

**Returns** Tree of fields on the grids `grid()`

**Return type** `tree.T` of `field.F`

**grids**

**Returns** Tree of grids of the return of `embed()`

**Return type**

**tree.T** of `field.Grid`

**offsets**

**Returns** Tree of offsets of ranges of every mapping `FRavel.restrict()` in the result of `TRavel.restrit()`

**Return type** `tree.T` of int

**rank**

**Returns** Rank of `TRavel.restrict()`

**Return type** int

Rank of the embedding `TRavel` is equals to sum of ranks of all the addenda `FRavel`.

**restrict**(*tree*)

Inverse mapping to `embed()`. The following invariant is valid for every object `A` of type `TRavel`:

```
A.restrict(A.embed(X)) == X
```

**Parameters** **tree** – A tree of scalar fields

**Returns** A vector from C^n, n=:meth:*rank*

**Return type** one dimensional `numpy.array`

`ravel.`**selectNonzero**(*X*)

**Parameters** **X** (`field.F`) – A scalar field

**Returns** Bijection between C^n and vertexes, where the given scalar field does not vanish.

**Return type** `FSelect`

```
>>> F = fi.zeros(fi.Grid((2,3)))
>>> F[1,2] = 1
>>> R = selectNonzero(F)
>>> R.restrict(F)
array([ 1.])
>>> print R.embed(3)
[[ 0.  0.  0.]
 [ 0.  0.  3.]]
```

### 5.1.7 scale

The module provides mappings between finer and coarser grids. The mappings are enclosed to descendants of `scale.Scale`. Currently there two mappings implemented: bilinear fil-

tering scale.FBilinear, and nearest-neighbor interpolation scale.FDouble. Class scale.TScale can be used to extend the mappings to tensor fields.

**class** scale.**FBilinear**(*grid*, *bc*)

Descendant of Scale. Interpolation-restriction between grids having two-times more/less vertexes along each axes. Interpolation and restriction are done by bilinear filtration. The mappings preserve parity of every dimension of the grids.

```
>>> G = fi.Grid((3,4))
>>> F = fi.ones(G)
>>> print F
[[ 1.  1.  1.  1.]
 [ 1.  1.  1.  1.]
 [ 1.  1.  1.  1.]]
>>> S = FBilinear(G, fi.nbc)
>>> F1 = S.restrict(F)
>>> print F1
[[ 1.  1.]
 [ 1.  1.]]
>>> F2 = S.smooth(F1)
>>> print F2
[[ 1.   1.   1.   0.5]
 [ 1.   1.   1.   0.5]
 [ 1.   1.   1.   0.5]]
```

**coarseGrid**

See Scale.coarserGrid().

**finnerGrid**

See Scale.finnerGrid().

**indices**()

**Returns** Return list of pairs (index of a vertex of finer grid, weight of the vertex) where vertexes corresponds to the origin of the coarser grid

**Return type** list of pairs (tuple of ints, float)

**restrict**(*field*)

See Scale.restrict().

**smooth**(*field*)

See Scale.smooth().

**class** scale.**FDouble**(*grid*)

Descendant of Scale. Interpolation-restriction between grids having two-times more/less vertexes along each axes. Interpolation is done by nearest-neighbor interpolation. Restriction is the arithmetic mean. If number of vertexes is odd, the vertexes with largest coordinates are simply copied.

```
>>> G = fi.Grid((3,4))
>>> F = fi.fromFunction(lambda x,y: x+y, G)
>>> print F
[[ 0.  1.  2.  3.]
```

```
 [ 1.  2.  3.  4.]
 [ 2.  3.  4.  5.]]
>>> S = FDouble(G)
>>> F1 = S.restrict(F)
>>> print F1
[[ 1.   3. ]
 [ 2.5  4.5]]
>>> F2 = S.smooth(F1)
>>> print F2
[[ 1.   1.   3.   3. ]
 [ 1.   1.   3.   3. ]
 [ 2.5  2.5  4.5  4.5]]
```

**coarseGrid**
See `Scale.coarserGrid()`.

**finnerGrid**
See `Scale.finnerGrid()`.

**indices**()

**Returns** Indexes of finer grid corresponding to origin of the coarser grid

**Return type** list of tuples of ints

**restrict**(*field*)
See `Scale.restrict()`.

**smooth**(*field*)
See `Scale.smooth()`.

**class** `scale.`**Scale**
Mappings between coarser and finer grids.

---

**Note:** Abstract class.

---

**coarseGrid**

**Returns** Coarser grid

**Return type** `field.Grid`

**finnerGrid**

**Returns** Finer grid

**Return type** `field.Grid`

**restrict**(*field*)
Restrict given field to the coarser grid.

**Parameters** **field** (`field.F`) – A scalar field on the grid `Scalar.finnerGrid()`.

**Returns** Restriction to the grid `Scalar.coarsrGrid()`

**Return type** field.F

**smooth** (*field*)
Interpolate given field to the finer grid.

**Parameters** **field** (field.F) – A scalar field on the grid Scalar.coarserGrid().

**Returns** Interpolation to the grid Scalar.finnerGrid()

**Return type** field.F

**class** scale.**TScale** (*scale*, *grids=None*)
Mappings between tensor fields on coarser and finer grids. Almost identical to sale.Scale, however maps trees of scalar fields. These mappings are defined node-wise.

**__init__** (*scale*, *grids=None*)
Create mappings between tensor fields on coarser and finer grids, where fields are stored in tree.T. The image of either of the mappings is obtained by application nodes of the tree of mappings on scalar fields to corresponding nodes of source tree of scalar fields.

**Parameters**

- **scale** (Class scale.Scale or tree tree.T of objects scale.Scale.) – Mappings on scalar fields defining the mapping on nodes
- **grids** (tree.T of field.Grid) – If scale is a descendant of class scale.Scale (not object), the nodes of grids are passed to constructor of the class to create tree of objects

**coarseGrid**

**Returns** Tree of coarser grids

**Return type** tree.T of field.Grid

**finnerGrid**

**Returns** Tree of finner grids

**Return type** tree.T of field.Grid

**restrict** (*fields*)
Apply the restriction mapping passed to the constructor as argument scale to every node of fields.

**Parameters** **fields** (tree.T of field.F) – A tensor field

**Returns** Tree of restrictions to coarser grids

**Return type** tree.T of field.F

**smooth** (*fields*)
Apply the interpolation mapping passed to the constructor as argument scale to every node of fields.

**Parameters** **fields** (`tree.T` of `field.F`) – A tensor field

**Returns** Tree of interpolations to finer grids

**Return type** `tree.T` of `field.F`

### 5.1.8 solver

The module provides solvers for linear systems. The systems must be of the form L X = Y, where L is a linear operator from a linear space V to the same space V, and Y is an element of V. Technically l is a callable object, and Y is an object of a type with overloaded arithmetic operations. In particular, L can be of type `toper.TO` or `foper.FO`, and Y can be `field.F` or `tree.T`.

In most general form solver is defined by `Solver`. A solver based on Woodbury formula is implemented as `Krein`. `Iteration` provides a basic iterative solver. Preconditioned solvers are inherited from `Preconditioned`, in particular `Jacobi` implements Jacobi solver, and `ArrowHurwitz` implements Arrow-Hurwitz method. General multi-grid solver is enclosed in `Multi`.

Every solver has ability to monitor its activity. After every iteration the solvers call descendants of `Handler` previously registered with `Solver.register()`. There several predefined classes providing information on solution progress, namely `Reporter`, `Graph`, `GraphLog`, see also `main.Viz`, `main.VizS`. To display progress only after a period of time, use `Periodic`.

**class** `solver.`**`ArrowHurwitz`**(*L*, *shape*, *relax=1.0*)

Solve linear system by Arrow-Hurwitz method. Descendant of `solver.Iteration`.

**`pre`**(*R*)

The multiplication by relax/L.arrowDiag is used as precondition.

**class** `solver.`**`Graph`**

Callback object, descendant of `solver.Handler`, parent of classes working with graphics.

**`__init__`**()

Create a figure upon initialization. The figure handler is stored in `self.figure`.

**`__del__`**()

Close the figure opened by constructor.

**class** `solver.`**`GraphLog`**

Callback object, descendant of `Graph`. Display graph of residue with respect to number of iterations.

---

**Note:** Must be registered to `Solver.Iteration` object.

---

**`__init__`**()

Initialize graphics and data.

**class** `solver.`**`Handler`**

The class provides callback interface. Descendants of the class can be passed to `Solver.register()`, which call their `Handler.__call__()` method after every iteration. Sequences of handlers can be created calling `Handler.register()`.

The class can be instantiated, but does nothing. For something useful see the descendants `Periodic`, `Reported`, `Graph`, `GraphLog`.

**`__init__`**`()`

Create handler.

**`__call__`**(*obj=None*, *parent=None*)

Callback function.

**Parameters**

- **obj** (`Solver`) – Information on solution progress
- **parent** (`Handler`) – Caller

**`register`**(*fun*)

Register callback function to be executed by self.__call__. The `parent` argument will be set to `self`.

**Parameters** **fun** (`Handler`) – Callback object

**Returns** `self`

**Return type** `Handler`

**class** `solver.`**`Iteration`**

General iterative solver of a linear system

P L X = P R,

where P is a pre-condition, L is a linear operator. Descendant of `Solver`. The solver is applicable only if dom L = ran L. Depending of pre-condition, the following methods can be obtained:

•P = Id – summation of sum_j (-L)^j,

•P = diag(L)^{-1} – Jacobi method,

•P = theta * diag(L)^{-1} – Jacobi method with relaxation `Jacobi`.

The solution is obtained as limit of the sequence X_k, where

X_k=sum_{j=0}^infty (1-P L)^j P G,

or

X_0 = P G, X_{k+1} = (1 - P L) X_k + P G = X_k + P(G - L X_k) = X_k + P R.

The iterations converge, if |1 - P L|<1. Error can be estimated as

|X_k-X| < |P G| * |1 - P L|^{k+1} / (1 - |1 - P L|).

Number of performed iterations is stored in `self.iteration`, obtained residue in `self.residue`,current approximation of the solution ib `self.X`.

**__init__**()
Initialize solver. Descendants of the class must call the constructor and specify the operator `self.oper` of the system and a pre-condition `self.pre`. By default number of iterations is unbounded

self.maxiter = None

and desired residue is

self.target = 1e-10

**init**(*R*)
Initialize internal structures. Called by `solve()` before iterations start. The method sets first guess `self.X` and residue `self.residue`.

**iterate**()
Make one iteration. Must recalculate `self.X`, `self.residue`, and `self.eps`.

**oper**(*X*)
Calculate operator `self.oper` of solving system

self.pre(self.oper(X)) = self.pre(R)

**pre**(*X*)
Compute precondition.

**Parameters** **X** (type of argument `Y` of `solve()`) – An element of a vector space

**Returns** Value of precondition on `X`

**Return type** type of `X`

---

**Note:** Abstract method

---

**solve**(*R*)
Solve system

self.pre(self.oper(X)) = self.pre(R)

for X. The residue is stored in

self.residue = self.oper(X)-R.

Computations stop after `self.maxiter` iterations or if `self.residue` is less than `self.eps`.

**Parameters** **R** (*Element of a linear space*) – Right hand side

**Returns** A solution

**Return type** type of `R`

**class** solver.**Jacobi**(*L*, *shape*, *relax=1.0*)

Solve linear system by Jacobi method. Descendant of solver.Iteration.

```
>>> import field as fi; import foper as fo
>>> G = fi.Grid((3,3))
>>> L = (fo.FD(0)*fo.FB(0)+fo.FD(1)*fo.FB(1))*fo.FExt(fi.dbc)
>>> S = Jacobi(L, G)
>>> X = fi.ones(G)
>>> print S(X)
[[-0.6875 -0.875  -0.6875]
 [-0.875  -1.125  -0.875 ]
 [-0.6875 -0.875  -0.6875]]
```

**pre**(*R*)

The multiplication by relax/L.diag is used as precondition.

**class** solver.**Krein**(*I*, *R*, *B*)

Solve linear system (A+R.embed*R.restrict*B)X = Y for X using Woodbury formula X = [I-I*R.embed*(1+R.restrict*B*I*R.embed)^{-1}*R.restrict*B*I]Y, where I is inverse of A.

```
>>> n = 3
>>> G = fi.Grid((n,))
>>> A = np.random.rand(n, n)
>>> I = fo.FMatrix(np.linalg.inv(A))
>>> B = fo.FMatrix(np.random.rand(n, n))
>>> R = ra.FFlatten(G)
>>> X = np.random.rand(n)
>>> Y = fi.F(np.dot(A, X)+np.dot(B.M, X), G)
>>> S = Krein(I, R, B)
>>> S(Y).almost_eq(fi.F(X, G))
True
```

**__init__**(*I*, *R*, *B*)

Initialize solver setting operators in Woodbury formula X = [I-I*R.embed*(1+R.restrict*B*I*R.embed)^{-1}*R.restrict*B*I]Y

**Parameters**

- **I** (*callable*) – Inverse of unperturbed operator
- **R** (ravel.FRavel) – Extension/restriction operators
- **B** (*callable*) – Additive perturbation

**solve**(*Y*)

Solve system (A+R.embed*R.restrict*B)X = Y with given Y for X.

**Parameters** **Y** (must be accepted as argument by operators I and R passed to the constructor) – Right hand side

**Returns** Solution of the system

**Return type** type of Y

**class** solver.**Multi**(*solvers*, *scales*)
Iterative solver by multi-grid method.

**init**(*R*)
Initial guess is:: self.X = self.pre(R)

**iterate**()
Do one V-cycle.

**oper**(*X*)
Operator of the system coincides with the operator on finest grid.

**pre**(*X*)
The multi-solver can be used as a part of another iterative solver. In this case pre-condition `Multi.pre()` is pre-condition of the solver on the finest grid.

**class** solver.**Periodic**(*time=1*)
Callback object, which calls its sub-handlers periodically after given period of time. Desired interval is specified in constructor. Sub-handlers are registered by `Handler.register`.

**__init__**(*time=1*)

**Parameters** **time** (*float*) – Wait specified interval in seconds before calling sub-handlers

**__call__**(*obj=None*, *parent=None*)
Callback function, see `Handler.__call__()`. Does nothing unless specified amount of time is passed. After the time all sub-handlers registered with `Handler.register()` are called.

**class** solver.**Preconditioned**(*L*, *P=<function <lambda> at 0x42f6b90>*)
Iterative solver of linear system L X = R, descendant of `solver.Iteration`. The operator L and pre-conditions are passed to the constructor.

**class** solver.**Reporter**
Callback object, descendant of `Handler`. Print number of iterations and residue to stdout. Can be linked with descendants of `Interation`.

**__init__**()
Call parent constructor and register `hook()` as callback function.

**hook**(*obj*, *parent=None*)
Print solution statistics.

**class** solver.**Solver**
Abstract solver class. Provides method `Solver.solve()`, which solve a linear system with operator previously passed to the constructor. Objects of the class are callable, the calls are redirected to `solve()` method. Callback objects monitoring solution progress can be registered by `register()`.

**__init__**()
Construct the solver.

---

**Note:** All children must call the constructor upon construction.

**__call__**(*R*)
Redirect call to `Solver.solve()`.

**register**(*fun*)
Register callback object to call after every iteration.

**Parameters** **fun** (`solver.Handler`) – Callback object

**report**()
Call registered callback objects.

**solve**(*R*)
Solve linear system A X = R with given right hand side R and the operator A passed to the constructor for X.

**Parameters** **R** (*an object of class implementing linear algebra*) – Right hand side

**Returns** Solution of the system

**Return type** type of `R`

### 5.1.9 toper

Module `toper` contains definitions of operators on tree structures `tree.T` and in particular on tensor field (trees of scalar fields `field.F`). `foper.TO` is an abstract class, parent of operators on trees. The class is callable, and `TO.__call__()` evaluates operator on the argument. To obtain grids for image of the operator use `TO.image()`. The matrix of the operator are returned by `TO.matrix()`. If you interesting only in diagonal elements use `TO.diag()` or `TO.arrowDiag()`. Operators algebra is implemented for `foper.TO`. Operators can be added, subtracted and multiplied by scalar or scalar fields, see `TO.__add__()`, `TO.__sub__()`, `TO.__mul__()`, `TO.__neg__()`. Multiplication of two operators are defined as composition, see `TO.__mul__()`.

**Descendants of `foper.TO` are**

- Identity operator `foper.TId`.
- Sum of operators `foper.TOSum`.
- Difference of operators `foper.TOSub`.
- Multiplication by scalar `foper.TOMulS`.
- Multiplication by scalar field `foper.TOMulF`.
- Composition of operators `foper.TOComp`.
- Element-wise application of an operator `foper.TF`.
- Operator with one column matrix `foper.TOVert`.
- Operator with one row matrix `foper.TOHor`.
- Operator defined by its matrix `foper.TOMat`.

- Tensor transpose `foper.TOT`.
- Forward and backward element-wise finite differences `foper.TD`, `foper.TB`.
- Extension operator `foper.TExt`.

Arbitrary callable object can be transformed to `foper.FO` with limited functionality using wrapper `TCallable`.

**Auxiliary function can be used to**

- Create free-slip boundary conditions `foper.freeSlip()`.
- Create forward and backward gradient `foper.gradF()`, `foper.gradB()`.
- Create forward and backward divergence `foper.divF()`, `foper.divB()`.
- Create operator calculating strain `foper.strainB()`.
- Create Stokes operator `foper.stokes()`.
- Create operators evaluating different addenda in the Stokes operator `foper.mulDivFa()`, `foper.divMulFa()`, `foper.mulDivF()`, `foper.divMulF()`, `foper.stokes1()`, `foper.stokes2()`.
- Calculate range of Stokes operator `foper.stokes2mask()`.
- Create direct sum of Stokes and divergence operators and its parts `stokesCont()`, `stokesCont1()`, `stokesCont2()`.
- Convert various types to operators `foper.conv()`.

**class** `toper.`**`TB`**(*a*)

Backward finite difference for every node. Descendant of `toper.TO`.

**`__init__`**(*a*)

**Parameters** **a** (*int*) – Axis to apply finite difference

**`__call__`**(*x*)

See `TO.__call__()`.

**`image`**(*tree*)

See `TO.image()`.

**`matrix`**(*t*)

See `TO.matrix()`.

**class** `toper.`**`TCallable`**(*fun*)

Wrapper for callable objects. Descendant of `toper.TO`. .. note

```
:meth:`FO.matrix`, :meth:`FO.diag`, :meth:`FO.arrowDiag`
are not implemented.
```

**`__init__`**(*fun*)

**Parameters** **fun** (*callable*) – A function evaluation the operator

**`__call__`**(*X*)

Redirect call to the function `fun` passed to the constructor.

**class** toper.**TD**(*a*)

Forward finite difference for every node. Descendant of toper.TO.

```
>>> A = TD(0)
>>> G = fi.Grid((3,3))
>>> X = fi.fromFunction(lambda x,y:x+y, G)
>>> print A(tr.fromList([X]))
{[[ 1.  1.  1.]
 [ 1.  1.  1.]]}
>>> Y = tr.fromList([X, X])
>>> print A(Y)
{[[ 1.  1.  1.]
 [ 1.  1.  1.]], [[ 1.  1.  1.]
 [ 1.  1.  1.]]}
>>> T = tr.fromList([G, G])
>>> A.image(T)
{<Grid: Size 2x3 Origin (0.0, 0.0) Spacing (1.0, 1.0)>, <Grid: Size 2x3 Orig
>>> A.matrix(T)
{{(0,): <FD0>}, {(1,): <FD0>}}
```

**__init__**(*a*)

**Parameters** **a** (*int*) – Axis to apply finite difference

**__call__**(*x*)

See TO.__call__().

**image**(*tree*)

See TO.image().

**matrix**(*tree*)

See TO.matrix().

**class** toper.**TExt**(*bc*)

Extension operator. Descendant of toper.TO. Apply operator foper.FExt to every node.

```
>>> G = fi.Grid((3,3))
>>> T = tr.fromList([[G], [G]])
>>> A = TExt(fi.nbc)
>>> A.image(T)
{{<Grid: Size 5x5 Origin (-1.0, -1.0) Spacing (1.0, 1.0)>}, {<Grid: Size 5x5
>>> A.matrix(T)
{{{(0, 0): <FExt<Neumann>>}}, {{(1, 0): <FExt<Neumann>>}}}
>>> A = TExt(tr.fromList([[fi.nbc],[fi.ubc]]))
>>> A.image(T)
{{<Grid: Size 5x5 Origin (-1.0, -1.0) Spacing (1.0, 1.0)>}, {<Grid: Size 3x3
>>> A.matrix(T)
{{{(0, 0): <FExt<Neumann>>}}, {{(1, 0): <FExt<BC>>}}}
```

**__init__**(*bc*)

**Parameters** **bc** (tree.T of arguments acceptable by field.parseBC) – Boundary conditions

**__call__**(*x*)
See TO.__call__().

**image**(*tree*)
See TO.image().

**matrix**(*tree*)
See TO.matrix().

**class** toper.**TF**(*A*)
Wrapper for operators foper.FO on scalar fields. Descendant of toper.TO.

Descendant of toper.TO (except of toper.TF) should not take foper.FO as argument. Instead class foper.FO should be wrapped to toper.TF. The class toper.TF applies given operator foper.FO to every node of passed tree.

```
>>> B = TF(fo.FId())
>>> G = fi.Grid((3,3))
>>> B(tr.fromList([fi.ones(G)]))
{<F: Size 3x3 Origin (0.0, 0.0) Spacing (1.0, 1.0)>}
>>> T = tr.fromList([G])
>>> B.matrix(T)
{{(0,): <FId>}}
>>> B.diag(T)
{1.0}
```

**__init__**(*A*)

**Parameters** **A** (foper.FO) – Operator on a scalar field

**__call__**(*tree*)
See FO.__class__().

**image**(*tree*)
See FO.image().

**matrix**(*tree*)
See FO.matrix().

**class** toper.**TId**
Identity operator. Descendant of toper.TO.

```
>>> T = tr.fromList([[1,1],1])
>>> A = TId()
>>> A.image(T)
{{1, 1}, 1}
>>> A(T)==T
True
>>> A.matrix(T)
{{{(0, 0): 1}, {(0, 1): 1}}, {(1,): 1}}
>>> TO.__call__(A, T)==A(T)
True
```

**__init__**()
Requires no arguments.

**__call__**(*X*)
See `TO.__call__()`. Does not change argument.

**image**(*tree*)
See `TO.image()`. Grids for image coincide with ones for preimage.

**matrix**(*tree*)
The matrix has only diagonal element equals to one.

**class** `toper.`**TO**
Linear operator acting on `tree.T`.

**arrowDiag**(*t*)

**Parameters** **t** – Tree of grids for the preimage

**Returns** Diagonal elements of the operator with ones substituted for zeros

**Return type** `tree.T` of `field.F` or scalars

**diag**(*t*)

**Parameters** **t** – Tree of grids for the preimage

**Returns** Diagonal elements of the operator

**Return type** `tree.T` of `field.F` or scalars

**image**(*tree*)

**Parameters** **tree** (`tree.T` of `field.Grid`) – Tree of grids for the preimage

**Returns** Tree of grids for the image

**Return type** `tree.T` of `field.Grid`

**matrix**(*tree*)

**Parameters** **tree** (`tree.T` of `field.Grid`) – Tree of grids for the preimage

**Returns** Matrix elements of the operator

**Return type** `tree.T` of `dict`

Matrix elements are stored in a tree `tree.T`. Every node of the tree is a rule to compute corresponding note of the image of the operator. Every rule is a dictionary, such that key of an element is indexes of node in source tree, and value of the element is the coefficient to multiply the node of source tree. The coefficient can be a scalar or a scalar field `field.F`.

**class** `toper.`**TOComp**(*A*, *B*)
Composition of operators. Descendant of `toper.TO`.

```
>>> A = TId()
>>> X = tr.fromList([[1,1],1])
>>> B = A*A
>>> B(X)==A(X)
```

```
True
>>> TO.__call__(B, X)==B(X)
True
>>> T = tr.fromList([1])
>>> B.image(T)
{1}
>>> B.matrix(T)
{{(0,): 1}}
>>> G = fi.Grid((3,3))
>>> T = tr.fromList([G])
>>> I = fo.FD(0)
>>> A = TF(I)
>>> B = TId()*A
>>> print B.matrix(T)
{{(0,): <FD0>}}
```

**__init__**(*A*, *B*)

**Parameters**

- **A** (toper.TO) – Operator to apply second
- **B** (toper.TO) – Operator to apply first

**__call__**(*X*)

See TO.__call__(). Evaluate composition

(AB)(x)=A(B(x)).

**image**(*tree*)

See FO.image().

**matrix**(*tree*)

See FO.matrix().

**class** toper.**TOHor**(*mat*)

Operator of the form

(x_1,... x_n) -> V_1 x_1+...+V_n x_n.

Descendant of toper.TO.

```
>>> A = TId()
>>> B = TOHor([-1, A])
>>> X = tr.fromList([[1], [3]])
>>> B(X)
{2}
>>> T = tr.fromList([[1],[1]])
>>> B.image(T)
{1}
>>> B.matrix(T)
{{(1, 0): 1, (0, 0): -1}}
>>> TO.__call__(B, X)==B(X)
True
>>> C = TOHor([fo.FId(), -fo.FId()])
```

```
>>> print C.matrix(tr.fromList([[fi.Grid((3,3))], [fi.Grid((3,3))]]))
{{(1, 0): ((-1*)*<FId>), (0, 0): <FId>}}
```

**__init__**(*mat*)

**Parameters** **mat** (list of `toper.TO`) – list of operators V_k

**__call__**(*X*)
See `TO.__call__()`.

**image**(*tree*)
See `TO.image()`.

**matrix**(*tree*)
See `TO.matrix()`.

**class** `toper.`**TOMat**(*mat*)
Operator defined by its matrix. Descendant of `toper.TO`.

```
>>> A = TOMat([[0,1],[1,0],[-1,1]])
>>> X = tr.fromList([2,3])
>>> A.image(X)
{6, 6, 6}
>>> A.matrix(X)
{{0: 0, 1: 1}, {0: 1, 1: 0}, {0: -1, 1: 1}}
>>> A(X)
{3, 2, 1}
>>> TO.__call__(A, X)==A(X)
True
>>> B = TOMat([[1,1,0],[0,1,1]])
>>> (B*A)(X)
{5, 3}
>>> (B*A).matrix(X)
{{0: 1, 1: 1}, {0: 0, 1: 1}}
```

**__init__**(*mat*)

**Parameters** **mat** (*list of lists of scalars*) – Matrix of coefficients stored as nested lists in row-major order

**image**(*tree*)
See `TO.image()`.

**matrix**(*tree*)
See `TO.matrix()`.

**class** `toper.`**TOMulF**(*C*)
Multiplication by a scalar field. Descendant of `toper.TO`.

```
>>> G = fi.Grid((3,3))
>>> X = 2*fi.ones(G)
>>> A = TOMulF(X)
>>> print A(X)
[[ 4.  4.  4.]
```

```
 [ 4.  4.  4.]
 [ 4.  4.  4.]]
```

**__init__**(*C*)

**Parameters** **C** (field.F) – Multiplier

**__call__**(*X*)

See FO.__class__().

**image**(*tree*)

See FO.image().

**matrix**(*tree*)

See FO.matrix().

**class** toper.**TOMulS**(*C*)

Operator of multiplication by scalar. Descendant of toper.TO.

```
>>> T = tr.fromList([2,1])
>>> A = TOMat([[1,2],[3,4]])
>>> B = 2*A
>>> B.matrix(T)
{{0: 2, 1: 4}, {0: 6, 1: 8}}
>>> TO.__call__(B, T)==B(T)
True
```

**__init__**(*C*)

**Parameters** **C** (*scalar*) – Multiplier

**__call__**(*X*)

See TO.__class__.

**image**(*tree*)

See TO.image.

**matrix**(*tree*)

See TO.matrix.

**class** toper.**TOSub**(*A*, *B*)

Difference of operators. Descendant of toper.TO.

```
>>> T = tr.fromList([[1,1],1])
>>> A = TId()
>>> B = A-A
>>> B.matrix(T)
{{(0, 0): 0}, {(0, 1): 0}}, {(1,): 0}}
>>> TO.__call__(B, T)==B(T)
True
```

**__init__**(*A*, *B*)

**Parameters**

- **A** (foper.FO) – Minuend

- **B** (foper.FO) – Subtrahend

**__call__**(*X*)

**Parameters** **X** (tree.T of field.F) – Argument

**Returns** Difference of values on X of arguments A and B of the constructor

**Return type** tree.T of field.F

**image**(*tree*)

**Parameters** **tree** (tree.T of field.Grid) – Grids for preimage

**Return type** tree.T of field.Grid

**Returns** Intersection of grids for addenda

**matrix**(*tree*)

See FO.matrix().

**class** toper.**TOSum**(*A*, *B*)

Sum of operators. Descendant of toper.TO.

```
>>> T = tr.fromList([[1,1],1])
>>> A = TId()
>>> B = A+A
>>> B(T)==A(T)+A(T)
True
>>> TO.__call__(B, T)==B(T)
True
>>> G = fi.Grid((2,2))
>>> T = tr.fromList([G, G])
>>> B.image(T)
{<Grid: Size 2x2 Origin (0.0, 0.0) Spacing (1.0, 1.0)>, <Grid: Size 2x2 Orig
```

**__init__**(*A*, *B*)

**Parameters** **B** (*A*,) – Addenda

**__call__**(*X*)

**Parameters** **X** (tree.T of field.F) – Argument

**Returns** Sum of values of addenda on X

**Return type** tree.T of field.F

**image**(*tree*)

**Parameters** **tree** (tree.T of field.Grid) – Grids for preimage

**Return type** tree.T of field.Grid

**Returns** Intersection of grids for addenda

**matrix**(*tree*)

See FO.matrix().

**class** toper.**TOT**

Tensor transpose operator. Descendant of toper.TO.

```
>>> X = tr.fromList([[1, 2], [3, 4]])
>>> A = TOT()
>>> A(X)
{{1, 3}, {2, 4}}
>>> TO.__call__(A, X)==A(X)
True
>>> T = tr.fromList([[1,1],[1,1],[1,1]])
>>> A.matrix(T)
{{{(0, 0): 1}, {(1, 0): 1}, {(2, 0): 1}}, {{(0, 1): 1}, {(1, 1): 1}, {(2, 1)
>>> A.image(T)
{{1, 1, 1}, {1, 1, 1}}
>>> (A*A).image(T)
{{1, 1}, {1, 1}, {1, 1}}
>>> (A*A).matrix(T)
{{{(0, 0): 1}, {(0, 1): 1}}, {{(1, 0): 1}, {(1, 1): 1}}, {{(2, 0): 1}, {(2,
```

**__call__**(*X*)

See TO.__call__().

**image**(*tree*)

See TO.image().

**matrix**(*tree*)

See TO.matrix().

**class** toper.**TOVert**(*mat*)

Operator with the matrix

x->(V_1 x,... V_n x).

Descendant of toper.TO.

```
>>> A = TId()
>>> B = TOVert([-1, A])
>>> X = tr.fromList([1,2])
>>> B(X)
{{-1, -2}, {1, 2}}
>>> T = tr.fromList([1,1])
>>> B.image(T)
{{1, 1}, {1, 1}}
>>> TO.__call__(B, X)==B(X)
True
>>> C = TOVert([fo.FId(), -fo.FId()])
>>> G = fi.Grid((3,3))
>>> T = tr.fromList([G])
>>> Y = fi.ones(G)
>>> C(Y)
{<F: Size 3x3 Origin (0.0, 0.0) Spacing (1.0, 1.0)>, <F: Size 3x3 Origin (0.
>>> C.image(T)
{{<Grid: Size 3x3 Origin (0.0, 0.0) Spacing (1.0, 1.0)>}, {<Grid: Size 3x3 O
```

```
>>> C.matrix(T)
{{{(0,): <FId>}}, {{(0,): ((-1*)*<FId>)}}}
```

**__init__**(*mat*)

**Parameters** **mat** (list of `toper`) – List of operators

**__call__**(*X*)
See `TO.__call__()`.

**image**(*tree*)
See `TO.image()`.

**matrix**(*tree*)
See `TO.matrix()`.

`toper.`**conv**(*y*)
Convert argument to operator.

**Parameters** **y** (`TO` or `foper.FO` or `field.F` or scalar) – Argument

**Return type** `TO`

**Returns** Depending on type of the argument returns: * copy of the argument, for type `FO` * `TF`, for type `foper.FO` * `TOMulS`, for scalar argument * `TOMulF`, for type `field.F`

`toper.`**divB**(*dim*)

**Parameters** **dim** (*integer*) – Dimension of the space

**Returns** Divergence of a tensor field, where derivative is approximated by backward difference

**Return type** `toper.TO`

`toper.`**divF**(*dim*)

**Parameters** **dim** (*integer*) – Dimension of the space

**Returns** Divergence of a tensor field, where derivative is approximated by forward difference

**Return type** `toper.TO`

```
>>> G = fi.Grid((3,3))
>>> X = fi.fromFunction(lambda x,y: x+2*y, G)
>>> Y = fi.fromFunction(lambda x,y: 2*x+y, G)
>>> Z = tr.fromList([[X], [Y]])
>>> A = divF(2)
>>> print A(Z)
{[[ 2.  2.]
 [ 2.  2.]]}
```

`toper.`**divMulF**(*dim*, *Nu*)
Create operator evaluating

divMulF(V)_j(x) = sum_i (DF_i Nu)(x) V_{ij}(x+h_i).

**Parameters**

- **dim** (*integer*) – Dimension of the space
- **Nu** (`field.F`) – Scalar field

**Returns** Operator defined by the formula above

**Return type** `toper.TO`

`toper.`**`divMulFa`**(*dim*, *Nu*)

Create operator evaluating

divMulF(V)_i(x) = sum_i (DF_i Nu)(x) V_i(x).

**Parameters**

- **dim** (*integer*) – Dimension of the space
- **Nu** (`field.F`) – Scalar field

**Returns** Operator defined by the formula above

**Return type** `toper.TO`

`toper.`**`freeSlip`**(*dim*)

**Parameters** **dim** (*integer*) – Dimension of the space

**Returns** Free-slip boundary conditions.

**Return type** `tree.T` of list of tuple of `field.BC`

```
>>> freeSlip(3)
{{(<Dirichlet>, <Neumann>, <Neumann>)}, {(<Neumann>, <Dirichlet>, <Neumann>)
```

`toper.`**`gradB`**(*dim*)

**Parameters** **dim** (*integer*) – Dimension of the space

**Returns** Gradient of a tensor field, where derivative is approximated by backward difference

**Return type** `toper.TO`

`toper.`**`gradF`**(*dim*)

**Parameters** **dim** (*integer*) – Dimension of the space

**Returns** Gradient of a tensor field, where derivative is approximated by forward difference

**Return type** `toper.TO`

```
>>> G = fi.Grid((2,2))
>>> X = fi.fromFunction(lambda x,y: x+2*y, G)
>>> A = gradF(G.dim)
>>> print A(tr.fromList([X]))
{{[[ 1.  1.]]}, {[[ 2.]
```

```
  [ 2.]]}}
>>> Y = tr.fromList([[X], [X]])
>>> print A(Y)
{{{[[ 1.  1.]]}, {[[ 1.  1.]]}}, {{[[ 2.]
  [ 2.]]}, {[[ 2.]
  [ 2.]]}}}
>>> T = tr.fromList([[G], [G]])
>>> A.image(T)
{{{<Grid: Size 1x2 Origin (0.0, 0.0) Spacing (1.0, 1.0)>}, {<Grid: Size 1x2
>>> A.matrix(T)
{{{{(0, 0): <FD0>}}, {{(1, 0): <FD0>}}}, {{{(0, 0): <FD1>}}, {{(1, 0): <FD1>
```

toper.**mulDivF**(*dim*, *Nu*)

Create operator evaluating

mulDivF(V)_j(x) = Nu(x) sum_i DF_i V_{ij}(x).

**Parameters**

- **dim** (*integer*) – Dimension of the space
- **Nu** (field.F) – Scalar field

**Returns** Operator defined by the formula above

**Return type** toper.TO

toper.**mulDivFa**(*dim*, *Nu*)

Create operator evaluating

mulDivF(V)_i(x) = sum_i Nu(x+h_i) DF_i V_i(x).

**Parameters**

- **dim** (*integer*) – Dimension of the space
- **Nu** (field.F) – Scalar field

**Returns** Operator defined by the formula above

**Return type** toper.TO

toper.**stokes**(*dim*, *Nu*)

Stokes operator

stokes(V) = -divF(Nu*strainB(V)).

**Parameters** **dim** (*integer*) – Dimension of the space

**Returns** Stokes operator

**Return type** toper.TO

```
>>> G = fi.Grid((4, 4))
>>> T = tr.fromList([[G], [G]])
>>> A = stokes(2, 1)
>>> A.image(T)
{{<Grid: Size 2x2 Origin (1.0, 1.0) Spacing (1.0, 1.0)>}, {<Grid: Size 2x2 O
>>> A.matrix(T)
{{{(1, 0): (<FD1>*<FB0>), (0, 0): ((<FD0>*(<FB0> + <FB0>)) + (<FD1>*<FB1>))}
>>> A.diag(T)
{{-6.0}, {-6.0}}
```

toper.**stokes1** (*dim*, *Nu*)

Create operator evaluating

stokes1(V)_j(x) = sum_i Nu(x+h_i) DF_i strainB(V)_{ij}(x).

**Parameters**

- **dim** (*integer*) – Dimension of the space
- **Nu** (`field.F`) – Scalar field

**Returns** Operator defined by the formula above

**Return type** `toper.TO`

toper.**stokes2** (*dim*, *Nu*)

Create operator evaluating

stokes2(V)_j = sum_i (DF_i Nu) strainB(V)_{ij}.

**Parameters**

- **dim** (*integer*) – Dimension of the space
- **Nu** (`field.F`) – Scalar field

**Returns** Operator defined by the formula above

**Return type** `toper.TO`

toper.**stokes2mask** (*dim*, *Nu*)

**Parameters**

- **dim** (*integer*) – Dimension of the space
- **Nu** (`field.F`) – Scalar field

**Returns** Return indexes of non-zero values of the image. Can be used by `selectNonzero()`.

**Return type** `toper.TO`

toper.**stokesCont** (*dim*, *Nu*, *bc=<Dirichlet>*)

Operator acting on trees

{velocity field, pressure field},

returning

{value of Stokes operator, divergene of velocity field}.

Solution of the equation

stokesCont(...)(tr.fromList([V, P])==tr.fromList([G,0])

is the solution of the Stokes equation with continuity equation for external force field `G`.

```
>>> N = 3
>>> GV = fi.Grid((N-1, N-1))
>>> GP = fi.Grid((N, N))
>>> VX = fi.fromFunction(lambda x,y: -y, GV)
>>> VY = fi.fromFunction(lambda x,y: x, GV)
>>> V = tr.fromList([[VX], [VY]])
>>> P = fi.fromFunction(lambda x,y: y, GP)
>>> X = tr.fromList([V, [P]])
>>> S = stokesCont(2, 1)
>>> S(X)
{{{<F: Size 2x2 Origin (0.0, 0.0) Spacing (1.0, 1.0)>}, {<F: Size 2x2 Origin
>>> T = tr.fromList([[[GV], [GV]], [GP]])
>>> S.image(T)
{{{<Grid: Size 2x2 Origin (0.0, 0.0) Spacing (1.0, 1.0)>}, {<Grid: Size 2x2

>>> S.diag(T)
{{{<F: Size 2x2 Origin (0.0, 0.0) Spacing (1.0, 1.0)>}, {<F: Size 2x2 Origin
```

toper.**strainB**(*dim*)

Evaluate strain field

strain(F) = (grad(F) + transpose(grad F))

**Parameters** **dim** (*integer*) – Dimension of the space

**Returns** Operator of strain calculation

**Return type** `toper.TO`

```
>>> G = fi.Grid((2, 2))
>>> T = tr.fromList([[G], [G]])
>>> X = fi.fromFunction(lambda x,y: 2*x+y, G)
>>> Y = fi.fromFunction(lambda x,y: x+2*y, G)
>>> Z = tr.fromList([[X], [Y]])
>>> A = strainB(2)
>>> print A(Z)
{{{[[ 4.  4.]]}, {[[ 2.]]}}, {{[[ 2.]]}, {[[ 4.]
  [ 4.]]}}}
>>> A(Z)
{{{<F: Size 1x2 Origin (1.0, 0.0) Spacing (1.0, 1.0)>}, {<F: Size 1x1 Origin
>>> A.image(Z.grid)==A(Z).grid
True
>>> print strainB(2).matrix(T)
{{{{(0, 0): (<FB0> + <FB0>)}}, {{(1, 0): <FB0>, (0, 0): <FB1>}}}, {{{(1, 0):
```

```
>>> TO.__call__(A, Z)==A(Z)
True
```

### 5.1.10 tree

The module containes definition of tree container `T` having special features to work with algebraic structures such as direct sums and tensors.

**class** `tree.T`(*data*)

Tree class. Internally trees are stored in nested lists

T = [T] | <arbitrary object>

`fromTree()` can be used for construction of `T` from nested lists. Strings representations returned by `T.__str__()` and `T.__repr__()` are the same and are coincide with representation of a list except of curly brackets.

Trees are equal, if and only if they have common shape (degree of all nodes), and nodes coincide, see `T.__eq__()`, `T.__neq__()`.

Elements of a tree are indexed by tuples (i1,.. in). where i1 is the index of child of parent node, i2 is the index of child of the child and so on. If an index is a slice or ellipsis, indexation operation returns a subtree, see details in `T.__getelem__()`.

Trees are immutable. There is no way to add an node to a tree after construction, neither to alter one.

Trees can be glued using `T.__add__()`.

`T.map()`, `T.mapIdx()`, `T.imap()` can be used to apply a function to all nodes of a tree or trees. `T.amap()` is an analog of reduce (fold) function.

If degree of all node of a level is the same, then the degree is called dimension of the level, which can be obtained using `T.shape()`. Indexes must satisfy 0<=ik<shape[k], if shape is defined.

To represent a direct sum of spaces, a tree of the form {a[1], a[2],.. } is recommended, where ak belongs to k-th addendum. For arbitrary tree arithmetic operators are defined element-wise, see `T.__add__()`, `T.__sub__()`, `T.__mul__()`, `T.__div__()`, which is correct for direct sums.

If all dimensions are defined, the tree is an another representation of a multidimensional array, which can store coefficients of a tensor. That is tensor T[i,j] in two dimensional space is stored in {{T[1,1], T[1,2]}, {T[2,1], T[2,2]}}. For this special case tensor dot operation is defined `T.dot()`.

```
>>> T = fromList([[1,2],3])
>>> T
{{1, 2}, 3}
>>> 2*T==T+T
True
>>> T.shape
(2, None)
```

```
>>> T[0,1]
2
>>> T[0,:]
{1, 2}
>>> T[1,...]
3
>>> T.map(type)
{{<type 'int'>, <type 'int'>}, <type 'int'>}
>>> T.mapIdx(lambda i,x:i)
{{(0, 0), (0, 1)}, (1,)}
>>> X = fromList([[1,2],[3,4]])
>>> X!=T
True
>>> X.shape
(2, 2)
>>> X.T
{{1, 3}, {2, 4}}
>>> X.dot(X.T)
{7, 22}
>>> X.map(lambda x: x%2).all()
False
>>> X.map(lambda x: x%2).any()
True
>>> X.mapReduce(lambda x:1, sum)
4
>>> X.grid
{{1, 1}, {1, 1}}
```

**__init__**(*data*)

Create a tree with given children of root note.

**Parameters** **data** (*list*) – List of children of root note. Given elements of type `T` are subtrees.

**__eq__**(*other*)

Trees are equals, if they have the same shape and their corresponding nodes are equal.

**Return** **bool** True, if `self` is equals to `other`

**__neq__**(*other*)

Negotiation of `T.__eq__()`.

**__getitem__**(*key*)

If given *key* is a tuple of integers, the node with given indexes is returned. If *key* contains slice or Ellipsis, the subtree, containing nodes satisfying given criteria, is returned. Ellipsis (...) does mean that complete tree must be returned, and indexes after Ellipses are ignored. Except of Ellipses, number of indexes must be equal to depth of desired element.

**Parameters** **key** (*tuple of int, slice or Ellipses*) – Index(es) of selected nodes

**Returns** Subtree or node with given index(es)

**Return type** `T` or an object

**__and__**(*other*)
Merge trees.

**Parameters** **other** (`tree.T`) – A tree or an object

**Returns** Tree such that a node is a child of the root, if the node is a child of root of `self` or of root of `other`

**Return type** `tree.T`

**__add__**(*other*)
Add elements of trees node-wise.

**__sub__**(*other*)
Subtract elements of `other` from elements of `self`.

**__mul__**(*other*)
If `other` is a tree, return node-wise multiplication of `self` and `other`. Otherwise return right multiplication of every node of the tree `self` by `other`.

**__div__**(*other*)
If `other` is a tree, return node-wise division of elements of `self` by elements of `other`. Otherwise return diviosion of every node of the tree `self` by `other`.

**T**
Permute first and second dimensions.

---

**Note:** First and second dimensions of the tree should not be `None`.

---

**Returns** Transposed tree

**Return type** `tree.T`

**accum**(*a*, *fun*)
Iterate all nodes of the tree (depth-first) and substitute `fun` (a_k, x_k)[1] for a node k having value x_k where a_1 = `a`, and a_k = `fun` (a_{k-1}, x_{k-1})[0]. An analog of `reduce()` for flattened tree, but preserve tree structure.

**Parameters**

- **a** (*object*) – Initial value of accumulator
- **fun** (*callable*) – Function of arguments (current value of accumulator, value of the node) returning the pair (new value of accumulator, new value of the node).

**Returns** Tree of the same shape with substituted values

**Return type** `T`

**all**()

**Returns** True, if all leafs are equal to True

**Return type** bool

**any**()

**Returns** True, if there is a leaf equals True

**Return type** bool

**check**(*other*)

**Parameters** **other** (T) – A tree to compare with

**Returns** True, iff shape of *other* matches shape of *self*, and x.check(y) is true for all elements *x* of *self* and *y* of *other*.

**Return type** bool

---

**Note:** Elements of the tree must implement method *check*. Most useful for trees of field.F.

---

**db**(*a*)

Call db() for every leaf node. Equivalent to:

```
tree.map(lambda x: x.db(a))
```

**df**(*a*)

Call df() for every leaf node. Equivalent to:

```
tree.map(lambda x: x.df(a))
```

**dot**(*other*)

Calculate tensor multiplication

(A*B)[i,j] = sum_k A[k,i]*B[k,j]

---

**Note:** First dimensions of multipliers must coincide.

---

**Parameters** **other** (tree.T) – Second multiplier

**grid**

**Returns** Maps every node of the tree to grid of the node, if node is field.F or to 1 otherwise.

**Return type** T of field.Grid or of *int*.

**imap**(*other*, *fun*, *throw=True*)

Given two trees of the same shape, apply function fun to pairs of nodes with equal indexes.

**Parameters**

- **other** (`tree.T`) – Second tree. `self` is the first tree. Shapes of `self` and `other` must coincide.
- **fun** (*function of two arguments*) – A mapping (node of first tree, node of second tree) -> new value of node
- **throw** (*bool*) – See `T.map()`

**Returns** Tree of values returned by `fun`

**Return type** `tree.T`

**map** (*fun*, *throw=True*)
Apply given function to all nodes of the tree.

**Parameters**

- **fun** (*callable*) – Function of one argument to apply
- **throw** (*bool*) – If the flag is False, return NotImplemented when `fun` raises exception `TypeError`. Otherwise reraise the exception.

**Returns** Tree of values of `fun`

**Return type** `tree.T`

**mapIdx** (*fun*, *idx=()*)
Do the same as `T.map()` but the function `fun` has two arguments (index of node, value of node).

**mapReduce** (*leaf*, *branch*)
Apply the function `leaf` to all leaf nodes, and then the function `branch` to all branch nodes. For example, to calculate depth of every node one can call:

```
tree.mapReduce(lambda x: 1, lambda x: max(x))
```

**Parameters**

- **leaf** (*function of one argument*) – Function to apply to leafs
- **branch** (*function taking a list as only argument*) – Function to apply to branches

**Returns** Tree of calculated values

**Return type** `tree.T`.

**shape**
Calculate dimensions of the tree. k-th dimension is the degree (number of children) of the children having the depth k. If some of the children have distinct degrees, the corresponding dimension is None. If all dimensions are not None, the stored structure is an multidimensional array, and `shape()` returns its dimensions (tensor dimensions).

**Returns** Dimensions of the tree

**Return type** tuple of int or None.

**sqnorm**()

**Return float** square of l-2 norms of stored elements

---

**Note:** Stored elements must implement `field.F.sqnorm()`

---

tree.**fromFunction**(*shape*, *fun*, *idx=()*)

Construct a tree of given shape by executing a function over each coordinate.

**Parameters**

- **shape** (*tuple of integers*) – Dimensions of constructed tree
- **fun** (*function of one argument*) – Function mapping tuple of indexes to value of the node

**Returns** Tree of values returned by `fun`

**Return type** `tree.T`

tree.**fromList**(*data*)

**Parameters** **data** (*list*) – Nested list to convert to tree

**Returns** The tree representation of `data`

**Return type** `tree.T`

```
>>> fromList([[1, 2], 3])
{{1, 2}, 3}
```

CHAPTER 6

# Indices and tables

- *genindex*
- *modindex*
- *search*

# Python Module Index


f

field, 13
foper, 21

g

graphics, 29

l

log, 30

m

main, 31

r

ravel, 34

s

scale, 37
solver, 41

t

toper, 46
tree, 61

# Index

## Symbols


__add__() (foper.FO method), 25
__add__() (tree.T method), 63
__and__() (tree.T method), 63
__call__() (foper.FMatrix method), 24
__call__() (foper.FO method), 25
__call__() (solver.Handler method), 42
__call__() (solver.Periodic method), 45
__call__() (solver.Solver method), 46
__call__() (toper.TB method), 47
__call__() (toper.TCallable method), 47
__call__() (toper.TD method), 48
__call__() (toper.TExt method), 48
__call__() (toper.TF method), 49
__call__() (toper.TId method), 49
__call__() (toper.TOComp method), 51
__call__() (toper.TOHor method), 52
__call__() (toper.TOMulF method), 53
__call__() (toper.TOMulS method), 53
__call__() (toper.TOSub method), 54
__call__() (toper.TOSum method), 54
__call__() (toper.TOT method), 55
__call__() (toper.TOVert method), 56
__del__() (solver.Graph method), 41
__div__() (tree.T method), 63
__eq__() (field.F method), 16
__eq__() (tree.T method), 62
__getitem__() (field.F method), 16
__getitem__() (tree.T method), 62
__init__() (field.F method), 15
__init__() (field.Grid method), 18
__init__() (foper.FB method), 22
__init__() (foper.FD method), 22
__init__() (foper.FExt method), 23
__init__() (foper.FId method), 24
__init__() (foper.FMatrix method), 24
__init__() (foper.FOComp method), 26
__init__() (foper.FOMulS method), 27
__init__() (foper.FOSub method), 27
__init__() (foper.FOSum method), 28
__init__() (foper.FShift method), 29
__init__() (ravel.FFlatten method), 34
__init__() (ravel.FSelect method), 36
__init__() (ravel.TRavel method), 36
__init__() (scale.TScale method), 40
__init__() (solver.Graph method), 41
__init__() (solver.GraphLog method), 41
__init__() (solver.Handler method), 42
__init__() (solver.Iteration method), 43
__init__() (solver.Krein method), 44
__init__() (solver.Periodic method), 45
__init__() (solver.Reporter method), 45
__init__() (solver.Solver method), 45
__init__() (toper.TB method), 47
__init__() (toper.TCallable method), 47
__init__() (toper.TD method), 48
__init__() (toper.TExt method), 48
__init__() (toper.TF method), 49
__init__() (toper.TId method), 49
__init__() (toper.TOComp method), 51
__init__() (toper.TOHor method), 52
__init__() (toper.TOMat method), 52
__init__() (toper.TOMulF method), 53
__init__() (toper.TOMulS method), 53
__init__() (toper.TOSub method), 53
__init__() (toper.TOSum method), 54
__init__() (toper.TOVert method), 56
__init__() (tree.T method), 62
__mul__() (field.Grid method), 18

__mul__() (tree.T method), 63
__neg__() (foper.FO method), 25
__neq__() (tree.T method), 62
__setitem__() (field.F method), 16
__sub__() (foper.FO method), 25
__sub__() (tree.T method), 63

## A

accum() (tree.T method), 63
add() (in module log), 30
addDict() (in module foper), 29
all() (tree.T method), 63
almost_eq() (field.F method), 16
any() (tree.T method), 64
arrowDiag() (foper.FO method), 25
arrowDiag() (toper.TO method), 50
ArrowHurwitz (class in solver), 41

## B

BC (class in field), 13
box() (field.Grid method), 19

## C

caseAnalytic() (in module main), 32
casePixel() (in module main), 32
caseVortex() (in module main), 32
caseVortexFS() (in module main), 32
caseVortexPie() (in module main), 32
check() (tree.T method), 64
close() (in module graphics), 29
coarseGrid (scale.FBilinear attribute), 38
coarseGrid (scale.FDouble attribute), 39
coarseGrid (scale.Scale attribute), 39
coarseGrid (scale.TScale attribute), 40
common() (field.Grid method), 19
compare() (in module main), 32
conv() (in module toper), 56
copy() (field.Grid method), 19

## D

data (field.F attribute), 16
db() (field.F method), 16
db() (tree.T method), 64
df() (field.F method), 16
df() (tree.T method), 64
diag() (foper.FO method), 25
diag() (toper.TO method), 50
dim (field.Grid attribute), 19
Dirichlet (class in field), 14
divB() (in module toper), 56
divF() (in module toper), 56
divMulF() (in module toper), 56
divMulFa() (in module toper), 57
dot() (tree.T method), 64
drawSV() (in module graphics), 30

## E

embed() (ravel.FFlatten method), 34
embed() (ravel.FRavel method), 34
embed() (ravel.TRavel method), 36
extend() (field.F method), 16

## F

F (class in field), 14
FB (class in foper), 22
FBilinear (class in scale), 38
FD (class in foper), 22
FDouble (class in scale), 38
FExt (class in foper), 23
FFlatten (class in ravel), 34
FId (class in foper), 23
field (module), 13
figure() (in module graphics), 30
finnerGrid (scale.FBilinear attribute), 38
finnerGrid (scale.FDouble attribute), 39
finnerGrid (scale.Scale attribute), 39
finnerGrid (scale.TScale attribute), 40
FMatrix (class in foper), 24
FO (class in foper), 24
FOComp (class in foper), 26
FOMulS (class in foper), 26
foper (module), 21
FOSub (class in foper), 27
FOSum (class in foper), 28
FRavel (class in ravel), 34
freeSlip() (in module toper), 57
fromArray() (in module field), 20
fromFunction() (in module field), 20
fromFunction() (in module tree), 66
fromList() (in module tree), 66
FSelect (class in ravel), 35
FShift (class in foper), 28

## G

get() (field.BC method), 13
gradB() (in module toper), 57
gradF() (in module toper), 57

Graph (class in solver), 41
graphics (module), 29
GraphLog (class in solver), 41
Grid (class in field), 18
grid (field.F attribute), 17
grid (ravel.FFlatten attribute), 34
grid (ravel.FRavel attribute), 34
grid (tree.T attribute), 64
grids (ravel.TRavel attribute), 36

## H

Handler (class in solver), 41
high (field.Grid attribute), 19
hook() (solver.Reporter method), 45

## I

image() (foper.FB method), 22
image() (foper.FD method), 23
image() (foper.FMatrix method), 24
image() (foper.FO method), 25
image() (foper.FOMulS method), 27
image() (foper.FOSub method), 27
image() (foper.FOSum method), 28
image() (toper.TB method), 47
image() (toper.TD method), 48
image() (toper.TExt method), 49
image() (toper.TF method), 49
image() (toper.TId method), 50
image() (toper.TO method), 50
image() (toper.TOComp method), 51
image() (toper.TOHor method), 52
image() (toper.TOMat method), 52
image() (toper.TOMulF method), 53
image() (toper.TOMulS method), 53
image() (toper.TOSub method), 54
image() (toper.TOSum method), 54
image() (toper.TOT method), 55
image() (toper.TOVert method), 56
imap() (field.F method), 17
imap() (tree.T method), 64
indices() (scale.FBilinear method), 38
indices() (scale.FDouble method), 39
init() (solver.Iteration method), 43
init() (solver.Multi method), 45
iterate() (solver.Iteration method), 43
iterate() (solver.Multi method), 45
Iteration (class in solver), 42

## J

Jacobi (class in solver), 43

## K

Krein (class in solver), 44

## L

laplace() (in module foper), 29
laplaceMultiSolver() (in module main), 32
let() (in module log), 31
log (module), 30
low (field.Grid attribute), 19

## M

main (module), 31
map() (field.F method), 17
map() (tree.T method), 65
mapIdx() (tree.T method), 65
mapReduce() (tree.T method), 65
matrix() (field.BC method), 14
matrix() (foper.FO method), 26
matrix() (toper.TB method), 47
matrix() (toper.TD method), 48
matrix() (toper.TExt method), 49
matrix() (toper.TF method), 49
matrix() (toper.TId method), 50
matrix() (toper.TO method), 50
matrix() (toper.TOComp method), 51
matrix() (toper.TOHor method), 52
matrix() (toper.TOMat method), 52
matrix() (toper.TOMulF method), 53
matrix() (toper.TOMulS method), 53
matrix() (toper.TOSub method), 54
matrix() (toper.TOSum method), 54
matrix() (toper.TOT method), 55
matrix() (toper.TOVert method), 56
mesh() (field.Grid method), 19
mulDivF() (in module toper), 58
mulDivFa() (in module toper), 58
Multi (class in solver), 44

## N

Neumann (class in field), 20

## O

offsets (ravel.TRavel attribute), 37
ones() (in module field), 21
oper() (solver.Iteration method), 43
oper() (solver.Multi method), 45

order (field.BC attribute), 14

## P

parseBC() (in module field), 21
Periodic (class in solver), 45
pie() (in module main), 32
plot() (in module graphics), 30
pre() (solver.ArrowHurwitz method), 41
pre() (solver.Iteration method), 43
pre() (solver.Jacobi method), 44
pre() (solver.Multi method), 45
Preconditioned (class in solver), 45

## R

random() (in module field), 21
rank (ravel.FFlatten attribute), 34
rank (ravel.FRavel attribute), 34
rank (ravel.TRavel attribute), 37
ravel (module), 34
readVTK() (in module main), 32
register() (solver.Handler method), 42
register() (solver.Solver method), 46
report() (solver.Solver method), 46
Reporter (class in solver), 45
reset() (in module log), 31
reset() (in module main), 32
restrict() (field.F method), 17
restrict() (ravel.FFlatten method), 34
restrict() (ravel.FRavel method), 35
restrict() (ravel.TRavel method), 37
restrict() (scale.FBilinear method), 38
restrict() (scale.FDouble method), 39
restrict() (scale.Scale method), 39
restrict() (scale.TScale method), 40
runCase() (in module main), 32
runKrein() (in module main), 32
runKreinMulti() (in module main), 33
runMulti() (in module main), 33

## S

Scale (class in scale), 39
scale (module), 37
selectNonzero() (in module ravel), 37
shape (tree.T attribute), 65
shift() (field.F method), 17
shift() (field.Grid method), 20
show() (in module graphics), 30
show() (in module log), 31
show() (in module main), 33
smooth() (scale.FBilinear method), 38
smooth() (scale.FDouble method), 39
smooth() (scale.Scale method), 40
smooth() (scale.TScale method), 40
solve() (solver.Iteration method), 43
solve() (solver.Krein method), 44
solve() (solver.Solver method), 46
Solver (class in solver), 45
solver (module), 41
sqnorm() (field.F method), 18
sqnorm() (tree.T method), 66
stokes() (in module toper), 58
stokes1() (in module toper), 59
stokes2() (in module toper), 59
stokes2mask() (in module toper), 59
stokesCont() (in module toper), 59
stokesCont1MultiSolver() (in module main), 33
stokesContMultiSolver() (in module main), 33
strainB() (in module toper), 60
submatrix() (ravel.FRavel method), 35

## T

T (class in tree), 61
T (tree.T attribute), 63
TB (class in toper), 47
TCallable (class in toper), 47
TD (class in toper), 47
testLaplace() (in module main), 33
TExt (class in toper), 48
TF (class in toper), 49
TId (class in toper), 49
TO (class in toper), 50
TOComp (class in toper), 50
TOHor (class in toper), 51
TOMat (class in toper), 52
TOMulF (class in toper), 52
TOMulS (class in toper), 53
toper (module), 46
TOSub (class in toper), 53
TOSum (class in toper), 54
TOT (class in toper), 54
TOVert (class in toper), 55
translate() (field.F method), 18
translate() (field.Grid method), 20
TRavel (class in ravel), 36
tree (module), 61

TScale (class in scale), 40

## V

view() (field.Grid method), 20
Viz (class in main), 31
VizS (class in main), 31

## W

writeVTK() (in module main), 33

## Z

zeros() (in module field), 21