FEMpy Documentation

Release 1.0

Benjamin Floyd

Jul 25, 2019

User Guide

FEMpy is a pure-Python finite element method differential equation solver.

CHAPTER 1

Basic Usage

To solve a Poisson equation on a 1D interval with Dirichlet boundary conditions:

```
import numpy as np
from FEMpy import Interval1D, IntervalBasis1D, BoundaryConditions, Poisson1D

def dirichlet_funct(x):
   if x == 0:
      return 1
   elif x == 1:
      return 2

coefficient_funct = lambda x: 1
source_funct = lambda x: 4*x

mesh = Interval1D(left=0, right=1, h=1/4, basis_type='linear')
basis = IntervalBasis1D('linear')

bcs = BoundaryConditions(mesh, boundary_types=('dirichlet', 'dirichlet'), dirichlet_
↪fun=dirichlet_funct)
poisson_eq = Poisson(mesh, test_basis=basis, trial_basis=basis, boundary_
↪conditions=bcs)
poisson_eq.solve(coeff_fun=coefficient_funct, source_fun=source_funct)
```

A more complete example is available in the *quickstart* tutorial.

1.1 Installation

At present, the only way to install FEMpy is via source. Installation with pip or conda will come soon.

1.1.1 From Source

FEMpy is developed on GitHub. To get the latest and cutting edge version, it is easy to clone the source repository and install from there.

```
git clone https://github.com/floydie7/FEMpy.git
cd FEMpy
python setup.py install
```

1.1.2 Test the Installation

Unit and integration tests are provided with this package. To run the tests you will need py.test. Simply run the following command from the terminal in the package directory.

```
pytest -v tests
```

1.2 Mesh Generation

1.2.1 1D interval domain meshes

class FEMpy.Mesh.**Interval1D**(*left*, *right*, *h*, *basis_type*)

Defines a one-dimensional mesh and associated information matrices.

Create a mesh object defined from *left* to *right* with step size *h*. This object provides the information matrices describing the mesh (*P* and *T*) as well as the information matrices describing the finite element of type *basis_type* (*Pb* and *Tb*).

Parameters

- **left** (*float*) – The left end point of the domain.
- **right** (*float*) – The right end point of the domain.
- **h** (*float*) – Mesh grid spacing.
- **basis_type** (*{101, 'linear', 102, 'quadratic'}*) – Finite element basis type. Can either be called as a integer code or a string identifier to indicate the type of basis function we are using.
  - 101, 'linear' : 1-dimensional, linear basis.
  - 102, 'quadratic' : 1-dimensional, quadratic basis.

P

Information matrix containing the coordinates of all mesh nodes.

Type ndarray

T

Information matrix containing the global node indices of the mesh nodes of all the mesh elements.

Type ndarray

Pb

Information matrix containing the coordinates of all finite element nodes.

Type ndarray

Tb
Information matrix containing the global node indices of the finite element nodes.

Type ndarray

Examples

Automatically generate the boundary nodes.

```
>>> from FEMpy import Interval1D
>>> mesh = Interval1D(0, 1, 1/2, 'linear')
>>> mesh.boundary_nodes
array([0., 1.])
```

Get the vertices of the nth element.

```
>>> from FEMpy import Interval1D
>>> mesh = Interval1D(0, 1, 1/4, 'quadratic')
>>> mesh.get_vertices(3)
array([0.75, 1.  ])
```

Show the number of elements in the interval.

```
>>> from FEMpy import Interval1D
>>> mesh = Interval1D(-1, 3, 1/8, 'quadratic')
>>> mesh.num_elements_x
32
```

boundary_nodes
Returns the boundary node coordinates generated from the mesh.

get_vertices(*n*)
Extracts the vertices of the mesh element *n*.

Parameters **n** (*int*) – Mesh element index.

Returns The vertices of the mesh element *En*.

Return type ndarray

num_elements_x
Returns the number of elements along the domain x-axis.

1.2.2 2D rectangular meshes with triangular elements

class FEMpy.Mesh.**TriangularMesh2D**(*left*, *right*, *bottom*, *top*, *h1*, *h2*, *basis_type*)
Defines a two-dimensional mesh with triangular elements and associated information matrices.

Create a mesh object defined on the domain [*left*, *right] x [‘bottom*, *top*] with step size *h1* in the x-direction and *h2* in the y-direction. This object provides the information matrices describing the mesh (*P* and *T*) as well as the information matrices describing the finite element of type *basis_type* (*Pb* and *Tb*).

Parameters

- **left** (*float*) – The left edge point of the domain.
- **right** (*float*) – The right edge point of the domain.
- **bottom** (*float*) – The bottom edge point of the domain.

- **top** (*float*) – The top edge point of the domain.
- **h1** (*float*) – Mesh grid spacing along x-direction.
- **h2** (*float*) – Mesh grid spacing along y-direction.
- **basis_type** (*{201, 'linear', 'linear2D_tri', 202, 'quadratic', 'quadratic2D_tri'}*) – Finite element basis type. Can either be called as a integer code or a string identifier to indicate the type of basis function we are using.
  - 201, 'linear', 'linear2D_tri : 1-dimensional, linear basis on triangular elements.
  - 202, 'quadratic', 'quadratic2D_tri : 1-dimensional, quadratic basis on triangular elements.

P

Information matrix containing the coordinates of all mesh nodes.

Type ndarray

T

Information matrix containing the global node indices of the mesh nodes of all the mesh elements.

Type ndarray

Pb

Information matrix containing the coordinates of all finite element nodes.

Type ndarray

Tb

Information matrix containing the global node indices of the finite element nodes.

Type ndarray

Examples

Automatically generate the boundary nodes and edges.

```
>>> from FEMpy import TriangularMesh2D
>>> mesh = TriangularMesh2D(0, 1, 0, 1, 1/2, 1/2, 'quadratic')
>>> mesh.boundary_nodes
array([[0.  , 0.25, 0.5 , 0.75, 1., 1.  , 1. , 1.  , 1., 0.75, 0.5, 0.25, 0.  , 0.
↪  , 0.  , 0.  ],
       [0.  , 0.  , 0.  , 0.  , 0., 0.25, 0.5, 0.75, 1., 1.  , 1. , 1.  , 1.  , 0.
↪75, 0.5 , 0.25]])
>>> mesh.boundary_edges
array([[0. , 0.5, 1. , 1. , 1. , 0.5, 0. , 0. ],
       [0. , 0. , 0. , 0.5, 1. , 1. , 1. , 0.5]])
```

Get the number of elements in the x-axis and y-axis.

```
>>> from FEMpy import TriangularMesh2D
>>> mesh = TriangularMesh2D(0, 2, 0, 1, 1/2, 1/2, 'linear')
>>> mesh.num_elements_x
4
>>> mesh.num_elements_y
2
```

boundary_edges

Returns the boundary edge coordinates generated from the mesh.

num_elements_y
Returns the number of elements along the domain y-axis.

1.3 Finite Element Bases

1.3.1 Interval Element Basis Functions

class FEMpy.FEBasis.**IntervalBasis1D**(*basis_type*)
Defines a finite element basis function set on a one-dimensional interval.

Creates a complete set of finite element basis functions for a one-dimensional domain of either linear or quadratic basis types.

Parameters **basis_type** (*{101, 'linear', 102, 'quadratic'}*) – Finite element basis type. Can either be called as a integer code or a string identifier to indicate the type of basis function we are using.

- 101, 'linear' : 1-dimensional, linear basis.
- 102, 'quadratic' : 1-dimensional, quadratic basis.

basis_type
Basis type standardized to an integer code.

Type int

Raises TypeError – If the basis type is not a string identifier or an integer code.

basis_type
Basis type standardized to an integer code.

fe_local_basis(*x*, *vertices*, *basis_idx*, *derivative_order*)
Defines the finite element local basis functions.

Parameters

- **x** (*float or array_like*) – A value or array of points to evaluate the function on.
- **vertices** (*array_like*) – Global node coordinates for the mesh element *En*.
- **basis_idx** (*int*) – Local basis index value.
- **derivative_order** (*int*) – The derivative order to take the basis function to.

Returns The local basis function (or derivative of funtion) evaluated at the points in *x*.

Return type float

1.3.2 2D Triangular Element Basis Functions

class FEMpy.FEBasis.**TriangularBasis2D**(*basis_type*)
Defines a finite element basis function set on a two-dimensional domian with triangular elements.

Creates a complete set of finite element basis functions for a two-dimensional domain with triangular elements of either linear or quadratic basis types.

Parameters

- **basis_type** (*{201, 'linear', 'linear2D_tri', 202, 'quadratic', 'quadratic2D_tri'}*)

- **Finite element basis type. Can either be called as a integer code or a string identifier to indicate the**
- **type of basis function we are using.**
- **- 201, 'linear', 'linear2D_tri** (*1-dimensional, linear basis on triangular elements.*)
- **- 202, 'quadratic', 'quadratic2D_tri** (*1-dimensional, quadratic basis on triangular elements.*)

basis_type
Basis type standardized to an integer code.

Type int

Raises TypeError – If the basis type is not a string identifier or an integer code.

fe_local_basis(*coords*, *vertices*, *basis_idx*, *derivative_order*)
Defines the finite element local basis functions.

Parameters

- **coords** (*array_like*) – The x and y coordinate values to evaluate the function on. e.g., (`x`, `y`).
- **vertices** (*array_like*) – Global node coordinates for all vertices of triangular mesh nodes on the mesh element *En*.
- **basis_idx** (*int*) – Local basis index value.
- **derivative_order** (*tuple of int*) – The derivative orders to take the basis function to. Should be specified as a tuple with the orders corresponding to the coordinate axes in the basis functions. e.g., (`x_order`, `y_order`).

Returns The local basis function (or derivative of function) evaluated at the points in *coords*.

Return type float

1.4 Boundary Conditions

1.4.1 1D Boundary Condiions

class FEMpy.Boundaries.**BoundaryConditions**(*mesh*, *boundary_types*, *boundary_coords=None*, *dirichlet_fun=None*, *neumann_fun=None*, *robin_fun_q=None*, *robin_fun_p=None*, *coeff_fun=None*)
Defines all boundary conditions to be applied to a Poisson equation.

Takes in the coordinates for the boundary nodes and the boundary condition types and provides the treatments for each of the boundry condition types.

Parameters

- **mesh** (*{:class: FEMpy.Mesh.Interval1D, :class: FEMpy.Mesh.TriangularMesh2D}*) – A Mesh class defining the mesh and associated information matrices.
- **boundary_types** (array_like of str {'dirichlet', 'neumann', 'robin'}) – Boundary condition type for each coordinate in *boundary_coords*.
- **boundary_coords** (*array_like, optional*) – List of coordinate values of the boundary nodes. If not specified, will use the boundary node coordinates stored in *mesh*.

- **dirichlet_fun** (*function, optional*) – The Dirichlet boundary condition function *g‘(‘x*). Must be defined at the boundary values specified in *boundary_coords*.
- **neumann_fun** (*function, optional*) – The Neumann boundary condition function *r‘(‘x*). Must be defined at the bounadry values specified in *boundary_coords*.
- **robin_fun_q** (*function, optional*) – The Robin boundary condition function *q‘(‘x*). Must be defined at the boundary values specified in *boundary_coords*.
- **robin_fun_p** (*function, optional*) – The Robin boundary condition function *p‘(‘x*). Must be defined at the boundary values specified in *boundary_coords*.
- **coeff_fun** (*function, optional*) – Function name of the coefficient function *c‘(‘x*) in the Poisson equation. Required if Neumann or Robin boundary conditions are included.

Warning: Both end point boundary conditions cannot be Neumann as this may result in a loss of uniqueness of the solution.

Notes

- The Dirichlet boundary conditions are be defined as

$$u(x) = g(x); x = a \text{ or } b.$$

- The Neumann boundary conditions are defined as

$$\frac{\mathrm{d}}{\mathrm{d}x}u(x) = r(x); x = a \text{ or } b.$$

- The Robin boundary conditions are defined as

$$\frac{\mathrm{d}}{\mathrm{d}x}u(x) + q(x)u(x) = p(x); x = a \text{ or } b.$$

treat_dirichlet (*matrix*, *vector*)

Overwrites the appropriate entries in the stiffness matrix and load vector.

Corrects the stiffness matrix and load vector to properly incorporate the boundary conditions.

Parameters

- **matrix** (*ndarray*) – Finite element stiffness matrix.
- **vector** (*ndarray*) – Finite element load vector.

Returns

- **matrix** (*ndarray*) – Corrected finite element stiffness matrix with Dirichlet boundary conditions incorporated.
- **vector** (*ndarray*) – Corrected finite element load vector with Dirichlet boundary conditions incorporated.

treat_neumann (*vector*)

Overwrites the appropriate entries in the load vector.

Corrects the load vector to properly incorporate the boundary conditions.

Parameters **vector** (*ndarray*) – Finite element load vector.

Returns **vector** – Corrected finite element load vector with Neumann boundary conditions incorporated.

Return type ndarray

treat_robin(*matrix*, *vector*)

Overwrites the appropriate entries in the stiffness matrix and load vector.

Corrects the stiffness matrix and load vector to properly incorporate the boundary conditions.

Parameters

- **matrix** (*ndarray*) – Finite element stiffness matrix.
- **vector** (*ndarray*) – Finite element load vector.

Returns

- **matrix** (*ndarray*) – Corrected finite element stiffness matrix with Robin boundary conditions incorporated.
- **vector** (*ndarray*) – Corrected finite element load vector with Robin boundary conditions incorporated.

1.4.2 2D Boundary Conditions

class FEMpy.Boundaries.**BoundaryConditions2D**(*mesh*, *boundary_node_types*, *boundary_edge_types*, *boundary_node_coords=None*, *boundary_edge_coords=None*, *trial_basis=None*, *test_basis=None*, *dirichlet_fun=None*, *neumann_fun=None*, *robin_fun_q=None*, *robin_fun_p=None*, *coeff_fun=None*)

Defines all boundary conditions to be applied to a Poisson equation.

Takes in the coordinates for the boundary nodes and boundary edges and the boundary condition types for each and provides the treatments for each of the boundry condition types.

Parameters

- **mesh** (*{:class: FEMpy.Mesh.Interval1D, :class: FEMpy.Mesh.TriangularMesh2D}*) – A `Mesh` class defining the mesh and associated information matrices.
- **boundary_node_types** (array_like of str {'dirichlet', 'neumann', 'robin'}) – Boundary condition type for each coordinate in *boundary_node_coords*.
- **boundary_edge_types** (array_like of str {'dirichlet', 'neumann', 'robin'}) – Boundary condition type for each edge segment in *boundary_edge_coords*.
- **boundary_node_coords** (*array_like, optional*) – List of coordinate values of the boundary nodes. If not specified, will use the boundary node coordinates stored in *mesh*.
- **boundary_edge_coords** (*array_like, optional*) – List of grid coordinates for edge nodes. If not specified, will use the boundary edge coordinates stored in *mesh*.
- **trial_basis, test_basis** (*{:class: FEMpy.FEBasis.IntervalBasis1D, :class: FEMpy.FEBasis.TriangularBasis2D}, optional*) – A :class: *FEBasis* class defining the finite element basis functions for the trial and test bases. If Neumann boundary conditions included *test_basis* is required. If Robin boundary conditions included, then both bases are required.

- **dirichlet_fun** (*function, optional*) – The Dirichlet boundary condition function *g‘(‘x, y*). Must be defined at the boundary values specified in *boundary_coords*.
- **neumann_fun** (*function, optional*) – The Neumann boundary condition function *r‘(‘x, y*). Must be defined at the bounadry values specified in *boundary_coords*.
- **robin_fun_q** (*function, optional*) – The Robin boundary condition function *q‘(‘x, y*). Must be defined at the boundary values specified in *boundary_coords*.
- **robin_fun_p** (*function, optional*) – The Robin boundary condition function *p‘(‘x, y*). Must be defined at the boundary values specified in *boundary_coords*.
- **coeff_fun** (*function, optional*) – Function name of the coefficient function *c‘(‘x, y*) in the Poisson equation. Required if Neumann or Robin boundary conditions are included.

Warning: All edge boundary conditions cannot be Neumann as this may result in a loss of uniqueness of the solution.

Notes

- The Dirichlet boundary conditions are be defined as

$$u(x, y) = g(x, y); (x, y) \in \delta\Omega \setminus (\Gamma_1 \cup \Gamma_2).$$

- The Neumann boundary conditions are defined as

$$\nabla u(x, y) \cdot \hat{\mathbf{n}} = r(x, y); (x, y) \in \Gamma_1 \subseteq \delta\Omega,$$

- The Robin boundary conditions are defined as

$$\nabla u(x, y) \cdot \hat{\mathbf{n}} + q(x, y)u(x, y) = p(x, y); (x, y) \in \Gamma_2 \subseteq \delta\Omega.$$

treat_neumann (*vector*)

Overwrites the appropriate entries in the load vector.

Corrects the load vector to properly incorporate the boundary conditions.

Parameters **vector** (*ndarray*) – Finite element load vector.

Returns **vector** – Corrected finite element load vector with Neumann boundary conditions incorporated.

Return type ndarray

treat_robin (*matrix*, *vector*)

Overwrites the appropriate entries in the stiffness matrix and load vector.

Corrects the stiffness matrix and load vector to properly incorporate the boundary conditions.

Parameters

- **matrix** (*ndarray*) – Finite element stiffness matrix.
- **vector** (*ndarray*) – Finite element load vector.

Returns

- **matrix** (*ndarray*) – Corrected finite element stiffness matrix with Robin boundary conditions incorporated.

- **vector** (*ndarray*) – Corrected finite element load vector with Robin boundary conditions incorporated.

1.5 Finite Element Solvers

1.5.1 1D Poisson Equation

class FEMpy.Solvers.**Poisson1D**(*mesh*, *fe_trial_basis*, *fe_test_basis*, *boundary_conditions*)

Solves a one-dimensional Poisson equation.

Uses finite element methods to solve a Poisson differential equation of the form

$$-\frac{\mathrm{d}}{\mathrm{d}x}\left(c(x)\frac{\mathrm{d}}{\mathrm{d}x}u(x)\right) = f(x); a \leq x \leq b.$$

with a combination of Dirichlet, Neumann, or Robin boundary conditions.

Warning: Both end point boundary conditions cannot be Neumann as this may result in a loss of uniqueness of the solution.

Parameters

- **mesh** (*FEMpy.Mesh.Interval1D*) – A `Mesh` class defining the mesh and associated information matrices.
- **fe_trial_basis, fe_test_basis** (*FEMpy.FEBasis.IntervalBasis1D*) – A `FEBasis` class defining the finite element basis functions for the trial and test bases.
- **boundary_conditions** (*FEMpy.Boundaries.BoundaryConditions*) – A `BoundaryConditions` class defining the boundary conditions on the domain.

Examples

```
>>> import numpy as np
>>> from FEMpy import Interval1D, IntervalBasis1D, BoundaryConditions, Poisson1D
>>> mesh = Interval1D(0, 1, h=1/2, basis_type='linear')
>>> basis = IntervalBasis1D('linear')
>>> dirichlet_funct = lambda x: 0 if x == 0 else np.cos(1)
>>> bcs = BoundaryConditions(mesh, ('dirichlet', 'dirichlet'), dirichlet_
↪fun=dirichlet_funct)
>>> coefficient_funct = lambda x: np.exp(x)
>>> source_funct = lambda x: -np.exp(x) * (np.cos(x) - 2*np.sin(x) - x*np.cos(x) -
↪ x*np.sin(x))
>>> poisson_eq = Poisson1D(mesh, basis, basis, bcs)
>>> poisson_eq.solve(coefficient_funct, source_funct)
array([0.        , 0.44814801, 0.54030231])
```

fe_solution(*x*, *local_sol*, *vertices*, *derivative_order*)

Defines the functional solution piecewise on the finite on the finite elements.

Uses the solution vector and the basis function to define a piecewise continuous solution over the element.

Parameters

- **x** (*float or array_like*) – A value or array of points to evaluate the function on.

- **local_sol** (*array_like*) – Finite element solution node vector local to the element *En*.
- **vertices** (*array_like*) – Global node coordinates for the mesh element *En*.
- **derivative_order** (*int*) – The derivative order to take the basis function to.

Returns Solution at all points in *x* in element.

Return type float

h1_seminorm_error(*diff_exact_sol*)

The H1 semi-norm error of the finite element solution compared against the given analyatical solution.

Parameters **diff_exact_sol** (*function*) – The first derivative of the analytical solution to the Poisson equation.

Returns The H1 semi-norm error of the finite element solution over the domain evaluated element-wise.

Return type float

l2_error(*exact_sol*)

The L2 norm error of the finite element solution compared against the given analytical solution.

Parameters **exact_sol** (*function*) – The analytical solution to the Poisson equation.

Returns The L2 norm error of the finite element solution over the domain evaluated element-wise.

Return type float

l_inf_error(*exact_sol*)

Computes the L-infinity norm error.

Computes the L-infinity norm error using the exact solution and the finite element function *fe_solution*.

Parameters **exact_sol** (*function*) – The analytical solution to compare the finite element solution against.

Returns The L-infinity norm error of the finite element solution over the domain evaluated element-wise.

Return type float

solve(*coeff_fun*, *source_fun*)

Method that performs the finte element solution algorithm.

Calls the assembly functions *FEMpy.Assemblers.assemble_matrix* and *FEMpy.Assemblers.assemble_vector* to create the stiffness matrix and load vector respectively. Then, applies the boundary condition treatments to the matrix and vector. Finally, solves the linear system $A\mathbf{x} = \mathbf{b}$.

Parameters

- **coeff_fun** (*function*) – Function name of the coefficient function *c‘(‘x*) in the Poisson equation.
- **source_fun** (*function*) – The nonhomogeneous source function *f‘(‘x*) of the Poisson equation.

Returns The nodal solution vector.

Return type ndarray

1.5.2 2D Poisson Equation

class FEMpy.Solvers.**Poisson2D**(*mesh*, *fe_trial_basis*, *fe_test_basis*, *boundary_conditions*)

Solves a two-dimensional Poisson equation.

Uses finite element methods to solve a Poisson differential equation of the form

$$-\nabla \left(c(\mathbf{x}) \cdot \nabla u(\mathbf{x}) \right) = f(\mathbf{x}); \mathbf{x} \in \Omega$$

with a combination of Dirichlet, Neumann, or Robin boundary conditions.

> **Warning:** All edge boundary conditions cannot be Neumann as this may result in a loss of uniqueness of the solution.

Parameters

- **mesh** (*FEMpy.Mesh.TriangularMesh2D*) – A Mesh class defining the mesh and associated information matrices.
- **fe_trial_basis, fe_test_basis** (*FEMpy.FEBasis.IntervalBasis1D*) – A FEBasis class defining the finite element basis functions for the trial and test bases.
- **boundary_conditions** (*FEMpy.Boundaries.BoundaryConditions2D*) – A :class: *BoundaryConditions* class defining the boundary conditions on the domain.

Examples

```
>>> import numpy as np
>>> from FEMpy import TriangularMesh2D, TriangularBasis2D, BoundaryConditions2D,␣
↪Poisson2D
>>> left, right, bottom, top = -1, 1, -1, 1
>>> h = 1
>>> def dirichlet_funct(coord):
>>>     x, y = coord
>>>     if x == -1:
>>>         return np.exp(-1 + y)
>>>     elif x == 1:
>>>         return np.exp(1 + y)
>>>     elif y == 1:
>>>         return np.exp(x + 1)
>>>     elif y == -1:
>>>         return np.exp(x - 1)
>>> coeff_funct = lambda coord: 1
>>> source_funct = lambda coord: -2 * np.exp(coord[0] + coord[1])
>>> mesh = TriangularMesh2D(left, right, bottom, top, h, h, 'linear')
>>> basis = TriangularBasis2D('linear')
>>> boundary_node_types = ['dirichlet'] * mesh.boundary_nodes.shape[1]
>>> boundary_edge_types = ['dirichlet'] * (mesh.boundary_edges.shape[1]-1)
>>> bcs = BoundaryConditions2D(mesh, boundary_node_types, boundary_edge_types,␣
↪dirichlet_fun=dirichlet_funct)
>>> poisson_eq = Poisson2D(mesh, basis, basis, bcs)
>>> poisson_eq.solve(coeff_funct, source_funct)
array([0.13533528, 0.36787944, 1., 0.36787944, 1.,  2.71828183, 1., 2.71828183, 7.
↪3890561])
```

fe_solution(*coords*, *local_sol*, *vertices*, *derivative_order*)
Defines the functional solution piecewise on the finite on the finite elements.

Uses the solution vector and the basis function to define a piecewise continuous solution over the element.

Parameters

- **coords** (*float or array_like*) – A value or array of points to evaluate the function on.
- **local_sol** (*array_like*) – Finite element solution node vector local to the element *En*.
- **vertices** (*array_like*) – Global node coordinates for the mesh element *En*.
- **derivative_order** (*tuple of int*) – The derivative orders in the x- and y-directions to take the basis function to.

Returns Solution at all points in *coords* in element.

Return type float

h1_seminorm_error(*diff_exact_sol*)
The H1 semi-norm error of the finite element solution compared against the given analyatical solution.

Parameters **diff_exact_sol** (*tuple of function*) – A tuple of first derivatives in the x- and the y-directions the analytical solution to the Poisson equation respectively.

Returns The full H1 semi-norm error of the finite element solution over the domain evaluated element-wise.

Return type float

l2_error(*exact_sol*)
The L2 norm error of the finite element solution compared against the given analytical solution.

Parameters **exact_sol** (*function*) – The analytical solution to the Poisson equation.

Returns The L2 norm error of the finite element solution over the domain evaluated element-wise.

Return type float

l_inf_error(*exact_sol*)
Computes the L-infinity norm error.

Computes the L-infinity norm error using the exact solution and the finite element function *fe_solution*.

Parameters **exact_sol** (*function*) – The analytical solution to compare the finite element solution against.

Returns The L-infinity norm error of the finite element solution over the domain evaluated element-wise.

Return type float

solve(*coeff_fun*, *source_fun*)
Method that performs the finte element solution algorithm.

Calls the assembly functions *FEMpy.Assemblers.assemble_matrix* and *FEMpy.Assemblers.assemble_vector* to create the stiffness matrix and load vector respectively. Then, applies the boundary condition treatments to the matrix and vector. Finally, solves the linear system $A\mathbf{x} = \mathbf{b}$.

Parameters

- **coeff_fun** (*function*) – Function name of the coefficient function *c‘(‘x*, *y*) in the Poisson equation.

- **source_fun** (*function*) – The nonhomogeneous source function *f(‘x, y)* of the Poisson equation.

1.6 Assemblers

FEMpy.Assemblers.**assemble_matrix**(*coeff_funct*, *mesh*, *trial_basis*, *test_basis*, *derivative_order_trial*, *derivative_order_test*)

Construct the finite element stiffness matrix. Meant to be used in a `FEMpy.Solvers` *solve* method.

Parameters

- **coeff_funct** (*function*) – Function name of the coefficient function *c(x(,y,...))* in a Poisson equation.
- **mesh** ({*FEMpy.Mesh.Interval1D*, *FEMpy.Mesh.TriangularMesh2D*}) – A `Mesh` class defining the mesh and associated information matrices.
- **trial_basis, test_basis** (*{:class: FEMpy.FEBasis.IntervalBasis1D, :class: FEMpy.FEBasis.TriangularBasis2D}*) – A :class: *FEBasis* class defining the finite element basis functions for the trial and test bases.
- **derivative_order_trial, derivative_order_test** (*int or tuple of int*) – The derivative order to be applied to the finite element basis functions. If basis function is one-dimensional, this should be specified as an int. Otherwise, the derivative order should be specified as a tuple with the orders corresponding to the coordinate axes in the basis functions. e.g., (`x_order`, `y_order`,...).

Returns The finite element stiffness matrix as a row-based linked list sparse matrix.

Return type sparse matrix

FEMpy.Assemblers.**assemble_vector**(*source_funct*, *mesh*, *test_basis*, *derivative_order_test*)

Constructs the finite element load vector. Meant to be used in a `FEMpy.Solvers` *solve* method.

Parameters

- **source_funct** (*function*) – The nonhomogeneous source function *f(x(,y,...))* of the Poisson equation.
- **mesh** ({*FEMpy.Mesh.Interval1D*, *FEMpy.Mesh.TriangularMesh2D*}) – A `Mesh` class defining the mesh and associated information matrices.
- **test_basis** (*{:class: FEMpy.FEBasis.IntervalBasis1D, :class: FEMpy.FEBasis.TriangularBasis2D}*) – A :class: *FEBasis* class defining the finite element basis functions for the test basis.
- **derivative_order_test** (*int or tuple of int*) – The derivative order to be applied to the finite element basis function. If basis function is one-dimensional, this should be specified as an int. Otherwise, the derivative order should be specified as a tuple with the orders corresponding to the coordinate axes in the basis functions. e.g., (`x_order`, `y_order`,...).

Returns The finite element load vector.

Return type ndarray

The following section was created from `tutorials/quickstart.ipynb`.

1.7 Quickstart

This notebook was made with the following version of FEMpy:

[1]:
```
import FEMpy
FEMpy.__version__
```

[1]:
```
'1.0'
```

FEMpy can be used to solve Poisson equations on 1D and 2D domains. For this example, let us consider the problem:

$$-\frac{\mathrm{d}}{\mathrm{d}x}\left(e^x\frac{\mathrm{d}}{\mathrm{d}x}u(x)\right) = -e^x[\cos x - 2\sin x - x\cos x - x\sin x]; x \in [0,1]$$

$$u(0) = 0, u(1) = \cos(1)$$

To start, we will need to import our necessary aditional packages.

[2]:
```
import numpy as np
```

Let us define our coefficient and source functions,

[3]:
```
coefficient_function = lambda x: np.exp(x)
source_function = lambda x: -np.exp(x) * (np.cos(x) - 2*np.sin(x) - x*np.cos(x) -␣
↪x*np.sin(x))
```

And our boundary condition,

[4]:
```
def dirichlet_function(x):
    if x == 0:
        return 0
    elif x == 1:
        return np.cos(1)
```

Under the Galerkin formulation, we can write our differiential equation as

$$\int_a^b cu'v'\mathrm{d}x = \int_a^b fv\mathrm{d}x$$

where $v(x)$ is a test function. We then can choose our test and trial basis functions such that

$$u_h \in \{\phi \in C[a,b] \mid \phi(x) \text{ linear on each } [x_n, x_{n+1}]; (n = 1, 2, \ldots, N)\}$$

and

$$\int_a^b cu_h'v_h'\mathrm{d}x = \int_a^b fv_h\mathrm{d}x$$

for any $v_h \in U_h$ Thus, we can define our test and trial basis function basis functions by:

[5]:
```
basis = FEMpy.IntervalBasis1D('linear')
```

We can set up our mesh using a step size of $h = 1/4$

[6]:
```
mesh = FEMpy.Interval1D(left=0, right=1, h=1/4, basis_type='linear')
```

The boundary conditions are defined by:

[7]:
```
bcs = FEMpy.BoundaryConditions(mesh, boundary_types=('dirichlet', 'dirichlet'),␣
↪dirichlet_fun=dirichlet_function)
```

Finally, we can input our mesh, basis functions, and boundary conditions into our Poisson equation then call our solver.

[8]:
```
poisson_eq = FEMpy.Poisson1D(mesh, fe_test_basis=basis, fe_trial_basis=basis,␣
↪boundary_conditions=bcs)
poisson_eq.solve(coefficient_function, source_function)
```

[8]:
```
array([0.        , 0.24411715, 0.44112525, 0.55036422, 0.54030231])
```

The following section was created from `tutorials/triangular2d.ipynb`.

1.8 2D Poisson Equation with Triangular Elements

This tutorial was made with the following version of FEMpy:

[1]:
```
import FEMpy
FEMpy.__version__
```

[1]:
```
'1.0'
```

FEMpy can solve 2D domains using similar inputs as in the 1D case. Here we will solve the following problem

$$-\nabla \cdot (c(x,y)\nabla u(x,y)) = f(x,y); (x,y) \in \Omega = [-1,1] \times [-1,1]$$

where $c(x,y) = 1$ and $f(x,y) = -2e^{x+y}$ and with boundary conditions

$$u(x,y) = e^{-1+y} \text{ for } x = -1, y \in \delta\Omega \setminus \Gamma_1$$

$$u(x,y) = e^{1+y} \text{ for } x = 1, y \in \delta\Omega \setminus \Gamma_1$$

$$u(x,y) = e^{x+1} \text{ for } y = 1, x \in \delta\Omega \setminus \Gamma_1$$

$$\nabla u(x,y) \cdot \hat{\mathbf{n}} = -e^{x-1} \text{ for } y = -1, x \in \Gamma_1 \subseteq \delta\Omega$$

Let us define our necessary functions

[2]:
```
import numpy as np


def coefficient_function(coords):
    return 1


def source_function(coord):
    x, y = coord
    return -2 * np.exp(x + y)


def dirichlet_function(coord):
    x, y = coord
    if x == -1:
        return np.exp(-1 + y)
    elif x == 1:
        return np.exp(1 + y)
```

(continues on next page)

(continued from previous page)

```
    elif y == 1:
        return np.exp(x + 1)
    elif y == -1:
        return np.exp(x - 1)


def neumann_function(coord):
    x, y = coord
    if y == -1:
        return -np.exp(x - 1)
```

Let's choose a linear basis for our problem and set up our grid to have step sizes of $h_1 = h_2 = 1/4$.

[3]:
```
basis = FEMpy.TriangularBasis2D('linear')
mesh = FEMpy.TriangularMesh2D(-1, 1, -1, 1, h1=1/4, h2=1/4, basis_type='linear')
```

Before we can set up our boundary conditions, we will need to define our boundary node and edge types. These can most easily be done by a list of strings indicating the boundary condition type for each boundary node and edge, respectively.

[4]:
```
boundary_node_types = ['dirichlet', *['neumann']*7, *['dirichlet']*9, *['dirichlet
↪']*8, *['dirichlet']*7]
boundary_edge_types = [*['neumann']*8, *['dirichlet']*8, *['dirichlet']*8, *[
↪'dirichlet']*8]
```

Now, we can define our boundary conditions, remembering to include the test basis function and the coefficient function since we have Neumann boundary conditions.

[5]:
```
bcs = FEMpy.BoundaryConditions2D(mesh, boundary_node_types, boundary_edge_types,
                                 test_basis=basis,
                                 dirichlet_fun=dirichlet_function,
                                 neumann_fun=neumann_function,
                                 coeff_fun=coefficient_function)
```

We can input our mesh, basis functions, and boundary conditions into our Poisson equation and call our solver:

[6]:
```
poisson_eq = FEMpy.Poisson2D(mesh, fe_test_basis=basis, fe_trial_basis=basis,␣
↪boundary_conditions=bcs)
poisson_eq.solve(coefficient_function, source_function)
```

[6]:
```
array([0.13533528, 0.17377394, 0.22313016, 0.2865048 , 0.36787944,
       0.47236655, 0.60653066, 0.77880078, 1.        , 0.17325304,
       0.22277292, 0.28625562, 0.36770659, 0.47224895, 0.60645367,
       0.77875452, 0.99997833, 1.28402542, 0.22221544, 0.28584593,
       0.36741281, 0.47204193, 0.60631008, 0.77865671, 0.99991359,
       1.28398499, 1.64872127, 0.28526803, 0.36698257, 0.4717327 ,
       0.60609224, 0.77850475, 0.99980769, 1.28391054, 1.64866766,
       2.11700002, 0.36639009, 0.47130853, 0.60579719, 0.7783016 ,
       0.99966716, 1.28381118, 1.64859408, 2.11694086, 2.71828183,
       0.47072395, 0.60541827, 0.77805796, 0.99950801, 1.28370351,
       1.64851667, 2.11687951, 2.71822603, 3.49034296, 0.60490964,
       0.77779468, 0.99936654, 1.28362138, 1.64846308, 2.11683828,
       2.71818738, 3.49029942, 4.48168907, 0.77755761, 0.99934246,
       1.28364454, 1.64848877, 2.11685491, 2.71819212, 3.49029095,
       4.48166518, 5.75460268, 1.        , 1.28402542, 1.64872127,
       2.11700002, 2.71828183, 3.49034296, 4.48168907, 5.75460268,
       7.3890561 ])
```

We can also look at the L^∞ and L^2 norm errors as well as the H^1 semi-norm error associated with our solution as compared against the analytical solution of

$$u(x, y) = e^{x+y}$$

[7]:
```
def analytic_solution(coord):
    x, y = coord
    return np.exp(x + y)


def dx_analytic_solution(coord):
    x, y = coord
    return np.exp(x + y)


def dy_analytic_solution(coord):
    x, y = coord
    return np.exp(x + y)


L_inf_err = poisson_eq.l_inf_error(analytic_solution)
L_2_err = poisson_eq.l2_error(analytic_solution)
H_1_err = poisson_eq.h1_seminorm_error((dx_analytic_solution, dy_analytic_solution))

from IPython.display import display, Math
display(Math('\|L^\infty\| = {l_inf:.3e}'.format(l_inf=L_inf_err)))
display(Math('\|L^2\| = {l_2:.3e}'.format(l_2=L_2_err)))
display(Math('|H^1| = {h_1:.3e}'.format(h_1=H_1_err)))
```

$$\|L^\infty\| = 3.176e+00$$

$$\|L^2\| = 7.357e-04$$

$$|H^1| = 1.417e-02$$

The following section was created from `tutorials/errors.ipynb`.

1.9 Error calculations

FEMpy provides the ability to compute the L^∞ and L^2 norm errors as well as the H^1 semi-norm error.

This tutorial was made with the following version of FEMpy:

[1]:
```
import FEMpy
FEMpy.__version__
```

[1]:
```
'1.0'
```

Let us examine the error of

$$-\frac{\mathrm{d}}{\mathrm{d}x}\left(e^x \frac{\mathrm{d}}{\mathrm{d}x} u(x)\right) = -e^x[\cos x - 2\sin x - x\cos x - x\sin x]; x \in [0, 1]$$

$$u(0) = 0, u'(1) = \cos(1) - \sin(1)$$

as we vary the mesh step size h.

[2]:
```
import numpy as np


def coefficient_function(x):
    return  np.exp(x)


def source_function(x):
    return -np.exp(x) * (np.cos(x) - 2*np.sin(x) - x*np.cos(x) - x*np.sin(x))


def dirichlet_function(x):
    if x == 0:
        return 0


def neumann_function(x):
    if x == 1:
        return np.cos(1) - np.sin(1)
```

We will need the analytical solution to our problem for the L^∞ and L^2 norm errors and the derivative of the solution for the H^1 semi-norm.

[3]:
```
def analytical_sol(x):
    return x * np.cos(x)


def dx_analytical_sol(x):
    return np.cos(x) - x * np.sin(x)
```

We will vary our mesh size for $h \in \left\{\frac{1}{4}, \frac{1}{8}, \frac{1}{16}, \frac{1}{32}, \frac{1}{128}, \frac{1}{256}\right\}$.

[4]:
```
h_list = [1/(2**n) for n in np.arange(2, 9)]
```

For our case we will use quadratic finite elements

[5]:
```
basis = FEMpy.IntervalBasis1D('quadratic')
```

Now we can iterate through our mesh sizes and store our errors.

[6]:
```
L_inf_err = []
L2_err = []
H1_err = []
for h in h_list:
    mesh = FEMpy.Interval1D(0, 1, h, 'quadratic')
    bcs = FEMpy.BoundaryConditions(mesh, ('dirichlet', 'neumann'), dirichlet_
↪fun=dirichlet_function, neumann_fun=neumann_function, coeff_fun=coefficient_
↪function)

    poisson_eq = FEMpy.Poisson1D(mesh, fe_trial_basis=basis, fe_test_basis=basis,␣
↪boundary_conditions=bcs)
    poisson_eq.solve(coefficient_function, source_function)

    L_inf_err.append(poisson_eq.l_inf_error(analytical_sol))
    L2_err.append(poisson_eq.l2_error(analytical_sol))
    H1_err.append(poisson_eq.h1_seminorm_error(dx_analytical_sol))
```

To display our results we can use a pandas dataframe or an astropy table.

[7]:
```
from astropy.table import Table

error_table = Table([h_list, L_inf_err, L2_err, H1_err], names=['h', 'L_inf Norm Error
↪', 'L_2 Norm Error', 'H_1 Semi-norm Error'],)
error_table['h'].format = '.4f'; error_table['L_inf Norm Error'].format = '.4e';␣
↪error_table['L_2 Norm Error'].format = '.4e'; error_table['H_1 Semi-norm Error'].
↪format = '.4e'
error_table
```

[7]:
```
<i>Table length=7</i>
<table id="table133885012620344" class="table-striped table-bordered table-condensed">
<thead><tr><th>h</th><th>L_inf Norm Error</th><th>L_2 Norm Error</th><th>H_1 Semi-
↪norm Error</th></tr></thead>
<thead><tr><th>float64</th><th>float64</th><th>float64</th><th>float64</th></tr></
↪thead>
<tr><td>0.2500</td><td>3.3279e-04</td><td>2.1050e-04</td><td>5.4213e-03</td></tr>
<tr><td>0.1250</td><td>3.9288e-05</td><td>2.6147e-05</td><td>1.3534e-03</td></tr>
<tr><td>0.0625</td><td>4.7533e-06</td><td>3.2632e-06</td><td>3.3823e-04</td></tr>
<tr><td>0.0312</td><td>5.8395e-07</td><td>4.0774e-07</td><td>8.4550e-05</td></tr>
<tr><td>0.0156</td><td>7.2344e-08</td><td>5.0962e-08</td><td>2.1137e-05</td></tr>
<tr><td>0.0078</td><td>9.0022e-09</td><td>6.3701e-09</td><td>5.2842e-06</td></tr>
<tr><td>0.0039</td><td>1.1227e-09</td><td>7.9626e-10</td><td>1.3211e-06</td></tr>
</table>
```

Index