pyiga Documentation

Release 0.1.0

Clemens Hofreither

Jan 20, 2023

Contents

CHAPTER 1

User Guide

```
[1]: %pylab inline
     from pyiga import bspline, geometry, vis
```

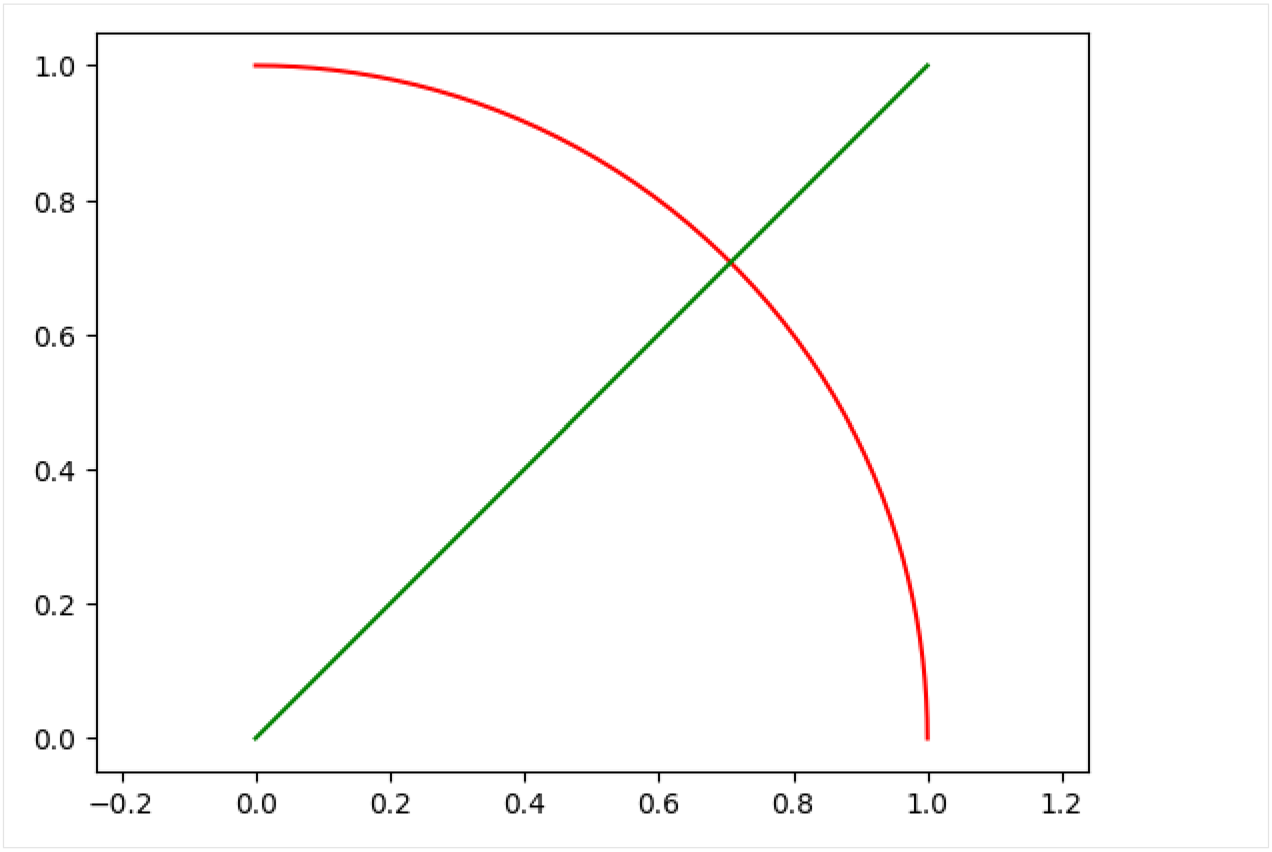
```
Populating the interactive namespace from numpy and matplotlib
```

1.1 Geometry manipulation in `pyiga`

We can define line segments or circular arcs using the builtin functions from the `geometry` module (see its documentation for details on all the functions used here). All kinds of geometries can be conveniently plotted using the `vis.plot_geo()` function.

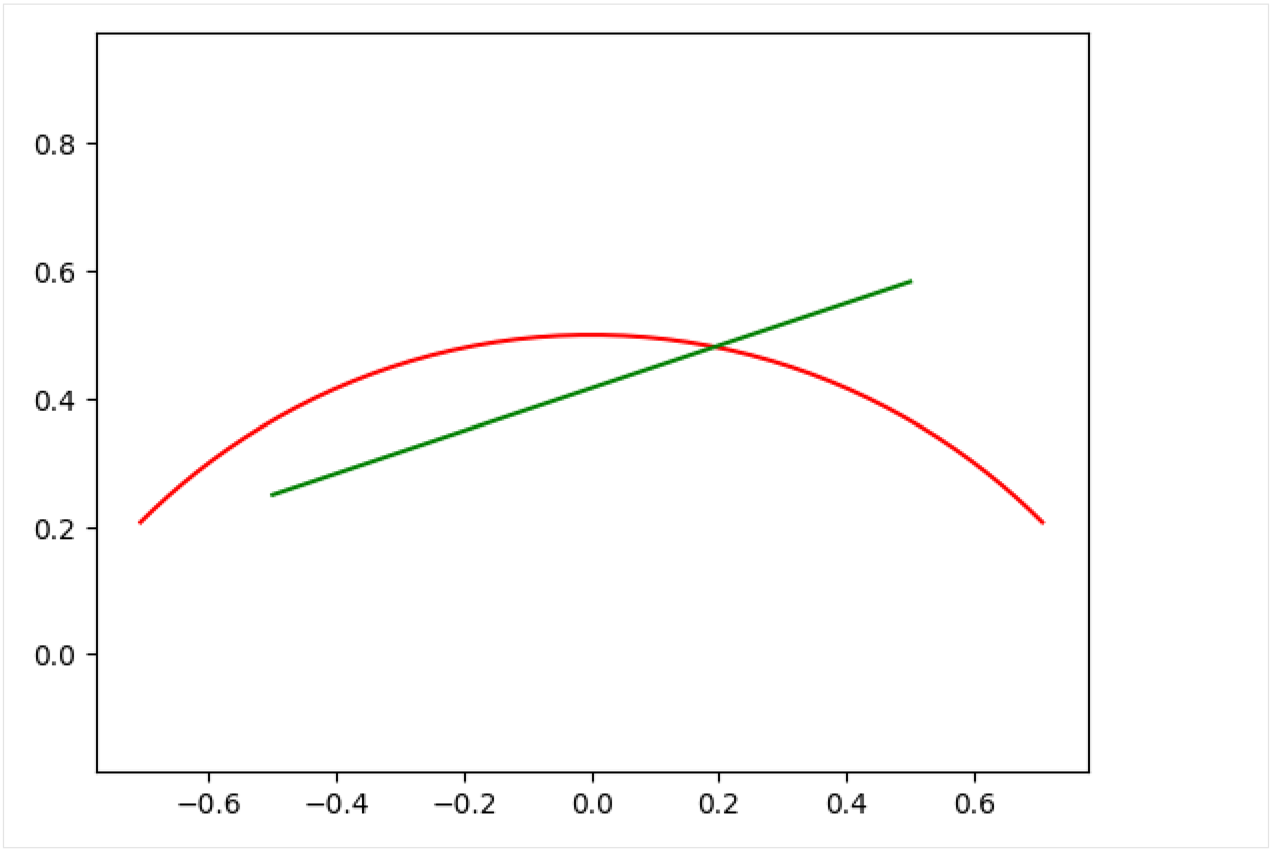
```
[2]: f = geometry.circular_arc(pi/2)
     g = geometry.line_segment([0,0], [1,1])

     vis.plot_geo(f, color='red')
     vis.plot_geo(g, color='green')
     axis('equal');
```



Geometries can be translated, rotated or scaled:

```
[3]: vis.plot_geo(f.rotate_2d(pi/4).translate([0,-0.5]), color='red')
     vis.plot_geo(g.scale([1,1/3]).translate([-.5,.25]), color='green')
     axis('equal');
```



We can combine the univariate geometry functions $f(y)$ and $g(x)$ to create biviariate ones using, for instance, the outer sum

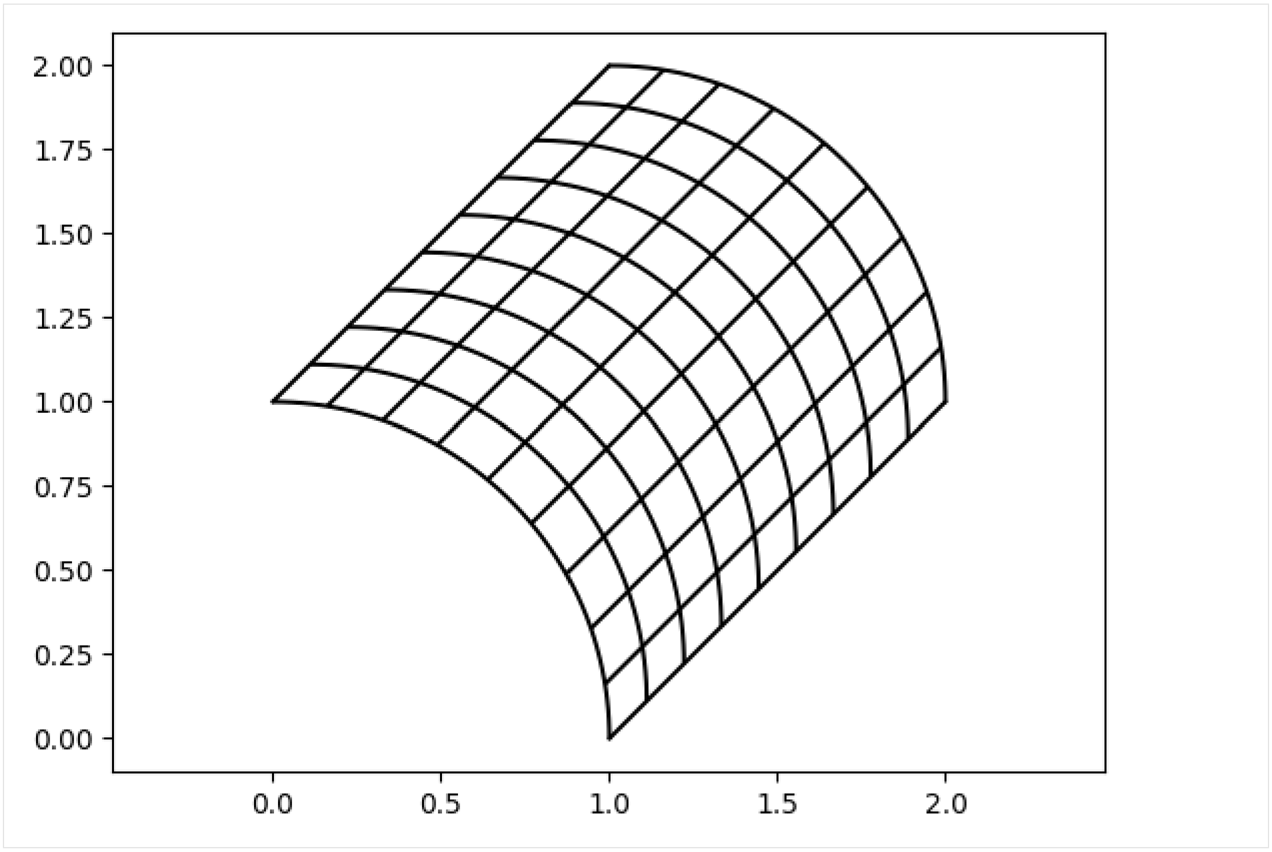
$$(f \oplus g)(x, y) = f(y) + g(x)$$

or the outer product

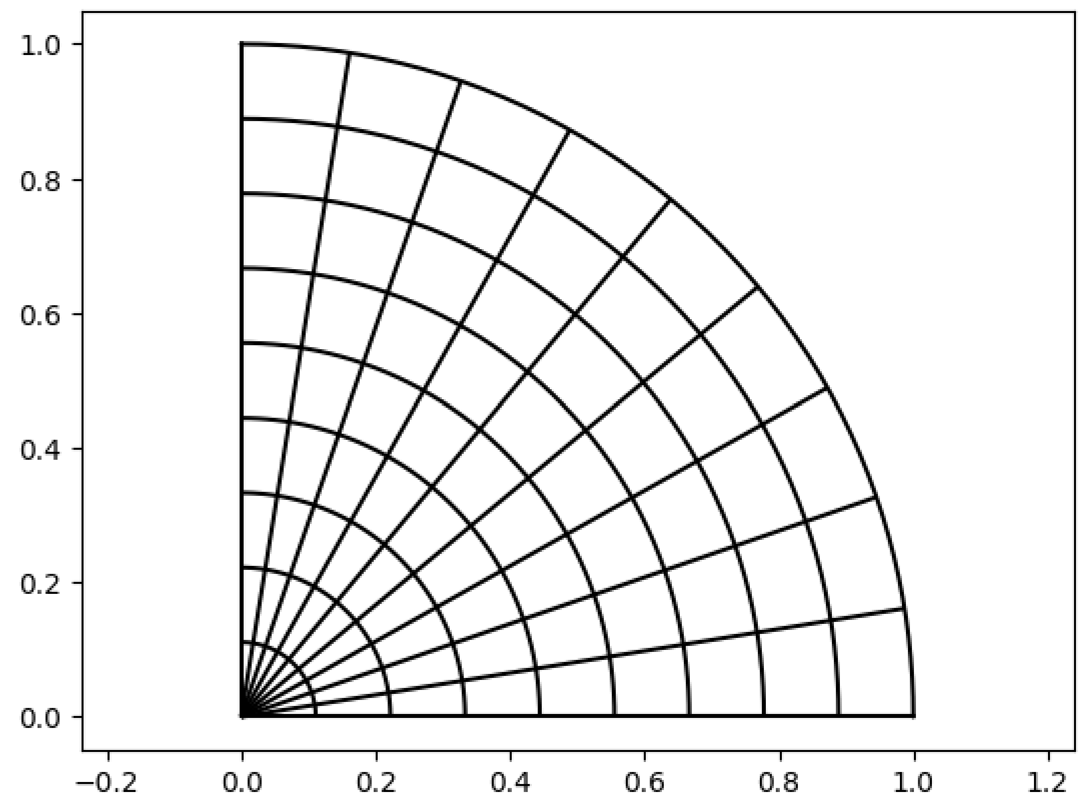
$$(f \otimes g)(x, y) = f(y) * g(x).$$

Here, both the addition and the product have to be understood in a componentwise fashion if f and/or g are vector-valued (as they are in our example).

[4]:
```
vis.plot_geo(geometry.outer_sum(f, g))
axis('equal');
```

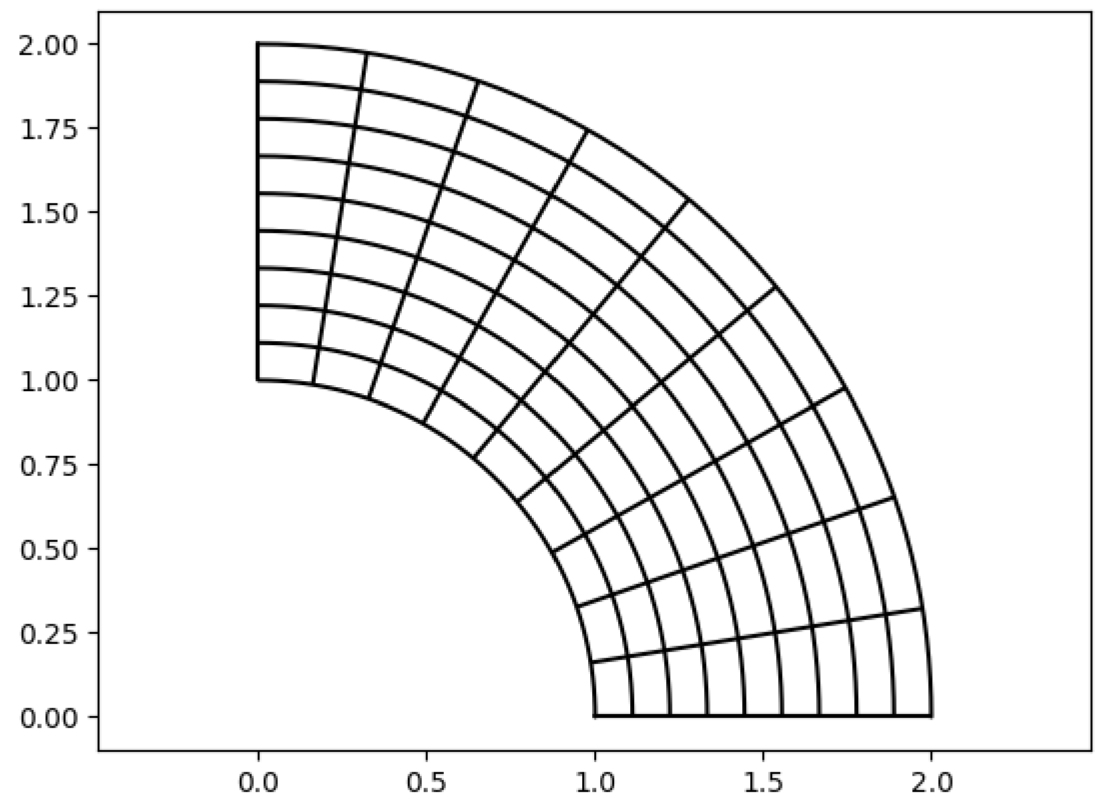


[5]:
```
vis.plot_geo(geometry.outer_product(f, g))
axis('equal');
```



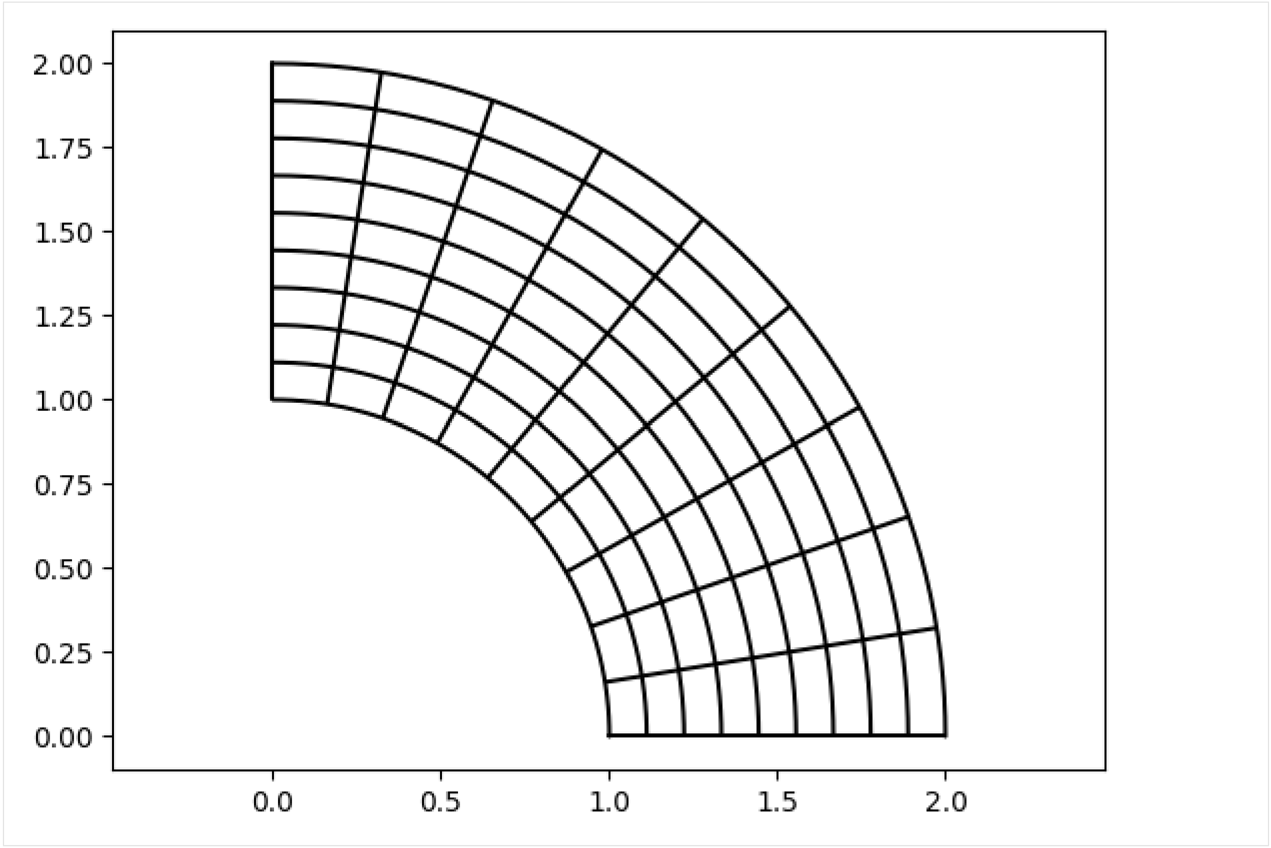
The last outer product has a singularity at the origin because multiplying with $g(0) = (0, 0)$ forces all points into $(0, 0)$. We can translate it first to avoid this, creating a quarter annulus domain in the process:

```
[6]: vis.plot_geo(geometry.outer_product(f, g.translate([1,1])))
     axis('equal');
```



In this example, the second operand was a line segment from $(1, 1)$ to $(2, 2)$. Since numpy-style broadcasting works for all these operations, we can also simply define the second operand as a linear scalar function ranging from 1 to 2 to obtain the same effect:

```
[7]: vis.plot_geo(geometry.outer_product(f, geometry.line_segment(1, 2)))
     axis('equal');
```

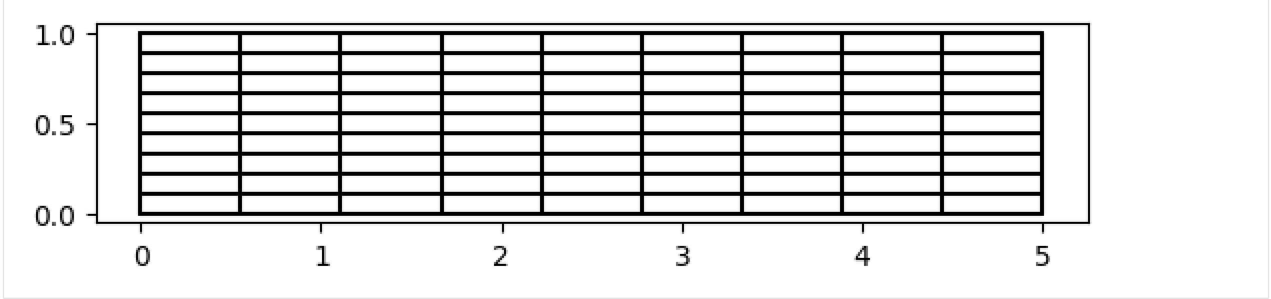


We can also generate tensor product geometries; here each input function f and g has to be scalar such that the resulting output function F is 2D-vector-valued, namely,

$$F(x, y) = (g(x), f(y)).$$

However, you can also use this with higher-dimensional inputs, for instance to build a 3D cylinder on top of a 2D domain.

```
[8]: G = geometry.tensor_product(geometry.line_segment(0,1),
                                 geometry.line_segment(0,5, intervals=3))
     vis.plot_geo(G)
     axis('scaled');
```



We can also define geometries through user-defined functions, where we have to specify the domain:

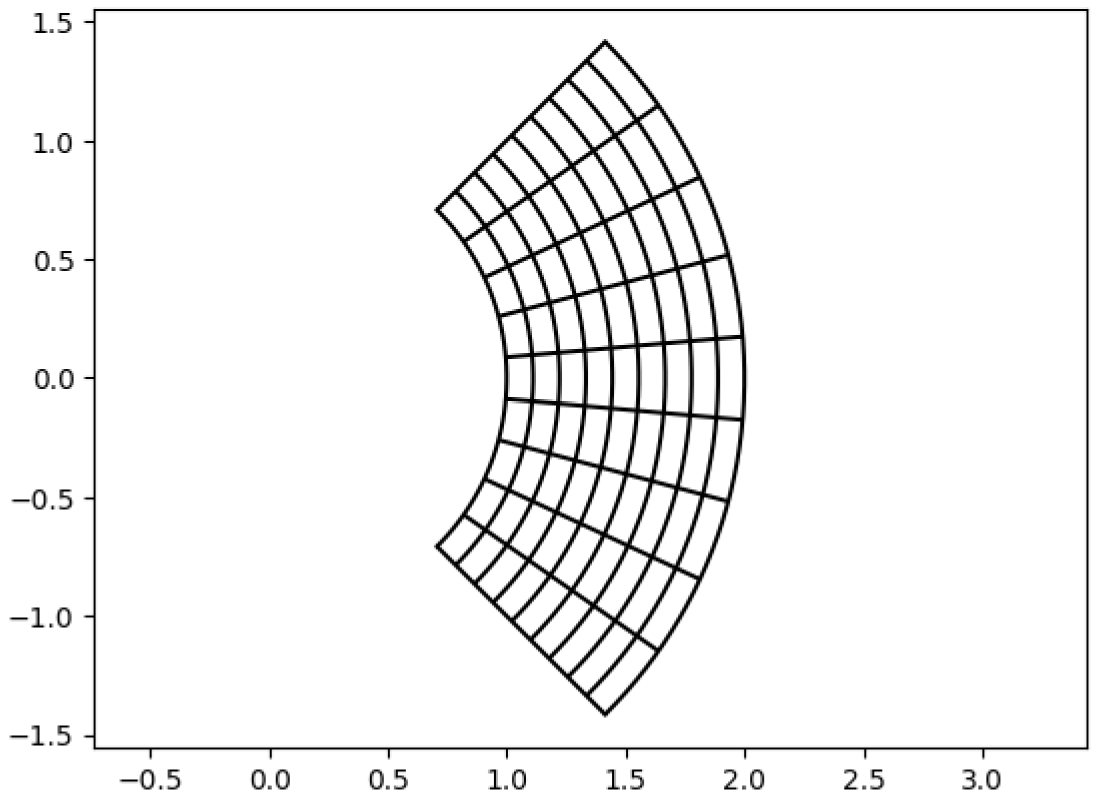
```
[9]: def f(x, y):
         r = 1 + x
         phi = (y - 0.5) * np.pi/2
```

(continues on next page)

(continued from previous page)

```
    return (r * np.cos(phi), r * np.sin(phi))

f_func = geometry.UserFunction(f, [[0,1],[0,1]])
vis.plot_geo(f_func)
axis('equal');
```

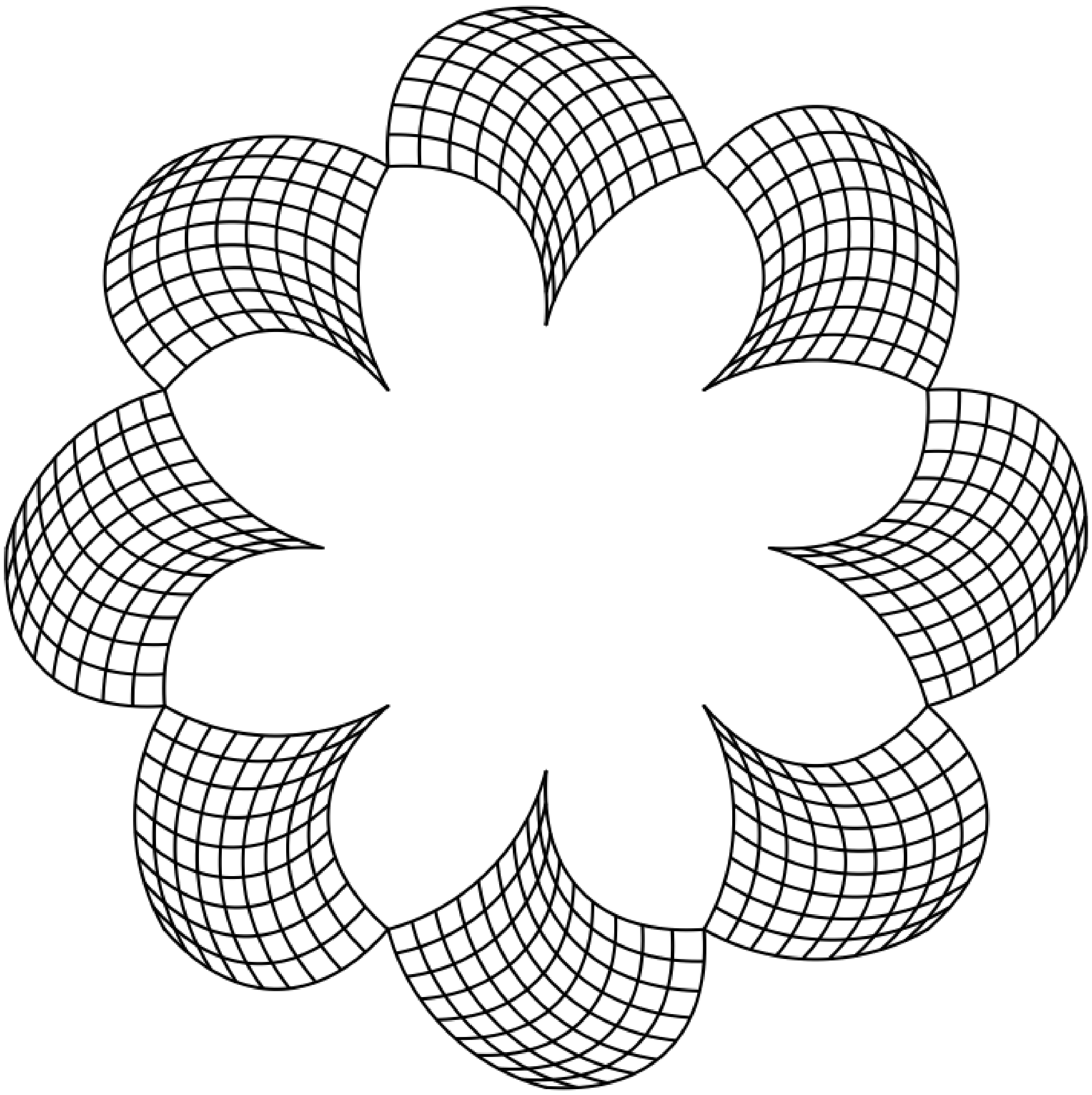


Finally, a bit of fun with translated and rotated outer products of circular arcs:

```
[10]: figsize(10,10)
G1 = geometry.circular_arc(pi/3).translate((-1,0)).rotate_2d(-pi/6)
G2 = G1.scale(-1).rotate_2d(pi/2)
G1 = G1.translate((1,1))
G2 = G2.translate((1,1))

G = geometry.outer_product(G1, G2).translate((-1,-2)).rotate_2d(3*pi/4).translate((0,
↪1))

for i in range(8):
    vis.plot_geo(G.rotate_2d(i*pi/4))
axis('equal');
axis('off');
```



1.2 Assembling custom forms

`pyiga` makes it possible to define custom linear or bilinear forms and have highly efficient code for assembling them automatically generated, similar to the facilities in FEniCS or NGSolve.

1.2.1 Assembling variational forms from their textual description

The most convenient interface for assembling custom forms is by calling `pyiga.assemble.assemble()` with a string describing the problem as the first argument. Here is a simple but complete example:

```
from pyiga import bspline, geometry, assemble

kvs = (bspline.make_knots(3, 0.0, 1.0, 30),
       bspline.make_knots(3, 0.0, 1.0, 30))
geo = geometry.quarter_annulus()
A = assemble.assemble('inner(grad(u), grad(v)) * dx', kvs, geo=geo)
```

Here we define a cubic 2D tensor product spline space with 30 intervals per coordinate direction and then assemble a standard Laplace variational form over a quarter annulus geometry. The result is the sparse stiffness matrix *A*.

The string representing the integrand occurring in this example should be mostly self-explanatory: it represents the inner product of the gradients of the trial and test functions *u* and *v*, multiplied by the volume measure *dx*.

In a similar way we can assemble linear functionals, such as we would need for computing a right-hand side for a discretized problem:

```
def f(x, y): return np.cos(x) * np.exp(y)
rhs = assemble.assemble('f * v * dx', kvs, geo=geo, f=f)
```

Note that we specified the additional input function *f* as a keyword argument to `assemble()`. The `assemble()` function automatically determined that we are assembling a linear functional from the fact that we are referring to only one basis function, *v*, in the description of our form. The result is not a 1D vector as one might expect, but a 2D array whose shape is

```
rhs.shape == (kvs[0].numdofs, kvs[1].numdofs)
```

However, it is easy to convert it to a 1D vector by calling `rhs.ravel()`.

Vector-valued problems are realized by informing `assemble()` of the number of components of the basis functions via the *bfuns* argument. For instance, the stiffness matrix for the vector Laplace problem could be realized as follows:

```
A = assemble.assemble('inner(grad(u), grad(v)) * dx', kvs,
        bfuns=[('u',2), ('v',2)], geo=geo)
```

Here `grad()` computes the Jacobian matrices of the vector-valued basis functions *u* and *v*, and `inner()` computes their Frobenius product.

In addition to the *Supported functions* described below, the following keywords can be used in the string description:

`dx`	volume measure
`ds`	surface measure (switches to *Surface integrals*)
`u, v`	default trial and test function (only when not using the *bfuns* argument of `assemble()`)
`x`	current position in physical coordinates (i.e., the value of the geometry map)
`n`	the unit normal vector (only for surface integrals)
`gw`	the Gauss quadrature weight at the current node (usually not needed since it is included in `dx` and `ds`)
`jac`	the Jacobian matrix of the geometry map with shape *geo_dim x dim*

The `assemble()` function also works in hierarchical spline spaces; simply pass an instance of `pyiga.hierarchical.HSpace` as the second argument instead of the tuple of knot vectors *kvs*. The result will be a sparse matrix or a 1D vector, and in each case the degrees of freedom are ordered according to the canonical order defined in the `pyiga.hierarchical` module.

For most problems, this simple string-based interface is sufficient to generate all required matrices and vectors. However, in some edge cases it may be necessary to define a custom vform by hand. The remainder of this document describes the internals of how to create such objects. It also gives a more formal description of how the expressions occurring in the strings above should be interpreted. In essence, these strings are nothing but Python expressions evaluated in a context where the members of the `pyiga.vform` module are made available.

1.2.2 Programmatically defining VForms

The `pyiga.vform` module contains the tools for describing variational forms, and `pyiga.vform.VForm` is the main class used to represent abstract variational forms. Its constructor has one required argument which describes the space dimension. For instance, to initialize a `VForm` for a three-dimensional problem:

```
from pyiga import vform

vf = vform.VForm(3)
```

In order to create expressions for our form, we need objects which represent the functions that our form operates on. By default `VForm` assumes a bilinear form, and therefore we can obtain objects for the trial and the test function using `VForm.basisfuns()` like this:

```
u, v = vf.basisfuns()
```

The objects that we work with when specifying vforms are abstract expressions (namely, instances of `Expr`) and all have certain properties such as a shape, `Expr.shape`, which is a tuple of dimensions just like a numpy array has. By default, `VForm` assumes a scalar-valued problem, and therefore both the trial function `u` and the test function `v` are scalar:

```
>>> u.shape
()
>>> v.shape
()
```

We can now start building expressions using these functions. Let's first import some commonly needed functions from the `vform` module.

```
from pyiga.vform import grad, div, inner, dx
```

We will often need the gradient of a function, obtained via `grad()`:

```
>>> gu = grad(u)
>>> gu.shape
(3,)
```

Note that `grad(u)` is itself an expression. As expected, the gradient of a scalar function is a three-component vector. We could take the divergence (`div()`) of the gradient and get back a scalar expression which represents the Laplacian $\Delta u = \operatorname{div} \nabla u$ of `u`:

```
>>> Lu = div(gu)
>>> Lu.shape
()
```

However, in order to express the standard variational form for the Laplace problem, we only require the inner product $\nabla u \cdot \nabla v$ of the gradients of our input functions, which can be computed using `inner()`:

```
>>> x = inner(grad(u), grad(v))
>>> x.shape
()
```

Again, this is a scalar since `inner()` represents a contraction over all axes of its input tensors; for vectors, it is the scalar product, and for matrices, it is the Frobenius product.

Note: In general, the syntax for constructing forms sticks as closely as possible to that of the UFL language used in

FEniCS, and therefore the UFL documentation is also a helpful resource.

Finally we want to represent the integral of this expression over the computational domain. We do this by multiplying with the symbol `dx`:

```
integral = inner(grad(u), grad(v)) * dx
```

Internally, `dx` is actually a scalar expression which represents the absolute value of the determinant of the geometry Jacobian, i.e., the scalar term that stems from transforming the integrand from the physical domain to the parameter domain.

We are now ready to add this expression to our `VForm` via `VForm.add()`, and since the expression is rather simple, we can skip all the intermediate steps and variables and simply do

```
vf.add(inner(grad(u), grad(v)) * dx)
```

Note that the expression passed to `VForm.add()` here is exactly the string we passed to `assemble()` in the first example in *the previous section*.

1.2.3 A simple example

It's usually convenient to define vforms in their own functions so that we don't pollute the global namespace with the definitions from the `vform` module. The Laplace variational form

$$a(u, v) = \int_\Omega \nabla u \cdot \nabla v \, dx$$

would be defined like this:

```
def laplace_vf(dim):
    from pyiga.vform import VForm, grad, inner, dx
    vf = VForm(dim)
    u, v = vf.basisfuns()
    vf.add(inner(grad(u), grad(v)) * dx)
    return vf
```

Calling this function results in a `VForm` object:

```
>>> laplace_vf(2)
<pyiga.vform.VForm at 0x7f0fdcf5c0f0>
```

Note: Currently, the predefined Laplace variational form in `pyiga` is defined in a different way which leads to a slightly higher performance of the generated code.

1.2.4 Vector-valued problems

By default, basis functions are assumed to be scalar-valued. To generate forms with vector-valued functions, simply pass the `components` argument with the desired sizes to `VForm.basisfuns()`:

```
>>> vf = vform.VForm(2)
>>> u, v = vf.basisfuns(components=(2,2))
```

(continues on next page)

(continued from previous page)

```
>>> u.shape, v.shape
((2,), (2,))
```

We can still compute gradients (Jacobians) using `grad()` as before:

```
>>> grad(u).shape
(2, 2)
```

As a simple example, the div-div bilinear form $a(u, v) = \int_\Omega \operatorname{div} u \operatorname{div} v \, dx$ would be implemented using

```
vf.add(div(u) * div(v) * dx)
```

It is also possible to mix vector and scalar functions, e.g. for Stokes-like problems:

```
vf = vform.VForm(2)
u, p = vf.basisfuns(components=(2,1))

vf.add(div(u) * p * dx)
```

In this example, `u` is a vector-valued function and `p` is scalar-valued.

1.2.5 Working with coefficient functions

Often you will need to provide additional functions as input to your assembler, for instance to represent a diffusion coefficient which varies over the computational domain. A scalar input field is declared using the `VForm.input()` method as follows:

```
>>> vf = vform.VForm(2)
>>> coeff = vf.input('coeff')

>>> coeff.shape
()
```

The new variable `coeff` now represents a scalar expression that we can work with just as with the basis functions, e.g.

```
>>> grad(coeff).shape
(2,)
```

As a simple example, to use this as a scalar diffusion coefficient, we would do

```
u, v = vf.basisfuns()
vf.add(coeff * inner(grad(u), grad(v)) * dx)
```

Input fields can be declared vector- or matrix-valued simply by prescribing their shape. In this example, we declare a 2x2 matrix-valued coefficient function:

```
>>> vf = vform.VForm(2)
>>> coeff = vf.input('coeff', shape=(2,2))

>>> coeff.shape
(2, 2)
```

To actually supply these functions to the assembler, they must be passed as keyword arguments to the constructor of the generated assembler class. It is possible to pass either standard Python functions (in which case differentiation is not supported) or instances of *pyiga.bspline.BSplineFunc* or *pyiga.geometry.NurbsFunc*. In fact, the predefined input `geo` for the geometry map is simply declared as a vector-valued input field. See the section *Compiling and assembling* for an example of how to pass these functions.

By default, input functions are considered to be defined in the coordinates of the parameter domain. If your input function is given in terms of physical coordinates, declare it as follows:

```
coeff = vf.input('coeff', physical=True)
```

Note: In the simple string-based interface described in *the first section*, functions passed as *BSplineFunc* or similar objects are assumed to be given in parametric coordinates, whereas standard Python functions are assumed to be given in physical coordinates. This simple heuristic usually produces the expected result.

For performance reasons, it is sometimes beneficial to be able to update a single input function without recreating the entire assembler class, for instance when assembling the same form many times with different coefficients in a Newton iteration for a nonlinear problem. In this case, we can declare the function as follows:

```
func = vf.input('func', updatable=True)
```

The generated assembler class then has an `update()` method which takes the function as a keyword argument and updates it accordingly, e.g.,

```
asm.update(func=F)
```

1.2.6 Defining constant values

If a needed coefficient function is constant, it is unnecessary to use the `VForm.input()` machinery. Instead, we can simply define constant values using the `as_expr()`, `as_vector()`, and `as_matrix()` functions as follows:

```
>>> coeff = vform.as_expr(5)
>>> coeff.shape
()

>>> vcoeff = vform.as_vector([2,3,4])
>>> vcoeff.shape
(3,)

>>> mcoeff = vform.as_matrix([[2,1],[1,2]])
>>> mcoeff.shape
(2, 2)
```

We can then work with these constants exactly as with any other expression.

For constant scalar values as well as tuples of constants or expressions, the coercion to expressions is performed implicitly, making `as_expr()` and `as_vector()` unnecessary in these cases. This means that we can directly write expressions such as

```
vf.add(inner(3 * grad(u), grad(v)) * dx)
vf.add(inner((2.0, 3.0), grad(u)) * dx)
```

The first example also shows that multiplication of a scalar with a vector works as expected, i.e., the vector is multiplied componentwise with the scalar.

The above approach has the disadvantage that the assembler needs to be recompiled every time a constant changes. It is possible to instead define constant parameters which are unknown at compile-time and specified only when assembling. Such constants are defined, analogously to the input fields in the previous section, using the `VForm.parameter()` method:

```
>>> vf = vform.VForm(2)
>>> b = vf.parameter('b', shape=(2,))

>>> b.shape
(2,)
```

Again, scalar, vector-valued and matrix-valued parameters are supported. Before assembling, these parameters must be set using the `update_params()` method of the assembler class, analogously to the `update()` method for input fields described in the previous section.

When using the *string-based interface*, such constants may be defined simply by passing them as keyword arguments into the assemble function:

```
f = assemble.assemble('inner(b, grad(v)) * dx',
                      kvs, geo=geo, b=(2.0, -1.0))
```

1.2.7 Defining linear (unary) forms

By default, `VForm` assumes the case of a bilinear form, i.e., having a trial function `u` and a test function `v`. For defining right-hand sides, we usually need linear forms which have only one argument. We can do this by passing `arity=1` to the `VForm` constructor. The `VForm.basisfuns()` method returns only a single basis function in this case.

Below is a simple example for defining the linear form $\langle F, v\rangle = \int_\Omega f v \, dx$ with a user-specified input function `f`:

```
vf = vform.VForm(3, arity=1)
v = vf.basisfuns()
f = vf.input('f')
vf.add(f * v * dx)
```

1.2.8 Working with parametric derivatives

By default, a `VForm` assumes that you will provide it with a geometry map under the input field name `geo` and automatically transforms the derivatives of the basis functions `u` and `v`, as well as gradients of any input fields, accordingly. If for some reason you need to work with untransformed gradients with respect to parametric coordinates, you can simply pass the keyword argument `parametric=True` to the derivative functions such as `grad()` like this:

```
vf = vform.VForm(2)
u, v = vf.basisfuns()
f = vf.input('f')
gu = grad(u, parametric=True)
gf = Dx(f, 1, parametric=True)
```

You can compute both parametric and physical derivatives of basis functions as well as input fields given in parametric coordinates. An input field that is given in terms of physical coordinates only supports physical derivatives.

Note that the symbol `dx` still includes the geometry Jacobian, and therefore you should not multiply your expression with it if you want to integrate over the parameter domain instead of the physical domain. In this case, you should multiply your expression with the attribute `VForm.GaussWeight` instead, which represents the weight for the Gauss quadrature. When using *the textual description*, use the keyword `gw` for this purpose. When computing integrals over the physical domain, the quadrature weight is automatically subsumed into `dx` and does not need to be specified explicitly.

1.2.9 Surface integrals

By default, a `VForm` will assume that the dimension of the image of the geometry map is the same as the dimension of the spline space over which we are integrating. For computing matrices and vectors over surfaces, we can specify the *geo_dim* argument of the `VForm` constructor to be one higher than the input dimension. Of course, the *geo* function we pass when assembling must match that output dimension. We also have to multiply with the surface measure `ds` instead of the volume measure `dx` when computing such integrals.

Here is a simple example which describes a linear functional over a surface:

```
def L2_surface_functional_vf(dim):
    V = VForm(dim, geo_dim=dim+1, arity=1)
    v = V.basisfuns()
    f = V.input('f')
    V.add(f * v * ds)
    return V
```

When called with *dim=2*, it represents a 2D surface integral in a 3D ambient space.

If you need to use the normal vector in your expression, you can access it via the `VForm.normal` attribute. It is oriented according to the standard right-hand rule and has unit length.

At the moment, transformations of derivatives on surfaces are not implemented, and therefore you can only use *parametric derivatives*.

The string-based interface *described above* will automatically switch to surface integration when it detects that `ds` was used instead of `dx`.

1.2.10 Supported functions

The following functions and expressions are implemented in `pyiga.vform` and have the same semantics as in the UFL language (see the UFL documentation):

`dx` `Dx()` `grad()` `div()` `curl()` `as_vector()` `as_matrix()` `inner()` `dot()` `tr()` `det()` `inv()` `cross()` `outer()` `abs()` `sqrt()` `exp()` `log()` `sin()` `cos()` `tan()`

In addition, all expressions have the members `Expr.dx()` for partial derivatives (analogous to the global function `Dx()`), `Expr.T` for transposing matrices, and `Expr.dot()` which is analogous to the global `dot()` function. Vector and matrix expressions can be indexed and sliced using the standard Python `[]` operator. Expressions also support the standard arithmetic operators `+, -, *, /, **`.

1.2.11 Compiling and assembling

Once a vform has been defined, it has to be compiled into efficient code and then invoked in order to assemble the desired matrix or vector. Currently, there is only one backend in `pyiga` which is based on Cython – the vform is translated into Cython code, compiled on the fly and loaded as a dynamic module. The compiled modules are cached so that compiling a given vform for a second time does not recompile the code.

Below is an example for assembling the Laplace variational form defined in the section *A simple example*:

```
from pyiga import assemble, bspline, geometry

# define the trial space
kv = bspline.make_knots(3, 0.0, 1.0, 20)
kvs = (kv, kv)    # 2D tensor product spline space

```

(continues on next page)

(continued from previous page)

```
# define the geometry map
geo = geometry.quarter_annulus()    # NURBS quarter annulus

A = assemble.assemble_vf(stiffness_vf(2), kvs, geo=geo, symmetric=True)
```

Any further input functions the assembler requires can be passed as further keyword arguments in the call to *assemble_vf()*. The function will automatically detect whether the VForm has arity 1 or 2 and generate a vector or a matrix correspondingly.

Manual compilation of the variational form

Sometimes it may be necessary to directly work with the assembler class that results from compiling a `VForm`. The functions used for compilation are contained in the `pyiga.compile` module, and the resulting matrices can be computed using the *pyiga.assemble.assemble_entries()* functions.

Using these functions, the Laplace variational form defined above can be assembled as follows:

```
from pyiga import compile, assemble, bspline, geometry

# compile the vform into an assembler class
Asm = compile.compile_vform(laplace_vf(2))

# define the trial space
kv = bspline.make_knots(3, 0.0, 1.0, 20)
kvs = (kv, kv)    # 2D tensor product spline space

# define the geometry map
geo = geometry.quarter_annulus()    # NURBS quarter annulus

A = assemble.assemble_entries(Asm(kvs, geo=geo), symmetric=True)
```

The geometry map is passed using `geo=` to the constructor of the assembler class, and further input functions defined as described in the section *Working with coefficient functions* can be passed in the same way using their given name as the keyword.

The resulting object `A` is a sparse matrix in CSR format; different matrix formats can be chosen by passing the `format=` keyword argument to *assemble_entries()*. The argument `symmetric=True` takes advantage of the symmetry of the bilinear form in order to speed up the assembly.

CHAPTER 2

API Reference

2.1 B-splines

Functions and classes for B-spline basis functions.

class `pyiga.bspline.`**`BSplineFunc`**(*kvs*, *coeffs*)

Any function that is given in terms of a tensor product B-spline basis with coefficients.

Parameters

- **`kvs`** (*`seq`*) – tuple of *d* *`KnotVector`*.
- **`coeffs`** (*`ndarray`*) – coefficient array

kvs represents a tensor product B-spline basis, where the *i*-th *`KnotVector`* describes the B-spline basis in the *i*-th coordinate direction.

coeffs is the array of coefficients with respect to this tensor product basis. The length of its first *d* axes must match the number of degrees of freedom in the corresponding *`KnotVector`*. Trailing axes, if any, determine the output dimension of the function. If there are no trailing dimensions or only a single one of length 1, the function is scalar-valued.

For convenience, if *coeffs* is a vector, it is reshaped to the proper size for the tensor product basis. The result is a scalar-valued function.

`kvs`

the knot vectors representing the tensor product basis

Type seq

`coeffs`

the coefficients for the function or geometry

Type ndarray

`sdim`

dimension of the parameter domain

Type int

dim
dimension of the output of the function

Type int

apply_matrix(*A*)
Apply a matrix to each control point of this function and return the result.

A should either be a single matrix or an array of matrices, one for each control point. Standard numpy broadcasting rules apply.

as_nurbs()
Return a NURBS version of this function with constant weights equal to 1.

as_vector()
Convert a scalar function to a 1D vector function.

boundary(*bdspec*)
Return one side of the boundary as a `BSplineFunc`.

Parameters **bdspec** – the side of the boundary to return; see `compute_dirichlet_bc()`

Returns `BSplineFunc` – representation of the boundary side; has `sdim` reduced by 1 and the same `dim` as this function

bounding_box(*grid=1*)
Compute a bounding box for the image of this geometry.

By default, only the corners are taken into account. By choosing *grid > 1*, a finer grid can be used (for non-convex geometries).

Returns a tuple of *(lower,upper)* limits per dimension (in XY order)

copy()
Return a copy of this geometry.

cylinderize(*z0=0.0*, *z1=1.0*, *support=(0.0, 1.0)*)
Create a patch with one additional space dimension by linearly extruding along the new axis from *z0* to *z1*.

By default, the new knot vector will be defined over the interval (0, 1). A different interval can be specified through the *support* parameter.

eval(**x*)
Evaluate the function at a single point of the parameter domain.

Parameters ***x** – the point at which to evaluate the function, in xyz order

find_inverse(*x*, *tol=1e-08*)
Find the coordinates in the parameter domain which correspond to the physical point *x*.

grid_eval(*gridaxes*)
Evaluate the function on a tensor product grid.

Parameters **gridaxes** (*seq*) – list of 1D vectors describing the tensor product grid.

Note: The gridaxes should be given in reverse order, i.e., the x axis last.

Returns *ndarray* – array of function values; shape corresponds to input grid.

grid_hessian(*gridaxes*)
Evaluate the Hessian matrix of a scalar or vector function on a tensor product grid.

Parameters **gridaxes** (*seq*) – list of 1D vectors describing the tensor product grid.

Note: The gridaxes should be given in reverse order, i.e., the x axis last.

Returns

ndarray – array of the components of the Hessian; symmetric part only, linearized. I.e., in 2D, it contains per grid point a 3-component vector corresponding to the derivatives *(d_xx, d_xy, d_yy)*, and in 3D, a 6-component vector with the derivatives *(d_xx, d_xy, d_xz, d_yy, d_yz, d_zz)*. If the input function is vector-valued, one such Hessian vector is computed per component of the function.

Thus, the output is an array of shape *grid_shape x num_comp x num_hess*, where *grid_shape* is the shape of the tensor grid described by the *gridaxes*, *num_comp* is the number of components of the function, and *num_hess* is the number of entries in the symmetric part of the Hessian as described above. The axis corresponding to *num_comp* is elided if the input function is scalar.

grid_jacobian(*gridaxes*)
Evaluate the Jacobian on a tensor product grid.

Parameters **gridaxes** (*seq*) – list of 1D vectors describing the tensor product grid.

Note: The gridaxes should be given in reverse order, i.e., the x axis last.

Returns *ndarray* – array of Jacobians (`dim` × `sdim`); shape corresponds to input grid. For scalar functions, the output is a vector of length `sdim` (the gradient) per grid point.

is_scalar()
Returns True if the function is scalar-valued.

is_vector()
Returns True if the function is vector-valued.

perturb(*noise*)
Create a copy of this function where all coefficients are randomly perturbed by noise of the given magnitude.

pointwise_eval(*points*)
Evaluate the B-spline function at an unstructured list of points.

Parameters **points** – an array or sequence such that *points[i]* is an array containing the coordinates for dimension *i*, where *i = 0, …, sdim - 1* (in xyz order). All arrays must have the same shape.

Returns An *ndarray* containing the function values at the *points*.

pointwise_jacobian(*points*)
Evaluate the Jacobian of the B-spline function at an unstructured list of points.

Parameters **points** – an array or sequence such that *points[i]* is an array containing the coordinates for dimension *i*, where *i = 0, …, sdim - 1* (in xyz order). All arrays must have the same shape.

Returns An *ndarray* containing the Jacobian matrices at the *points*, i.e., a matrix of size *dim x sdim* per evaluation point.

`rotate_2d`(*angle*)
Rotate a geometry with `dim` = 2 by the given angle and return the result.

`scale`(*factor*)
Scale all control points either by a scalar factor or componentwise by a vector and return the resulting new function.

`support`
Return a sequence of pairs *(lower,upper)*, one per source dimension, which describe the extent of the support in the parameter space.

`transformed_jacobian`(*geo*)
Create a function which evaluates the physical (transformed) gradient of the current function after a geometry transform.

`translate`(*offset*)
Return a version of this geometry translated by the specified offset.

class `pyiga.bspline.`**`KnotVector`**(*knots*, *p*)
Represents an open B-spline knot vector together with a spline degree.

Parameters

- **`knots`** (*`ndarray`*) – the 1D knot vector. Should be an open knot vector, i.e., the first and last knot should be repeated *p+1* times. Interior knots may be single or repeated up to *p* times.
- **`p`** (*`int`*) – the spline degree.

This class is commonly used to represent the B-spline basis over the given knot vector with the given spline degree. The B-splines are normalized in the sense that they satisfy a partition of unity property.

Tensor product B-spline bases are typically represented simply as tuples of the univariate knot spans. E.g., `(kv1, kv2)` would represent the tensor product basis formed from the two B-spline bases over the *`KnotVector`* instances `kv1` and `kv2`.

A more convenient way to create knot vectors is the *`make_knots()`* function.

`kv`
vector of knots

Type ndarray

`p`
spline degree

Type int

`copy`()
Return a copy of this knot vector.

`findspan`(*u*)
Returns an index i such that kv[i] <= u < kv[i+1] (except for the boundary, where u <= kv[m-p] is allowed) and p <= i < len(kv) - 1 - p

`first_active`(*k*)
Index of first active basis function in interval (kv[k], kv[k+1])

`first_active_at`(*u*)
Index of first active basis function in the interval which contains *u*.

`greville`()
Compute Gréville abscissae for this knot vector

mesh
Return the mesh, i.e., the vector of unique knots in the knot vector.

mesh_span_indices()
Return an array of indices i such that kv[i] != kv[i+1], i.e., the indices of the nonempty spans. Return value has length self.numspans.

mesh_support_idx(*j*)
Return the first and last mesh index of the support of i

mesh_support_idx_all()
Compute an integer array of size $N \times 2$, where N = self.numdofs, which contains for each B-spline the result of `mesh_support_idx()`.

meshsize_avg()
Compute average length of the knot spans of this knot vector

numdofs
Number of basis functions in a B-spline basis defined over this knot vector

numspans
Number of nontrivial intervals in the knot vector

refine(*new_knots=None*)
Return the refinement of this knot vector by inserting *new_knots*, or performing uniform refinement if none are given.

support(*j=None*)
Support of the knot vector or, if *j* is passed, of the j-th B-spline

support_idx(*j*)
Knot indices of support of j-th B-spline

class pyiga.bspline.**PhysicalGradientFunc**(*func*, *geo*)
A class for function objects which evaluate physical (transformed) gradients of scalar functions with geometry transforms.

boundary(*bdspec*)
Return one side of the boundary as a `UserFunction`.

Parameters **bdspec** – the side of the boundary to return; see `compute_dirichlet_bc()`

Returns `UserFunction` – representation of the boundary side; has *sdim* reduced by 1 and the same *dim* as this function

bounding_box(*grid=1*)
Compute a bounding box for the image of this geometry.

By default, only the corners are taken into account. By choosing *grid > 1*, a finer grid can be used (for non-convex geometries).

Returns a tuple of *(lower,upper)* limits per dimension (in XY order)

find_inverse(*x*, *tol=1e-08*)
Find the coordinates in the parameter domain which correspond to the physical point *x*.

is_scalar()
Returns True if the function is scalar-valued.

is_vector()
Returns True if the function is vector-valued.

pyiga.bspline.**active_ev**(*knotvec*, *u*)
Evaluate all active B-spline basis functions at the points *u*.

Returns an array of shape (p+1, u.size) if *u* is an array.

`pyiga.bspline.`**`collocation`**(*kv*, *nodes*)

Compute collocation matrix for B-spline basis at the given interpolation nodes.

Parameters

- **`kv`** (`KnotVector`) – the B-spline knot vector
- **`nodes`** (`array`) – array of nodes at which to evaluate the B-splines

Returns A Scipy CSR matrix with shape *(len(nodes), kv.numdofs)* whose entry at *(i,j)* is the value of the *j*-th B-spline evaluated at *nodes[i]*.

`pyiga.bspline.`**`collocation_derivs`**(*kv*, *nodes*, *derivs=1*)

Compute collocation matrix and derivative collocation matrices for B-spline basis at the given interpolation nodes.

Returns a list of derivs+1 sparse CSR matrices with shape (nodes.size, kv.numdofs).

`pyiga.bspline.`**`collocation_derivs_info`**(*kv*, *nodes*, *derivs=1*)

Similar to `collocation_info()`, but the second array also contains coefficients for computing derivatives up to order *derivs*. It has shape *(derivs + 1) x len(nodes) x (p + 1)*.

Corresponds to a row-wise representation of the matrices computed by `collocation_derivs()`.

`pyiga.bspline.`**`collocation_info`**(*kv*, *nodes*)

Return two arrays: one containing the index of the first active B-spline per evaluation node, and one containing, per node, the coefficient vector of length *p+1* for the linear combination of basis functions which yields the point evaluation at that node.

Corresponds to a row-wise representation of the collocation matrix (see `collocation()`).

`pyiga.bspline.`**`deriv`**(*knotvec*, *coeffs*, *deriv*, *u*)

Evaluate a derivative of the spline with given B-spline coefficients at all points *u*.

Parameters

- **`knotvec`** (`KnotVector`) – B-spline knot vector
- **`coeffs`** (*ndarray*) – 1D array of coefficients, length *knotvec.numdofs*
- **`deriv`** (`int`) – which derivative to evaluate
- **`u`** (*ndarray*) – 1D array of evaluation points

Returns *ndarray* – the vector of function derivatives

`pyiga.bspline.`**`ev`**(*knotvec*, *coeffs*, *u*)

Evaluate a spline with given B-spline coefficients at all points *u*.

Parameters

- **`knotvec`** (`KnotVector`) – B-spline knot vector
- **`coeffs`** (*ndarray*) – 1D array of coefficients, length *knotvec.numdofs*
- **`u`** (*ndarray*) – 1D array of evaluation points

Returns *ndarray* – the vector of function values

`pyiga.bspline.`**`interpolate`**(*kv*, *func*, *nodes=None*)

Interpolate function in B-spline basis at given nodes (or Gréville abscissae by default)

`pyiga.bspline.`**`knot_insertion`**(*kv*, *u*)

Return a sparse matrix of size *(n+1) x n*, with *n = kv.numdofs´, which maps coefficients from ‘kv* to a new knot vector obtained by inserting the new knot *u* into *kv*.

`pyiga.bspline.`**`load_vector`**(*kv*, *f*)

Compute the load vector (L_2 inner products of basis functions with *f*).

`pyiga.bspline.`**`make_knots`**(*p*, *a*, *b*, *n*, *mult=1*)

Create an open knot vector of degree *p* over an interval *(a,b)* with *n* knot spans.

This automatically repeats the first and last knots *p+1* times in order to create an open knot vector. Interior knots are single by default, i.e., have maximum continuity.

Parameters

- **`p`** (*`int`*) – the spline degree
- **`a`** (*`float`*) – the starting point of the interval
- **`b`** (*`float`*) – the end point of the interval
- **`n`** (*`int`*) – the number of knot spans to divide the interval into
- **`mult`** (*`int`*) – the multiplicity of interior knots

Returns *`KnotVector`* – the new knot vector

`pyiga.bspline.`**`numdofs`**(*kvs*)

Convenience function which returns the number of dofs in a single knot vector or in a tensor product space represented by a tuple of knot vectors.

`pyiga.bspline.`**`project_L2`**(*kv*, *f*)

Compute the B-spline coefficients for the L_2-projection of *f*.

`pyiga.bspline.`**`prolongation`**(*kv1*, *kv2*)

Compute prolongation matrix between B-spline bases.

Given two B-spline bases, where the first spans a subspace of the second one, compute the matrix which maps spline coefficients from the first basis to the coefficients of the same function in the second basis.

Parameters

- **`kv1`** (*`KnotVector`*) – source B-spline basis knot vector
- **`kv2`** (*`KnotVector`*) – target B-spline basis knot vector

Returns *csr_matrix* – sparse matrix which prolongs coefficients from *kv1* to *kv2*

`pyiga.bspline.`**`single_ev`**(*knotvec*, *i*, *u*)

Evaluate i-th B-spline at all points *u*

`pyiga.bspline.`**`tp_bsp_eval_pointwise`**(*kvs*, *coeffs*, *points*)

Evaluate a tensor-product B-spline function at an unstructured list of points.

Parameters

- **`kvs`** – tuple of *`KnotVector`* instances representing a tensor-product B-spline basis
- **`coeffs`** (*`ndarray`*) – coefficient array; see *`BSplineFunc`* for details
- **`points`** – an array or sequence such that *points[i]* is an array containing the coordinates for dimension *i*, where *i = 0, …, len(kvs) - 1* (in xyz order). All arrays must have the same shape.

Returns An *ndarray* containing the function values of the spline function at the *points*.

`pyiga.bspline.`**`tp_bsp_eval_with_jac_pointwise`**(*kvs*, *coeffs*, *points*)

Evaluate the values and Jacobians of a tensor-product B-spline function at an unstructured list of points.

Parameters

- **kvs** – tuple of `KnotVector` instances representing a tensor-product B-spline basis
- **coeffs** (*ndarray*) – coefficient array; see `BSplineFunc` for details
- **points** – an array or sequence such that *points[i]* is an array containing the coordinates for dimension *i*, where *i = 0, . . . , len(kvs) - 1* (in xyz order). All arrays must have the same shape.

Returns A pair of ‘ndarray‘s – one for the values and one for the Jacobians.

`pyiga.bspline.`**`tp_bsp_jac_pointwise`**(*kvs*, *coeffs*, *points*)

Evaluate the Jacobian of a tensor-product B-spline function at an unstructured list of points.

Parameters

- **kvs** – tuple of `KnotVector` instances representing a tensor-product B-spline basis
- **coeffs** (*ndarray*) – coefficient array; see `BSplineFunc` for details
- **points** – an array or sequence such that *points[i]* is an array containing the coordinates for dimension *i*, where *i = 0, . . . , len(kvs) - 1* (in xyz order). All arrays must have the same shape.

Returns An *ndarray* containing the Jacobians of the spline function at the *points*.

2.2 Geometry

Classes and functions for creating and manipulating tensor product B-spline and NURBS patches.

See *Geometry manipulation in pyiga* for examples on how to create custom geometries.

class `pyiga.geometry.`**`NurbsFunc`**(*kvs*, *coeffs*, *weights*, *premultiplied=False*)

Any function that is given in terms of a tensor product NURBS basis with coefficients and weights.

Parameters

- **kvs** (*seq*) – tuple of *d* `KnotVector`s.
- **coeffs** (*ndarray*) – coefficient array; see `BSplineFunc` for format. The constructor may modify *coeffs* during premultiplication!
- **weights** (*ndarray*) – coefficients for weight function in the same format as *coeffs*. If *weights=None* is passed, the weights are assumed to be given as the last vector component of *coeffs* instead.
- **premultiplied** (*bool*) – pass *True* if the coefficients are already premultiplied by the weights.

kvs

the knot vectors representing the tensor product basis

Type seq

coeffs

the premultiplied coefficients for the function, including the weights in the last vector component

Type ndarray

sdim

dimension of the parameter domain

Type int

dim
dimension of the output of the function

Type int

The evaluation functions have the same prototypes and behavior as those in `BSplineFunc`.

apply_matrix(*A*)
Apply a matrix to each control point of this function, leave the weights unchanged, and return the result.

A should either be a single matrix or an array of matrices, one for each control point. Standard numpy broadcasting rules apply.

boundary(*bdspec*)
Return one side of the boundary as a `NurbsFunc`.

Parameters **bdspec** – the side of the boundary to return; see `compute_dirichlet_bc()`

Returns `NurbsFunc` – representation of the boundary side; has *sdim* reduced by 1 and the same *dim* as this function

bounding_box(*grid=1*)
Compute a bounding box for the image of this geometry.

By default, only the corners are taken into account. By choosing *grid > 1*, a finer grid can be used (for non-convex geometries).

Returns a tuple of *(lower,upper)* limits per dimension (in XY order)

coeffs_weights()
Return the non-premultiplied coefficients and weights as a pair of arrays.

copy()
Return a copy of this geometry.

eval(**x*)
Evaluate the function at a single point of the parameter domain.

Parameters ***x** – the point at which to evaluate the function, in xyz order

find_inverse(*x*, *tol=1e-08*)
Find the coordinates in the parameter domain which correspond to the physical point *x*.

is_scalar()
Returns True if the function is scalar-valued.

is_vector()
Returns True if the function is vector-valued.

pointwise_eval(*points*)
Evaluate the NURBS function at an unstructured list of points.

Parameters **points** – an array or sequence such that *points[i]* is an array containing the coordinates for dimension *i*, where *i = 0, . . . , sdim - 1* (in xyz order). All arrays must have the same shape.

Returns An *ndarray* containing the function values at the *points*.

pointwise_jacobian(*points*)
Evaluate the Jacobian of the NURBS function at an unstructured list of points.

Parameters **points** – an array or sequence such that *points[i]* is an array containing the coordinates for dimension *i*, where *i = 0, . . . , sdim - 1* (in xyz order). All arrays must have the same shape.

Returns An *ndarray* containing the Jacobian matrices at the *points*, i.e., a matrix of size *dim x sdim* per evaluation point.

rotate_2d(*angle*)
Rotate a geometry with `dim` = 2 by the given angle and return the result.

scale(*factor*)
Scale all control points either by a scalar factor or componentwise by a vector, leave the weights unchanged, and return the resulting new function.

support
Return a sequence of pairs *(lower,upper)*, one per source dimension, which describe the extent of the support in the parameter space.

translate(*offset*)
Return a version of this geometry translated by the specified offset.

class `pyiga.geometry.`**UserFunction**(*f*, *support*, *dim=None*, *jac=None*)
A function (supporting the same basic protocol as `BSplineFunc`) which is given in terms of a user-defined callable.

Parameters

- **f** (`callable`) – a function of *d* variables; may be scalar or vector-valued
- **support** – a sequence of *d* pairs of the form *(lower,upper)* describing the support of the function (see `BSplineFunc.support`)
- **dim** (`int`) – the dimension of the function output; by default, is automatically determined by calling *f*
- **jac** (`callable`) – optionally, a function evaluating the Jacobian matrix of the function

The `sdim` attribute (see `BSplineFunc.sdim`) is determined from the length of *support*.

boundary(*bdspec*)
Return one side of the boundary as a `UserFunction`.

Parameters **bdspec** – the side of the boundary to return; see `compute_dirichlet_bc()`

Returns `UserFunction` – representation of the boundary side; has *sdim* reduced by 1 and the same *dim* as this function

bounding_box(*grid=1*)
Compute a bounding box for the image of this geometry.

By default, only the corners are taken into account. By choosing *grid > 1*, a finer grid can be used (for non-convex geometries).

Returns a tuple of *(lower,upper)* limits per dimension (in XY order)

find_inverse(*x*, *tol=1e-08*)
Find the coordinates in the parameter domain which correspond to the physical point *x*.

is_scalar()
Returns True if the function is scalar-valued.

is_vector()
Returns True if the function is vector-valued.

`pyiga.geometry.`**bspline_quarter_annulus**(*r1=1.0*, *r2=2.0*)
A B-spline approximation of a quarter annulus in the first quadrant.

Parameters

- **r1** (*float*) – inner radius
- **r2** (*float*) – outer radius

Returns *BSplineFunc* 2D geometry

`pyiga.geometry.`**`circle`**(*r=1.0*)
Construct a circle with radius *r* using NURBS.

`pyiga.geometry.`**`circular_arc`**(*alpha*, *r=1.0*)
Construct a circular arc with angle *alpha* and radius *r*.

The arc is centered at the origin, starts on the positive *x* axis and travels in counterclockwise direction.

`pyiga.geometry.`**`circular_arc_3pt`**(*alpha*, *r=1.0*)
Construct a circular arc with angle *alpha* and radius *r* using 3 control points.

The angle *alpha* must be between 0 and *pi*.

`pyiga.geometry.`**`circular_arc_5pt`**(*alpha*, *r=1.0*)
Construct a circular arc with angle *alpha* and radius *r* using 5 control points.

`pyiga.geometry.`**`circular_arc_7pt`**(*alpha*, *r=1.0*)
Construct a circular arc with angle *alpha* and radius *r* using 7 control points.

`pyiga.geometry.`**`disk`**(*r=1.0*)
A NURBS representation of a circular disk.

The parametrization has four boundary singularities where the determinant of the Jacobian becomes 0, at the bottom, top, left and right edges.

Parameters **r** (*float*) – radius

Returns *NurbsFunc* 2D geometry

`pyiga.geometry.`**`identity`**(*extents*)
Identity mapping (using linear splines) over a d-dimensional box given by *extents* as a list of (min,max) pairs or of *KnotVector*.

Returns *BSplineFunc* geometry

`pyiga.geometry.`**`line_segment`**(*x0*, *x1*, *support=(0.0, 1.0)*, *intervals=1*)
Return a *BSplineFunc* which describes the line between the vectors *x0* and *x1*.

If specified, *support* describes the interval in which the function is supported; by default, it is the interval (0,1).

If specified, *intervals* is the number of intervals in the underlying linear spline space. By default, the minimal spline space with 2 dofs is used.

`pyiga.geometry.`**`outer_product`**(*G1*, *G2*)
Compute the outer product of two *BSplineFunc* or *NurbsFunc* geometries. This means that given two input functions

$$G_1(y), G_2(x),$$

it returns a new function

$$G(x, y) = G_1(y)G_2(x),$$

where the multiplication is componentwise in the case of vector functions. The returned function is a *NurbsFunc* if either input is and a *BSplineFunc* otherwise. It has source dimension (*BSplineFunc.sdim*) equal to the sum of the source dimensions of the input functions.

G1 and *G2* should have the same image dimension (`BSplineFunc.dim`), and the output then has the same as well. However, broadcasting according to standard Numpy rules is permissible; e.g., one function can be vector-valued and the other scalar-valued.

The coefficients of the result are the pointwise products of the coefficients of the input functions over a new tensor product spline space.

`pyiga.geometry.`**`outer_sum`**`(`*G1*, *G2*`)`

Compute the outer sum of two `BSplineFunc` or `NurbsFunc` geometries. This means that given two input functions

$$G_1(y), G_2(x),$$

it returns a new function

$$G(x, y) = G_1(y) + G_2(x).$$

The returned function is a `NurbsFunc` if either input is and a `BSplineFunc` otherwise. It has source dimension (`BSplineFunc.sdim`) equal to the sum of the source dimensions of the input functions.

G1 and *G2* should have the same image dimension (`BSplineFunc.dim`), and the output then has the same as well. However, broadcasting according to standard Numpy rules is permissible; e.g., one function can be vector-valued and the other scalar-valued.

The coefficients of the result are the pointwise sums of the coefficients of the input functions over a new tensor product spline space.

`pyiga.geometry.`**`perturbed_square`**`(`*num_intervals=5*, *noise=0.02*`)`

Randomly perturbed unit square.

Unit square with given number of intervals per direction; the control points are perturbed randomly according to the given noise level.

Returns `BSplineFunc` 2D geometry

`pyiga.geometry.`**`quarter_annulus`**`(`*r1=1.0*, *r2=2.0*`)`

A NURBS representation of a quarter annulus in the first quadrant. The 'bottom' and 'top' boundaries (with respect to the reference domain) lie on the x and y axis, respectively.

Parameters

- **r1** (*float*) – inner radius
- **r2** (*float*) – outer radius

Returns `NurbsFunc` 2D geometry

`pyiga.geometry.`**`semicircle`**`(`*r=1.0*`)`

Construct a semicircle in the upper half-plane with radius *r* using NURBS.

`pyiga.geometry.`**`tensor_product`**`(`*G1*, *G2*, **Gs*`)`

Compute the tensor product of two or more `BSplineFunc` or `NurbsFunc` functions. This means that given two input functions

$$G_1(y), G_2(x),$$

it returns a new function

$$G(x, y) = G_2(x) \times G_1(y),$$

where $\times$ means that vectors are joined together. The resulting `BSplineFunc` or `NurbsFunc` has source dimension (`BSplineFunc.sdim`) equal to the sum of the source dimensions of the input functions, and target dimension (`BSplineFunc.dim`) equal to the sum of the target dimensions of the input functions.

`pyiga.geometry.`**`twisted_box`**`()`

A 3D volume that resembles a box with its right face twisted and bent upwards.

Corresponds to gismo data file twistedFlatQuarterAnnulus.xml.

Returns *`BSplineFunc`* 3D geometry

`pyiga.geometry.`**`unit_cube`**`(`*dim=3*, *num_intervals=1*`)`

The *dim*-dimensional unit cube with *num_intervals* intervals per coordinate direction.

Returns *`BSplineFunc`* geometry

`pyiga.geometry.`**`unit_square`**`(`*num_intervals=1*`)`

Unit square with given number of intervals per direction.

Returns *`BSplineFunc`* 2D geometry

2.3 Approximation

2.4 Assembling

Assembling functions for B-spline IgA.

This module contains functions to assemble stiffness matrices and load vectors for isogeometric discretizations. Furthermore, it provides support for computing and eliminating Dirichlet dofs from a linear system.

2.4.1 Assembling general problems

General variational forms can be assembled using the following function. See the section *Assembling custom forms* for further details.

`pyiga.assemble.`**`assemble`**`(`*problem*, *kvs*, *args=None*, *bfuns=None*, *boundary=None*, *symmetric=False*, *format='csr'*, *layout='blocked'*, ***kwargs*`)`

Assemble a matrix or vector in a function space.

Parameters

- **`problem`** – the description of the variational form to assemble. It can be passed in a number of formats (see *Assembling custom forms* for details):
  - string: a textual description of the variational form
  - `VForm`: an abstract description of the variational form
  - assembler class (result of compiling a `VForm` using `pyiga.compile.compile_vform()`)
  - assembler object (result of instantiating an assembler class with a concrete space and input functions) – in this case *kvs* and *args* are ignored
- **`kvs`** – the space or spaces in which to assemble the problem. Either:
  - a tuple of *`KnotVector`* instances, describing a tensor product spline space
  - if the variational form requires more than one space (e.g., for a Petrov-Galerkin discretization), then a tuple of such tuples, each describing one tensor product spline space (usually one for the trial space and one for the test space)
  - an `HSpace` instance for problems in hierarchical spline spaces

- **args** (*dict*) – a dictionary which provides named inputs for the assembler. Most problems will require at least a geometry map; this can be given in the form `{'geo': geo}`, where `geo` is a geometry function defined using the `pyiga.geometry` module. Further values used in the *problem* description must be passed here.

  For convenience, any additional keyword arguments to this function are added to the *args* dict automatically.

- **bfuns** – a list of used basis functions. By default, scalar basis functions 'u' and 'v' are assumed, and the arity of the variational form is determined automatically based on whether one or both of these functions are used. Otherwise, *bfuns* should be a list of tuples *(name, components, space)*, where *name* is a string, *components* is an integer describing the number of components the basis function has, and *space* is an integer referring to which input space the function lives in. Shorter tuples are valid, in which case the components default to *components=1* (scalar basis function) and *space=0* (all functions living in the same, first, input space).

  This argument is only used if *problem* is given as a string.

For the meaning of the remaining arguments and the format of the output, refer to `assemble_entries()`.

`pyiga.assemble.`**`assemble_vf`**(*vf*, *kvs*, *symmetric=False*, *format='csr'*, *layout='blocked'*, *args=None*, ***kwargs*)

Compile the given variational form (`VForm`) into a matrix or vector.

Any named inputs defined in the vform must be given in the *args* dict or as keyword arguments. For the meaning of the remaining arguments, refer to `assemble_entries()`.

`pyiga.assemble.`**`assemble_entries`**(*asm*, *symmetric=False*, *format='csr'*, *layout='blocked'*)

Given an instance *asm* of an assembler class, assemble all entries and return the resulting matrix or vector.

Parameters

- **asm** – an instance of an assembler class, e.g. one compiled using `pyiga.compile.compile_vform()`
- **symmetric** (*bool*) – (matrices only) exploit symmetry of the matrix to speed up the assembly
- **format** (*str*) – (matrices only) the sparse matrix format to use; default 'csr'
- **layout** (*str*) – (vector-valued problems only): the layout of the generated matrix or vector. Valid options are:
  - 'blocked': the matrix is laid out as a *k_1 x k_2* block matrix, where *k_1* and *k_2* are the number of components of the test and trial functions, respectively
  - 'packed': the interactions of the components are packed together, i.e., each entry of the matrix is a small *k_1 x k_2* block

Returns

ndarray or sparse matrix –

- if the assembler has arity=1: an ndarray of vector entries whose shape is given by the number of degrees of freedom per coordinate direction. For vector-valued problem, an additional final axis is added which has the number of components as its length.
- if the assembler has arity=2: a sparse matrix in the given *format*

class `pyiga.assemble.`**`Assembler`**(*problem*, *kvs*, *args=None*, *bfuns=None*, *boundary=None*, *symmetric=False*, *updatable=[]*, ***kwargs*)

A high-level interface to an assembler class.

Usually, you will simply call `assemble()` to assemble a matrix or vector. When assembling a sequence of problems with changing parameters, it may be more efficient to instantiate the assembler class only once and inform it of the changing inputs via the *updatable* argument. You may then call `update()` to change these input fields and assemble the problem via `assemble()`. Instead of calling `update()` explicitly, you can also simply pass the fields to be updated as additional keyword arguments to `assemble()`.

updatable is a list of names of assembler arguments which are to be considered updatable. The other arguments have the same meaning as for `assemble()`.

`assemble`(*format='csr'*, *layout='blocked'*, ***upd_fields*)

Assemble the problem.

Any additional keyword arguments are passed to `update()`.

`update`(***kwargs*)

Update all input fields given as *name=func* keyword arguments.

All fields updated in this manner must have been specified in the list of *updatable* arguments when creating the `Assembler` object.

2.4.2 Tensor product Gauss quadrature assemblers

Standard Gauss quadrature assemblers for mass and stiffness matrices. They take one or two arguments:

- *kvs* (list of `KnotVector`): Describes the tensor product B-spline basis for which to assemble the matrix. One `KnotVector` per coordinate direction. In the 1D case, a single `KnotVector` may be passed directly.
- *geo* (`BSplineFunc` or `NurbsFunc`; optional): Geometry transform, mapping from the parameter domain to the physical domain. If omitted, assume the identity map; a fast Kronecker product implementation is used in this case.

`pyiga.assemble.`**`mass`**(*kvs*, *geo=None*, *format='csr'*)

Assemble a mass matrix for the given basis (B-spline basis or tensor product B-spline basis) with an optional geometry transform.

`pyiga.assemble.`**`stiffness`**(*kvs*, *geo=None*, *format='csr'*)

Assemble a stiffness matrix for the given basis (B-spline basis or tensor product B-spline basis) with an optional geometry transform.

2.4.3 Fast low-rank assemblers

Fast low-rank assemblers based on the paper “A Black-Box Algorithm for Fast Matrix Assembly in Isogeometric Analysis”. They may achieve significant speedups over the classical Gauss assemblers, in particular for fine discretizations and higher spline degrees. They only work well if the geometry transform is rather smooth so that the resulting matrix has relatively low (numerical) Kronecker rank.

They take the following additional arguments:

- *tol*: the stopping accuracy for the Adaptive Cross Approximation (ACA) algorithm
- *maxiter*: the maximum number of ACA iterations
- *skipcount*: terminate after finding this many successive near-zero pivots
- *tolcount*: terminate after finding this many successive pivots below the desired accuracy
- *verbose*: the amount of output to display. *0* is silent, *1* prints basic information, *2* prints detailed information

`pyiga.assemble.`**`mass_fast`**(*kvs*, *geo=None*, *tol=1e-10*, *maxiter=100*, *skipcount=3*, *tolcount=3*, *verbose=2*)

Assemble a mass matrix for the given tensor product B-spline basis with an optional geometry transform, using the fast low-rank assembling algorithm.

`pyiga.assemble.`**`stiffness_fast`**(*kvs*, *geo=None*, *tol=1e-10*, *maxiter=100*, *skipcount=3*, *tolcount=3*, *verbose=2*)

Assemble a stiffness matrix for the given tensor product B-spline basis with an optional geometry transform, using the fast low-rank assembling algorithm.

2.4.4 Right-hand sides

`pyiga.assemble.`**`inner_products`**(*kvs*, *f*, *f_physical=False*, *geo=None*)

Compute the L_2 inner products between each basis function in a tensor product B-spline basis and the function *f* (i.e., the load vector).

Parameters

- **`kvs`** (*`seq`*) – a sequence of *`KnotVector`*, representing a tensor product basis
- **`f`** – a function or *`BSplineFunc`* object
- **`f_physical`** (*`bool`*) – whether *f* is given in physical coordinates. If *True*, *geo* must be passed as well.
- **`geo`** – a *`BSplineFunc`* or *`NurbsFunc`* which describes the integration domain; if not given, the integrals are computed in the parameter domain

Returns *ndarray* – the inner products as an array of size *kvs[0].ndofs* × *kvs[1].ndofs* × … × *kvs[-1].ndofs*. Each entry corresponds to the inner product of the corresponding basis function with *f*. If *f* is not scalar, then each of its components is treated separately and the corresponding dimensions are appended to the end of the return value.

2.4.5 Boundary and initial conditions

`pyiga.assemble.`**`compute_dirichlet_bcs`**(*kvs*, *geo*, *bdconds*)

Compute indices and values for Dirichlet boundary conditions on several boundaries at once.

Parameters

- **`kvs`** – a tensor product B-spline basis
- **`geo`** (*`BSplineFunc`* or *`NurbsFunc`*) – the geometry transform
- **`bdconds`** – a list of *(bdspec, dir_func)* pairs, where *bdspec* specifies the boundary to apply a Dirichlet boundary condition to and *dir_func* is the function providing the Dirichlet values. For the exact meaning, refer to *`compute_dirichlet_bc()`*. As a shorthand, it is possible to pass a single pair `("all", dir_func)` which applies Dirichlet boundary conditions to all boundaries.

Returns A pair *(indices, values)* suitable for passing to *`RestrictedLinearSystem`*.

`pyiga.assemble.`**`compute_dirichlet_bc`**(*kvs*, *geo*, *bdspec*, *dir_func*)

Compute indices and values for a Dirichlet boundary condition using interpolation.

Parameters

- **`kvs`** – a tensor product B-spline basis
- **`geo`** (*`BSplineFunc`* or *`NurbsFunc`*) – the geometry transform

- **bdspec** – a pair *(axis, side)*. *axis* denotes the axis along which the boundary condition lies, and *side* is either 0 for the "lower" boundary or 1 for the "upper" boundary. Alternatively, one of the following six strings can be used for *bdspec*:

value	Meaning
`"left"`	x low
`"right"`	x high
`"bottom"`	y low
`"top"`	y high
`"front"`	z low
`"back"`	z high

- **dir_func** – a function which will be interpolated to obtain the Dirichlet boundary values. Assumed to be given in physical coordinates. If it is vector-valued, one Dirichlet dof is computed per component, and they are numbered according to the "blocked" matrix layout. If *dir_func* is a scalar value, a constant function with that value is assumed.

Returns A pair of arrays *(indices, values)* which denote the indices of the dofs within the tensor product basis which lie along the Dirichlet boundary and their computed values, respectively.

`pyiga.assemble.`**`compute_initial_condition_01`**(*kvs*, *geo*, *bdspec*, *g0*, *g1*, *physical=True*)

Compute indices and values for an initial condition including function value and derivative for a space-time discretization using interpolation. This only works for a space-time cylinder with constant (in time) geometry. To be precise, the space-time geometry transform *geo* should have the form

$$G(\vec{x}, t) = (\widetilde{G}(\vec{x}), t).$$

Parameters

- **kvs** – a tensor product B-spline basis
- **geo** (`BSplineFunc` or `NurbsFunc`) – the geometry transform of the space-time cylinder
- **bdspec** – a pair *(axis, side)*. *axis* denotes the time axis of *geo*, and *side* is either 0 for the "lower" boundary or 1 for the "upper" boundary.
- **g0** – a function which will be interpolated to obtain the initial function values
- **g1** – a function which will be interpolated to obtain the initial derivatives.
- **physical** (`bool`) – whether the functions *g0* and *g1* are given in physical (True) or parametric (False) coordinates. Physical coordinates are assumed by default.

Returns A pair of arrays *(indices, values)* which denote the indices of the dofs within the tensor product basis which lie along the initial face of the space-time cylinder and their computed values, respectively.

`pyiga.assemble.`**`combine_bcs`**(*bcs*)

Given a sequence of *(indices, values)* pairs such as returned by `compute_dirichlet_bc()`, combine them into a single pair *(indices, values)*.

Dofs which occur in more than one *indices* array take their value from an arbitrary corresponding *values* array.

class `pyiga.assemble.`**`RestrictedLinearSystem`**(*A*, *b*, *bcs*, *elim_rows=None*)

Represents a linear system with some of its dofs eliminated.

Parameters

- **A** – the full matrix
- **b** – the right-hand side (may be 0)

- **bcs** – a pair of arrays *(indices, values)* which contain the indices and values, respectively, of dofs to be eliminated from the system
- **elim_rows** – for Petrov-Galerkin discretizations, the equations to be eliminated from the linear system may not match the dofs to be eliminated. In this case, an array of indices of rows to be eliminated may be passed in this argument.

Once constructed, the restricted linear system can be accessed through the following attributes:

A
the restricted matrix

b
the restricted and updated right-hand side

complete(*u*)
Given a solution *u* of the restricted linear system, complete it with the values of the eliminated dofs to a solution of the original system.

extend(*u*)
Given a vector *u* containing only the free dofs, pad it with zeros to all dofs.

restrict(*u*)
Given a vector *u* containing all dofs, return its restriction to the free dofs.

restrict_matrix(*B*)
Given a matrix *B* which operates on all dofs, return its restriction to the free dofs.

restrict_rhs(*f*)
Given a right-hand side vector *f*, return its restriction to the non-eliminated rows.

If *elim_rows* was not passed, this is equivalent to `restrict()`.

2.4.6 Multipatch problems

class pyiga.assemble.**Multipatch**(*patches*, *automatch=False*)
Represents a multipatch structure, consisting of a number of patches together with their discretizations and the information about shared dofs between patches. Currently, only conforming patches are supported.

Parameters

- **patches** – a list of *(kvs, geo)* pairs, each of which describes a patch. Here *kvs* is a tuple of `KnotVector` instances describing the tensor product spline space used for discretization and *geo* is the geometry map.
- **automatch** (*bool*) – if True, attempt to automatically determine the matching interfaces between all patches and finalize then. If False, the user has to manually join the patches by calling `join_boundaries()` as often as needed, followed by `finalize()`.

assemble_system(*problem*, *rhs*, *args=None*, *bfuns=None*, *symmetric=False*, *format='csr'*, *layout='blocked'*, ***kwargs*)
Assemble both the system matrix and the right-hand side vector for a variational problem over the multipatch geometry.

Here *problem* represents a bilinear form and *rhs* a linear functional. See `assemble()` for the precise meaning of the arguments.

Returns A pair *(A, b)* consisting of the sparse system matrix and the right-hand side vector.

compute_dirichlet_bcs(*bdconds*)
Performs the same operation as the global function `compute_dirichlet_bcs()`, but for a multipatch problem.

The sequence *bdconds* should contain triples of the form *(patch, bdspec, dir_func)*.

Returns A pair *(indices, values)* suitable for passing to `RestrictedLinearSystem`.

`finalize`()

After all shared dofs have been declared using `join_boundaries()` or `join_dofs()`, call this function to set up the internal data structures.

`global_to_patch`(*p*)

Compute a sparse binary matrix which maps global dofs to local dofs in patch *p*. This is just the transpose of `patch_to_global()` and also its left-inverse.

Parameters **p** (*int*) – the index of the patch

Returns a sparse matrix

`join_boundaries`(*p1*, *bdspec1*, *p2*, *bdspec2*, *flip=None*)

Join the dofs lying along boundary *bdspec1* of patch *p1* with those lying along boundary *bdspec2* of patch *p2*.

See `compute_dirichlet_bc()` for the format of the boundary specification.

If *flip* is given, it should be a sequence of booleans indicating for each coordinate axis of the boundary if the coordinates of *p2* have to be flipped along that axis.

`join_dofs`(*p1*, *I1*, *p2*, *I2*)

Join the dofs *I1* of patch *p1* with the dofs *I2* of patch *p2*.

`numdofs`

Number of dofs after eliminating shared dofs.

May only be called after `finalize()`.

`numpatches`

Number of patches in the multipatch structure.

`patch_to_global`(*p*, *j_global=False*)

Compute a sparse binary matrix which maps dofs local to patch *p* to the corresponding global dofs.

Parameters

- **p** (*int*) – the index of the patch
- **j_global** (*bool*) – if False, the matrix has only as many columns as *p* has dofs; if True, the number of columns is the sum of the number of local dofs over all patches

Returns a CSR sparse matrix

`patch_to_global_idx`(*p*)

Return an array which maps local tensor product indices for patch *p* to global indices.

2.4.7 Integration

`pyiga.assemble.`**`integrate`**(*kvs*, *f*, *f_physical=False*, *geo=None*)

Compute the integral of the function *f* over the geometry *geo* or a simple tensor product domain.

Parameters

- **kvs** (*seq*) – a sequence of `KnotVector`; determines the parameter domain and the quadrature rule
- **f** – a function or `BSplineFunc` object

- **f_physical** (*bool*) – whether *f* is given in physical coordinates. If *True*, *geo* must be passed as well.
- **geo** – a *BSplineFunc* or *NurbsFunc* which describes the integration domain; if not given, the integral is computed in the parameter domain

Returns *float* – the integral of *f* over the specified domain

2.5 Hierarchical Spline Spaces

2.5.1 Hierarchical spline spaces

2.5.2 Discretization

2.6 Operators

Classes and functions for creating custom instances of `scipy.sparse.linalg.LinearOperator`.

class pyiga.operators.**BaseBlockOperator**(*shape*, *ops*, *ran_out*, *ran_in*)

pyiga.operators.**BlockDiagonalOperator**(**ops*)
Return a `LinearOperator` with block diagonal structure, with the given operators on the diagonal.

pyiga.operators.**BlockOperator**(*ops*)
Construct a block operator.

Parameters **ops** (*list*) – a rectangular list of lists of operators or matrices. All operators in a given row should have the same height (output dimension). All operators in a given column should have the same width (input dimension). Empty blocks should use *NullOperator* as a placeholder.

Returns *LinearOperator* – a block structured linear operator. Its height is the total height of one input column of operators, and its width is the total width of one input row.

See also `numpy.block()`, which has analogous functionality for dense matrices.

class pyiga.operators.**DiagonalOperator**(*diag*)
A `LinearOperator` which acts like a diagonal matrix with the given diagonal.

class pyiga.operators.**IdentityOperator**(*n*, *dtype=<class 'numpy.float64'>*)
Identity operator of size *n*.

class pyiga.operators.**KroneckerOperator**(**ops*)
A `LinearOperator` which efficiently implements the application of the Kronecker product of the given input operators.

class pyiga.operators.**NullOperator**(*shape*, *dtype=<class 'numpy.float64'>*)
Null operator of the given shape which always returns zeros. Used as placeholder.

class pyiga.operators.**PardisoSolverWrapper**(*shape*, *dtype*, *solver*)
Wraps a PARDISO solver object and frees up the memory when deallocated.

class pyiga.operators.**SubspaceOperator**(*subspaces*, *Bs*)
Implements an abstract additive subspace correction operator.

Parameters

- **subspaces** (*seq*) – a list of *k* prolongation matrices $P_j \in \mathbb{R}^{n \times n_j}$

- **Bs** (*seq*) – a list of k square matrices or instances of `LinearOperator` $B_j \in \mathbb{R}^{n_j \times n_j}$

Returns

LinearOperator – operator with shape $n \times n$ that implements the action

$$Lx = \sum_{j=1}^{k} P_j B_j P_j^T x$$

`pyiga.operators.`**`make_kronecker_solver`**(**Bs*)

Given several square matrices, return an operator which efficiently applies the inverse of their Kronecker product.

`pyiga.operators.`**`make_solver`**(*B*, *symmetric=False*, *spd=False*)

Return a `LinearOperator` that acts as a linear solver for the (dense or sparse) square matrix *B*.

If *B* is symmetric, passing `symmetric=True` may try to take advantage of this. If *B* is symmetric and positive definite, pass `spd=True`.

2.7 Tensors

Functions and classes for manipulating tensors in full, canonical, and Tucker format, and for tensor approximation.

A **full tensor** is simply represented as a `numpy.ndarray`. Additional tensor formats are implemented in the following classes:

- *`CanonicalTensor`*
- *`TuckerTensor`*

In addition, arbitrary tensors can be composed into sums or tensor products using the following classes:

- *`TensorSum`*
- *`TensorProd`*

Below, whenever we refer generically to "a tensor", we mean either an *ndarray* or an instance of any of these tensor classes.

All tensor classes have members `ndim`, `shape`, and `ravel` which have the same meaning as for an *ndarray*. Any tensor can be expanded to a full *ndarray* using *`asarray()`*. In addition, most tensor classes have overloaded operators for adding and subtracting tensors in their native format.

All tensors can be sliced using the standard numpy `[]` indexing syntax. The result is a tensor in the same format, except for the case where all axes have a single scalar index, in which case the entry at the corresponding index is returned as a scalar value.

Linear operators on tensors, themselves represented in suitable low-rank formats, are described by

- *`CanonicalOperator`*

2.7.1 Module members

`class` `pyiga.tensor.`**`CanonicalOperator`**(*terms*)

Represents a linear operator on tensors which is described as a sum of rank one operators (Kronecker products), i.e.,

$$\mathcal{A} = \sum_{r=1}^{R} A_r^1 \otimes \cdots \otimes A_r^d.$$

The argument *terms* is a list of length *R* of *d*-tuples containing the matrices A_r^k.

R
: Kronecker rank of the operator

  Type int

shape
: a pair where *shape[1]* is the shape of input tensors accepted by this operator and *shape[0]* is the shape of output tensors produced

  Type tuple

ndim
: the number of dimensions, i.e., *d* in the formula above

  Type int

T
: Return the transpose of this operator as a `CanonicalOperator`.

apply(*X*)
: Return the result of applying this operator to a tensor *X*.

asmatrix(*format='csr'*)
: Return the raveled form of this operator as a sparse matrix in the given format.

static eye(*ns*, *format='dia'*)
: Represent the identity as a tensor product of identity matrices with sizes given by the tuple of integers *ns*.

kron(*other*)
: Construct a new `CanonicalOperator` as the Kronecker product of this and *other*.

class pyiga.tensor.**CanonicalTensor**(*Xs*)
: A tensor in CP (canonical/PARAFAC) format, i.e., a sum of rank 1 tensors.

  For a tensor of order *d*, *Xs* should be a tuple of *d* matrices. Their number of columns should be identical and determines the rank *R* of the tensor. The number of rows of the *j*-th matrix determines the size of the tensor along the *j*-th axis.

  The tensor is given by the sum, for *r* up to *R*, of the outer products of the *r*-th columns of the matrices *Xs*.

  asarray()
  : Convert canonical tensor to a full *ndarray*.

  copy()
  : Create a deep copy of this tensor.

  static from_tensor(*A*)
  : Convert *A* from other tensor formats to canonical format.

  static from_terms(*terms*)
  : Construct a canonical tensor from a list of rank 1 terms, represented as tuples of vectors.

  norm()
  : Compute the Frobenius norm of the tensor.

  nway_prod(*Bs*)
  : Implements `apply_tprod()` for canonical tensors.

    Returns `CanonicalTensor` – the result in canonical format

  static ones(*shape*)
  : Construct a constant canonical tensor with all entries one and the given shape.

ravel()
Return the vectorization of this tensor.

squeeze(*axis=None*)
Eliminate singleton axes. Equivalent to `numpy.squeeze()`.

terms()
Return the rank one components as a list of tuples.

static zeros(*shape*)
Construct a zero canonical tensor with the given shape.

class pyiga.tensor.**TensorProd**(**Xs*)
Represents the abstract tensor product of an arbitrary number of tensors.

asarray()
Convert sum of tensors to a full *ndarray*.

nway_prod(*Bs*)
Implements *apply_tprod()* for tensor products.

Returns *TensorProd* – the result as a tensor product

ravel()
Return the vectorization of this tensor.

class pyiga.tensor.**TensorSum**(**Xs*)
Represents the abstract sum of an arbitrary number of tensors with identical shapes.

asarray()
Convert sum of tensors to a full *ndarray*.

nway_prod(*Bs*)
Implements *apply_tprod()* for sums of tensors.

Returns *TensorSum* – the result as a sum of tensors

ravel()
Return the vectorization of this tensor.

class pyiga.tensor.**TuckerTensor**(*Us*, *X*)
A *d*-dimensional tensor in **Tucker format** is given as a list of *d* basis matrices

$$U_k \in \mathbb{R}^{n_k \times m_k}, \qquad k = 1, \ldots, d$$

and a (typically small) core coefficient tensor

$$X \in \mathbb{R}^{m_1 \times \ldots \times m_d}.$$

When expanded (using *TuckerTensor.asarray()*), a Tucker tensor turns into a full tensor

$$A \in \mathbb{R}^{n_1 \times \ldots \times n_d}.$$

One way to compute a Tucker tensor approximation from a full tensor is to first compute the HOSVD using *hosvd()* and then truncate it using *TuckerTensor.truncate()* to the rank estimated by *find_truncation_rank()*. Such a rank compression is implemented in *TuckerTensor.compress()*.

asarray()
Convert Tucker tensor to a full *ndarray*.

compress(*tol=1e-15*, *rtol=1e-15*)
Approximate this Tucker tensor by another one of smaller rank, up to an absolute error tolerance *tol* or a relative error tolerance *rtol*.

Returns the approximation as a `TuckerTensor`

copy()
Create a deep copy of this tensor.

static from_tensor(*A*)
Convert *A* from other tensor formats to Tucker format.

norm()
Compute the Frobenius norm of the tensor.

nway_prod(*Bs*)
Implements `apply_tprod()` for Tucker tensors.

Returns `TuckerTensor` – the result in Tucker format

static ones(*shape*)
Construct a constant Tucker tensor with all entries one and the given shape.

orthogonalize()
Compute an equivalent Tucker representation of the current tensor where the matrices *U* have orthonormal columns.

Returns `TuckerTensor` – the orthonormalized Tucker tensor

ravel()
Return the vectorization of this tensor.

squeeze(*axis=None*)
Eliminate singleton axes. Equivalent to `numpy.squeeze()`.

truncate(*k*)
Truncate a Tucker tensor *T* to the given rank *k*.

static zeros(*shape*)
Construct a zero Tucker tensor with the given shape.

pyiga.tensor.**als**(*A*, *R*, *tol=1e-10*, *maxiter=10000*, *startval=None*)
Compute best rank *R* approximation to tensor *A* using Alternating Least Squares.

Parameters

- **A** (*tensor*) – the tensor to be approximated
- **R** (*int*) – the desired rank
- **tol** (*float*) – tolerance for the stopping criterion
- **maxiter** (*int*) – maximum number of iterations
- **startval** – starting tensor for iteration. By default, a random rank *R* tensor is used. A `CanonicalTensor` with rank *R* may be supplied for *startval* instead.

Returns `CanonicalTensor` – a rank *R* approximation to *A*; generally close to the best rank *R* approximation if the algorithm converged to a small enough tolerance.

pyiga.tensor.**als1**(*A*, *tol=1e-15*)
Compute best rank 1 approximation to tensor *A* using Alternating Least Squares.

Parameters

- **A** (*tensor*) – the tensor to be approximated
- **tol** (*float*) – tolerance for the stopping criterion

Returns A tuple of vectors *(x1, . . . , xd)* such that `outer(x1, ..., xd)` is the approximate best rank 1 approximation to *A*.

`pyiga.tensor.`**`als1_ls`**(*A*, *B*, *tol=1e-15*, *maxiter=10000*, *spd=False*)
Compute rank 1 approximation to the solution of a linear system by Alternating Least Squares.

`pyiga.tensor.`**`als1_ls_structured`**(*A*, *B*, *tol=1e-15*, *maxiter=10000*)
Compute rank 1 approximation to the solution of a linear system by Alternating Least Squares.

Faster version of `als1_ls()`, but works only if all the matrices in the operator *A* have identical sparsity structure.

`pyiga.tensor.`**`apply_tprod`**(*ops*, *A*)
Apply multi-way tensor product of operators to tensor *A*.

Parameters

- **`ops`** (*`seq`*) – a list of matrices, sparse matrices, or LinearOperators
- **`A`** (*`tensor`*) – the tensor to apply the multi-way tensor product to

Returns a new tensor with the same number of axes as *A* that is the result of applying the tensor product operator `ops[0] x ... x ops[-1]` to *A*. The return type is typically the same type as *A*.

The initial dimensions of *A* must match the sizes of the operators, but *A* is allowed to have an arbitrary number of trailing dimensions. `None` is a valid operator and is treated like the identity.

An interpretation of this operation is that the Kronecker product of the matrices *ops* is applied to the vectorization of the tensor *A*.

`pyiga.tensor.`**`array_outer`**(**xs*)
Outer product of an arbitrary number of ndarrays.

Parameters **`xs`** – an arbitrary number of input ndarrays

Returns *ndarray* – the outer product of the inputs. Its shape is the concatenation of the shapes of the inputs.

`pyiga.tensor.`**`asarray`**(*X*)
Return the tensor *X* as a full ndarray.

`pyiga.tensor.`**`find_truncation_rank`**(*X*, *tol=1e-12*)
A greedy algorithm for finding a good truncation rank for a HOSVD core tensor.

`pyiga.tensor.`**`fro_norm`**(*X*)
Compute the Frobenius norm of the tensor *X*.

`pyiga.tensor.`**`grou`**(*B*, *R*, *tol=1e-12*, *return_errors=False*)
Canonical tensor approximation by Greedy Rank One Updates.

References

https://doi.org/10.1016/j.cam.2019.03.002

Parameters

- **`B`** (*`tensor`*) – the tensor to be approximated
- **`R`** (*`int`*) – the desired canonical rank for the approximation
- **`tol`** (*`double`*) – the desired absolute error tolerance
- **`return_errors`** (*`bool`*) – whether to return the error history as a second return value

Returns The computed approximation as a *CanonicalTensor*. If *return_errors* is true, instead returns a tuple containing the tensor and a list of the error history over the iterations.

`pyiga.tensor.`**`gta`**(*A*, *R*, *tol=1e-12*, *rtol=1e-12*, *return_errors=False*)

Greedy Tucker approximation of the tensor *A*.

References

https://doi.org/10.1016/j.cam.2019.03.002

Parameters

- **A** (*tensor*) – the tensor to be approximated
- **R** (*int*) – the desired multilinear rank of the approximation
- **tol** (*double*) – target absolute error tolerance
- **rtol** (*double*) – target relative error tolerance
- **return_errors** (*bool*) – whether to return the error history as a second return value

Returns The computed approximation as a *TuckerTensor*. If *return_errors* is true, instead returns a tuple containing the tensor and a list of the error history over the iterations.

`pyiga.tensor.`**`gta_ls`**(*A*, *F*, *R*, *tol=1e-12*, *verbose=0*, *gs=None*, *spd=False*)

Greedy Tucker approximation of the solution of a linear system $A U = F$.

References

https://doi.org/10.1016/j.cam.2019.03.002

Parameters

- **A** (*list*) – the linear operator in low Kronecker rank format given as a list of tuples. Each tuple represents a Kronecker product operator and contains *d* matrices or linear operators; the operator is considered as the Kronecker product of these operators
- **F** (*tensor*) – the right-hand side of the linear system as a (possibly low-rank) tensor
- **R** (*int*) – the desired multilinear rank of the approximation (number of iterations)
- **tol** (*double*) – desired reduction of the initial residual
- **verbose** (*int*) – 0 = no printed output, 1 = moderate detail, 2 = full detail
- **gs** (*int*) – if this is not None, then this many Gauss-Seidel iterations are used on the core linear system instead of direct solution; see the paper for details
- **spd** (*bool*) – pass True if *A* is a symmetric positive definite operator; uses a more efficient and accurate rank 1 approximation algorithm (see the corresponding parameter of *als1_ls()*)

Returns the computed approximation as a *TuckerTensor*

`pyiga.tensor.`**`hosvd`**(*X*)

Compute higher-order SVD (Tucker decomposition).

Parameters **X** (*ndarray*) – a full tensor of arbitrary size

Returns *TuckerTensor* – a Tucker tensor which represents *X* with the core tensor having the same shape as *X* and the factor matrices *Uk* being square and orthogonal.

`pyiga.tensor.`**`join_tucker_bases`**(*T1*, *T2*)

Represent the two Tucker tensors *T1* and *T2* in a joint basis.

Returns *tuple* – *(U,X1,X2)* such that `T1 == TuckerTensor(U,X1)` and `T2 == TuckerTensor(U,X2)`. The basis *U* is the concatenation of the bases of *T1* and *T2*.

`pyiga.tensor.`**`matricize`**(*X*, *k*)

Return the mode-*k* matricization of the ndarray *X*.

`pyiga.tensor.`**`modek_tprod`**(*B*, *k*, *X*)

Compute the mode-*k* tensor product of the ndarray *X* with the matrix or operator *B*.

Parameters

- **`B`** – an *ndarray*, sparse matrix, or *LinearOperator* of size *m x nk*
- **`k`** (*`int`*) – the mode along which to multiply *X*
- **`X`** (*`ndarray`*) – tensor with `X.shape[k] == nk`

Returns *ndarray* – the mode-*k* tensor product of size *(n1, … nk-1, m, nk+1, …, nN)*

`pyiga.tensor.`**`outer`**(**xs*)

Outer product of an arbitrary number of vectors.

Parameters **`xs`** – *d* input vectors *(x1, …, xd)* with lengths *n1, …, nd*

Returns *ndarray* – the outer product as an *ndarray* with *d* dimensions

`pyiga.tensor.`**`pad`**(*X*, *pad_width*)

Pad a tensor with zero rows in each direction.

Parameters **`pad_width`** (*`list`*) – a list of *(before,after)* tuples, the same length as dimensions of *X*, which specifices how many zeros to prepend/append in each direction. *None* is admissible and is equivalent to *(0,0)*.

Returns the padded tensor

2.8 Solvers

Solvers for linear, nonlinear, and time-dependent problems.

`pyiga.solvers.`**`GaussSeidelSmoother`**(*iterations=1*, *sweep='forward'*)

Gauss-Seidel smoother.

By default, *iterations* is 1. The direction to be used is specified by *sweep* and may be either 'forward', 'backward', or 'symmetric'.

`exception` `pyiga.solvers.`**`NoConvergenceError`**(*method*, *num_iter*, *last_iterate*)

`pyiga.solvers.`**`OperatorSmoother`**(*S*)

A smoother which applies an arbitrary operator *S* to the residual and uses the result as an update, i.e.,

$$u \leftarrow u + S(f - Au).$$

`pyiga.solvers.`**`SequentialSmoother`**(*smoothers*)

Smoother which applies several smoothers in sequence.

`pyiga.solvers.`**`crank_nicolson`**(*M*, *F*, *J*, *x*, *tau*, *t_end*, ***, *t0=0.0*, *progress=False*)

Solve a time-dependent problem using the Crank-Nicolson method.

Parameters

- **`M`** (*`matrix`*) – the mass matrix
- **`F`** (*`function`*) – the right-hand side
- **`J`** (*`function`*) – function computing the Jacobian of *F*
- **`x`** (*`vector`*) – the initial value
- **`tau0`** (*`float`*) – the initial time step
- **`t_end`** (*`float`*) – the final time up to which to integrate
- **`t0`** (*`float`*) – the initial time; 0 by default
- **`progress`** (*`bool`*) – whether to show a progress bar

Returns A tuple *(times, solutions)*, where *times* is a list of increasing times in the interval *(t0, t_end)*, and *solutions* is a list of vectors which contains the computed solutions at these times.

`pyiga.solvers.`**`dirk34`**(*M*, *F*, *J*, *x*, *tau0*, *t_end*, *tol*, ***, *t0=0.0*, *step_factor=0.9*, *progress=False*)

Solve a time-dependent problem using the DIRK34 Runge-Kutta method.

Parameters

- **`M`** (*`matrix`*) – the mass matrix
- **`F`** (*`function`*) – the right-hand side
- **`J`** (*`function`*) – function computing the Jacobian of *F*
- **`x`** (*`vector`*) – the initial value
- **`tau0`** (*`float`*) – the initial time step
- **`t_end`** (*`float`*) – the final time up to which to integrate
- **`tol`** (*`float`*) – error tolerance for choosing the adaptive time step; if *None*, use constant time steps
- **`t0`** (*`float`*) – the initial time; 0 by default
- **`step_factor`** (*`float`*) – the safety factor for choosing the step size
- **`progress`** (*`bool`*) – whether to show a progress bar

Returns A tuple *(times, solutions)*, where *times* is a list of increasing times in the interval *(t0, t_end)*, and *solutions* is a list of vectors which contains the computed solutions at these times.

`pyiga.solvers.`**`esdirk23`**(*M*, *F*, *J*, *x*, *tau0*, *t_end*, *tol*, ***, *t0=0.0*, *step_factor=0.9*, *progress=False*)

Solve a time-dependent problem using the ESDIRK23 Runge-Kutta method.

Parameters

- **`M`** (*`matrix`*) – the mass matrix
- **`F`** (*`function`*) – the right-hand side
- **`J`** (*`function`*) – function computing the Jacobian of *F*
- **`x`** (*`vector`*) – the initial value
- **`tau0`** (*`float`*) – the initial time step
- **`t_end`** (*`float`*) – the final time up to which to integrate
- **`tol`** (*`float`*) – error tolerance for choosing the adaptive time step; if *None*, use constant time steps

- **t0** (*float*) – the initial time; 0 by default
- **step_factor** (*float*) – the safety factor for choosing the step size
- **progress** (*bool*) – whether to show a progress bar

Returns A tuple *(times, solutions)*, where *times* is a list of increasing times in the interval *(t0, t_end)*, and *solutions* is a list of vectors which contains the computed solutions at these times.

pyiga.solvers.**esdirk34**(*M*, *F*, *J*, *x*, *tau0*, *t_end*, *tol*, ***, *t0=0.0*, *step_factor=0.9*, *progress=False*)
Solve a time-dependent problem using the ESDIRK34 Runge-Kutta method.

Parameters

- **M** (*matrix*) – the mass matrix
- **F** (*function*) – the right-hand side
- **J** (*function*) – function computing the Jacobian of *F*
- **x** (*vector*) – the initial value
- **tau0** (*float*) – the initial time step
- **t_end** (*float*) – the final time up to which to integrate
- **tol** (*float*) – error tolerance for choosing the adaptive time step; if *None*, use constant time steps
- **t0** (*float*) – the initial time; 0 by default
- **step_factor** (*float*) – the safety factor for choosing the step size
- **progress** (*bool*) – whether to show a progress bar

Returns A tuple *(times, solutions)*, where *times* is a list of increasing times in the interval *(t0, t_end)*, and *solutions* is a list of vectors which contains the computed solutions at these times.

pyiga.solvers.**fastdiag_solver**(*KM*)
The fast diagonalization solver as described in [Sangalli, Tani 2016].

Parameters **KM** – a sequence of length *d* (dimension of the problem) containing pairs of symmetric matrices *(K_i, M_i)*

Returns A *LinearOperator* which realizes the inverse of the generalized Laplacian matrix described by the input matrices.

pyiga.solvers.**gauss_seidel**(*A*, *x*, *b*, *iterations=1*, *indices=None*, *sweep='forward'*)
Perform Gauss-Seidel relaxation on the linear system *Ax=b*, updating *x* in place.

Parameters

- **A** – the matrix; either sparse or dense, but should be in CSR format if sparse
- **x** – the current guess for the solution
- **b** – the right-hand side
- **iterations** – the number of iterations to perform
- **indices** – if given, relaxation is only performed on this list of indices
- **sweep** – the direction; either 'forward', 'backward', or 'symmetric'

pyiga.solvers.**iterative_solve**(*step*, *A*, *f*, *x0=None*, *active_dofs=None*, *tol=1e-08*, *maxiter=5000*)
Solve the linear system Ax=f using a basic iterative method.

Parameters

- **step** (*callable*) – a function which performs the update x_old -> x_new for the iterative method
- **A** – matrix or linear operator describing the linear system of equations
- **f** (*ndarray*) – the right-hand side
- **x0** – the starting vector; 0 is used if not specified
- **active_dofs** (*list or ndarray*) – list of active dofs on which the residual is computed. Useful for eliminating Dirichlet dofs without changing the matrix. If not specified, all dofs are active.
- **tol** (*float*) – the desired reduction in the Euclidean norm of the residual relative to the starting residual
- **maxiter** (*int*) – the maximum number of iterations

Returns a pair *(x, iterations)* containing the solution and the number of iterations performed. If *maxiter* was reached without convergence, the returned number of iterations is infinite.

`pyiga.solvers.`**`newton`**(*F*, *J*, *x0*, *atol=1e-06*, *rtol=1e-06*, *maxiter=100*, *freeze_jac=1*)

Solve the nonlinear problem F(x) == 0 using Newton iteration.

Parameters

- **F** (*function*) – function computing the residual of the nonlinear equation
- **J** (*function*) – function computing the Jacobian matrix of *F*
- **x0** (*ndarray*) – the initial guess as a vector
- **atol** (*float*) – absolute tolerance for the norm of the residual
- **rtol** (*float*) – relative tolerance with respect to the initial residual
- **maxiter** (*int*) – the maximum number of iterations
- **freeze_jac** (*int*) – if >1, the Jacobian is only updated every *freeze_jac* steps

Returns *ndarray* – a vector *x* which approximately satisfies F(x) == 0

`pyiga.solvers.`**`rodasp`**(*M*, *F*, *J*, *x*, *tau0*, *t_end*, *tol*, ***, *t0=0.0*, *step_factor=0.9*, *progress=False*)

Solve a time-dependent problem using the RODASP Rosenbrock method.

Parameters

- **M** (*matrix*) – the mass matrix
- **F** (*function*) – the right-hand side
- **J** (*function*) – function computing the Jacobian of *F*
- **x** (*vector*) – the initial value
- **tau0** (*float*) – the initial time step
- **t_end** (*float*) – the final time up to which to integrate
- **tol** (*float*) – error tolerance for choosing the adaptive time step; if *None*, use constant time steps
- **t0** (*float*) – the initial time; 0 by default
- **step_factor** (*float*) – the safety factor for choosing the step size
- **progress** (*bool*) – whether to show a progress bar

Returns A tuple *(times, solutions)*, where *times* is a list of increasing times in the interval *(t0, t_end)*, and *solutions* is a list of vectors which contains the computed solutions at these times.

`pyiga.solvers.`**`ros3p`**(*M*, *F*, *J*, *x*, *tau0*, *t_end*, *tol*, ***, *t0=0.0*, *step_factor=0.9*, *progress=False*)

Solve a time-dependent problem using the ROS3P Rosenbrock method.

Parameters

- **M** (*matrix*) – the mass matrix
- **F** (*function*) – the right-hand side
- **J** (*function*) – function computing the Jacobian of *F*
- **x** (*vector*) – the initial value
- **tau0** (*float*) – the initial time step
- **t_end** (*float*) – the final time up to which to integrate
- **tol** (*float*) – error tolerance for choosing the adaptive time step; if *None*, use constant time steps
- **t0** (*float*) – the initial time; 0 by default
- **step_factor** (*float*) – the safety factor for choosing the step size
- **progress** (*bool*) – whether to show a progress bar

Returns A tuple *(times, solutions)*, where *times* is a list of increasing times in the interval *(t0, t_end)*, and *solutions* is a list of vectors which contains the computed solutions at these times.

`pyiga.solvers.`**`ros3pw`**(*M*, *F*, *J*, *x*, *tau0*, *t_end*, *tol*, ***, *t0=0.0*, *step_factor=0.9*, *progress=False*)

Solve a time-dependent problem using the ROS3PW Rosenbrock method.

Parameters

- **M** (*matrix*) – the mass matrix
- **F** (*function*) – the right-hand side
- **J** (*function*) – function computing the Jacobian of *F*
- **x** (*vector*) – the initial value
- **tau0** (*float*) – the initial time step
- **t_end** (*float*) – the final time up to which to integrate
- **tol** (*float*) – error tolerance for choosing the adaptive time step; if *None*, use constant time steps
- **t0** (*float*) – the initial time; 0 by default
- **step_factor** (*float*) – the safety factor for choosing the step size
- **progress** (*bool*) – whether to show a progress bar

Returns A tuple *(times, solutions)*, where *times* is a list of increasing times in the interval *(t0, t_end)*, and *solutions* is a list of vectors which contains the computed solutions at these times.

`pyiga.solvers.`**`rosi2p1`**(*M*, *F*, *J*, *x*, *tau0*, *t_end*, *tol*, ***, *t0=0.0*, *step_factor=0.9*, *progress=False*)

Solve a time-dependent problem using the ROSI2P1 Rosenbrock method.

Parameters

- **M** (*matrix*) – the mass matrix
- **F** (*function*) – the right-hand side

- **J** (*function*) – function computing the Jacobian of *F*
- **x** (*vector*) – the initial value
- **tau0** (*float*) – the initial time step
- **t_end** (*float*) – the final time up to which to integrate
- **tol** (*float*) – error tolerance for choosing the adaptive time step; if *None*, use constant time steps
- **t0** (*float*) – the initial time; 0 by default
- **step_factor** (*float*) – the safety factor for choosing the step size
- **progress** (*bool*) – whether to show a progress bar

Returns A tuple *(times, solutions)*, where *times* is a list of increasing times in the interval *(t0, t_end)*, and *solutions* is a list of vectors which contains the computed solutions at these times.

`pyiga.solvers.`**`rowdaind2`**(*M*, *F*, *J*, *x*, *tau0*, *t_end*, *tol*, ***, *t0=0.0*, *step_factor=0.9*, *progress=False*)

Solve a time-dependent problem using the ROWDAIND2 Rosenbrock method.

Parameters

- **M** (*matrix*) – the mass matrix
- **F** (*function*) – the right-hand side
- **J** (*function*) – function computing the Jacobian of *F*
- **x** (*vector*) – the initial value
- **tau0** (*float*) – the initial time step
- **t_end** (*float*) – the final time up to which to integrate
- **tol** (*float*) – error tolerance for choosing the adaptive time step; if *None*, use constant time steps
- **t0** (*float*) – the initial time; 0 by default
- **step_factor** (*float*) – the safety factor for choosing the step size
- **progress** (*bool*) – whether to show a progress bar

Returns A tuple *(times, solutions)*, where *times* is a list of increasing times in the interval *(t0, t_end)*, and *solutions* is a list of vectors which contains the computed solutions at these times.

`pyiga.solvers.`**`sdirk21`**(*M*, *F*, *J*, *x*, *tau0*, *t_end*, *tol*, ***, *t0=0.0*, *step_factor=0.9*, *progress=False*)

Solve a time-dependent problem using the SDIRK21 (Ellsiepen) Runge-Kutta method.

Parameters

- **M** (*matrix*) – the mass matrix
- **F** (*function*) – the right-hand side
- **J** (*function*) – function computing the Jacobian of *F*
- **x** (*vector*) – the initial value
- **tau0** (*float*) – the initial time step
- **t_end** (*float*) – the final time up to which to integrate
- **tol** (*float*) – error tolerance for choosing the adaptive time step; if *None*, use constant time steps

- **`t0`** (*`float`*) – the initial time; 0 by default
- **`step_factor`** (*`float`*) – the safety factor for choosing the step size
- **`progress`** (*`bool`*) – whether to show a progress bar

Returns A tuple *(times, solutions)*, where *times* is a list of increasing times in the interval *(t0, t_end)*, and *solutions* is a list of vectors which contains the computed solutions at these times.

`pyiga.solvers.`**`sdirk3`**(*M*, *F*, *J*, *x*, *tau*, *t_end*, ***, *t0=0.0*, *progress=False*)

Solve a time-dependent problem using the SDIRK3 Runge-Kutta method.

Parameters

- **`M`** (*`matrix`*) – the mass matrix
- **`F`** (*`function`*) – the right-hand side
- **`J`** (*`function`*) – function computing the Jacobian of *F*
- **`x`** (*`vector`*) – the initial value
- **`tau0`** (*`float`*) – the initial time step
- **`t_end`** (*`float`*) – the final time up to which to integrate
- **`t0`** (*`float`*) – the initial time; 0 by default
- **`progress`** (*`bool`*) – whether to show a progress bar

Returns A tuple *(times, solutions)*, where *times* is a list of increasing times in the interval *(t0, t_end)*, and *solutions* is a list of vectors which contains the computed solutions at these times.

`pyiga.solvers.`**`sdirk3_b`**(*M*, *F*, *J*, *x*, *tau*, *t_end*, ***, *t0=0.0*, *progress=False*)

Solve a time-dependent problem using the SDIRK3 (alternate) Runge-Kutta method.

Parameters

- **`M`** (*`matrix`*) – the mass matrix
- **`F`** (*`function`*) – the right-hand side
- **`J`** (*`function`*) – function computing the Jacobian of *F*
- **`x`** (*`vector`*) – the initial value
- **`tau0`** (*`float`*) – the initial time step
- **`t_end`** (*`float`*) – the final time up to which to integrate
- **`t0`** (*`float`*) – the initial time; 0 by default
- **`progress`** (*`bool`*) – whether to show a progress bar

Returns A tuple *(times, solutions)*, where *times* is a list of increasing times in the interval *(t0, t_end)*, and *solutions* is a list of vectors which contains the computed solutions at these times.

`pyiga.solvers.`**`solve_hmultigrid`**(*hs*, *A*, *f*, *strategy='cell_supp'*, *smoother='gs'*, *smooth_steps=2*, *tol=1e-08*, *maxiter=5000*)

Solve a linear scalar problem in a hierarchical spline space using local multigrid.

Parameters

- **`hs`** – the `HSpace` which describes the HB- or THB-spline space
- **`A`** – the matrix describing the discretization of the problem
- **`f`** – the right-hand side vector
- **`strategy`** (*`string`*) – how to choose the smoothing sets. Valid options are

  - `"new"`: only the new dofs per level
  - `"trunc"`: all dofs which interact via the truncation operator
  - `"cell_supp"`: all dofs whose support intersects that of the new ones (support extension)
  - `"func_supp"`
- **`smoother`** (*`string`*) – the multigrid smoother to use. Valid options are
  - `"gs"`: forward Gauss-Seidel for pre-smoothing, backward Gauss-Seidel for post-smoothing
  - `"forward_gs"`: always use forward Gauss-Seidel
  - `"backward_gs"`: always use backward Gauss-Seidel
  - `"symmetric_gs"`: use complete symmetric Gauss-Seidel sweep for both pre- and post-smoothing
  - `"exact"`: use an exact direct solver as a pre-smoother (no post-smoothing)
- **`smooth_steps`** (*`int`*) – the number of pre- and post-smoothing steps
- **`tol`** (*`float`*) – the desired reduction in the residual
- **`maxiter`** (*`int`*) – the maximum number of iterations

Returns a pair *(x, iterations)* containing the solution and the number of iterations performed. If *maxiter* was reached without convergence, the returned number of iterations is infinite.

`pyiga.solvers.`**`twogrid`**(*A*, *f*, *P*, *smoother*, *u0=None*, *tol=1e-08*, *smooth_steps=2*, *maxiter=1000*)
Generic two-grid method with arbitrary smoother.

Parameters

- **`A`** – stiffness matrix on fine grid
- **`f`** – right-hand side
- **`P`** – prolongation matrix from coarse to fine grid
- **`smoother`** – a function with arguments *(A,u,f)* which applies one smoothing iteration in-place to *u*
- **`u0`** – starting value; 0 if not given
- **`tol`** – desired reduction relative to initial residual
- **`smooth_steps`** – number of smoothing steps
- **`maxiter`** – maximum number of iterations

Returns *ndarray* – the computed solution to the equation $Au = f$

2.9 Visualization

Visualization functions.

`pyiga.vis.`**`animate_field`**(*fields*, *geo*, *vrange=None*, *res=(50, 50)*, *cmap=None*, *interval=50*, *progress=False*)
Animate a sequence of scalar fields over a geometry.

`pyiga.vis.`**`plot_active_cells`**(*hspace*, *values*, *cmap=None*, *edgecolor=None*)
Plot the mesh of active cells with colors chosen according to the given *values*.

`pyiga.vis.`**`plot_curve`**(*geo*, *res=50*, *linewidth=None*, *color='black'*)
Plot a 2D curve.

`pyiga.vis.`**`plot_field`**(*field*, *geo=None*, *res=80*, *physical=False*, ***kwargs*)
Plot a scalar field, optionally over a geometry.

`pyiga.vis.`**`plot_geo`**(*geo*, *grid=10*, *gridx=None*, *gridy=None*, *res=50*, *linewidth=None*, *color='black'*)
Plot a wireframe representation of a 2D geometry.

`pyiga.vis.`**`plot_hierarchical_cells`**(*hspace*, *cells*, *color_act='steelblue'*, *color_deact='white'*)
Visualize cells of a 2D hierarchical spline space.

Parameters

- **`hspace`** (`HSpace`) – the space to be plotted
- **`cells`** – dict of sets of selected active cells
- **`color_act`** – the color to use for the selected cells
- **`color_deact`** – the color to use for the remaining cells

`pyiga.vis.`**`plot_hierarchical_mesh`**(*hspace*, *levels='all'*, *levelwise=False*, *color_act='steelblue'*, *color_deact='lavender'*)
Visualize the mesh of a 2D hierarchical spline space.

Parameters

- **`hspace`** (`HSpace`) – the space to be plotted
- **`levels`** – either 'all' or a list of levels to plot
- **`levelwise`** (*`bool`*) – if True, show each level (including active and deactivated basis functions) in a separate subplot
- **`color_act`** – the color to use for the active cells
- **`color_deact`** – the color to use for the deactivated cells (only shown if *levelwise* is True)

2.10 Variational forms

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index