
nutils Documentation

Release 3.0

Evalf

Aug 22, 2018

Contents

1	Contents	3
1.1	Introduction	3
1.2	Wiki	4
1.3	Library	5
2	Indices and tables	35
	Python Module Index	37

Nutils: open source numerical utilities for Python, is a collaborative programming effort aimed at the creation of a modern, general purpose programming library for [Finite Element](#) applications and related computational methods. Identifying features are a heavily object oriented design, strict separation of topology and geometry, and CAS-like function arithmetic such as found in Maple and Mathematica. Primary design goals are:

- **Readability.** Finite element scripts built on top of Nutils should focus on work flow and maths, unobscured by Finite Element infrastructure.
- **Flexibility.** The Nutils are tools; they do not enforce a strict work flow. Missing components can be added locally without loosing interoperability.
- **Compatibility.** Exposed objects are of native python type or allow for easy conversion to leverage third party tools.
- **Speed.** Nutils are self-optimizing and support parallel computation. Typical scripting inefficiencies are discouraged by design.

For latest project news and developments visit the project website at nutils.org.

1.1 Introduction

To get one thing out of the way first, note that Nutils is not your classical Finite Element program. It does not have menus, no buttons to click, nothing to make a screenshot of. To get it to do *anything* some programming is going to be required.

That said, let's see what Nutils can be instead.

1.1.1 Design

Nutils is a programming library, providing components that are rich enough to handle a wide range of problems by simply linking them together. This blurs the line between classical graphical user interfaces and a programming environment, both of which serve to offer some degree of mixing and matching of available components. The former has a lower entry bar, whereas the latter offers more flexibility, the possibility to extend the toolkit with custom algorithms, and the possibility to pull in third party modules. It is our strong belief that on the edge of science where Nutils strives to be a great degree of extensibility is adamant.

For those so inclined, one of the lesser interesting possibilities this gives is to write a dedicated, Nutils powered GUI application.

What Nutils specifically does not offer are problem specific components, such as, conceivably, a “crack growth” module or “solve navier stokes” function. As a primary design principle we aim for a Nutils application to be closely readable as a high level mathematical problem description; *i.e.* the weak form, domain, boundary conditions, time stepping of Newton iterations, etc. It is the supporting operations like integrating over a domain or taking gradients of compound functions that are being kept out of sight as much as possible.

1.1.2 Quick demo

As a small but representative demonstration of what is involved in setting up a problem in Nutils we solve the [Laplace problem](#) on a unit square, with zero Dirichlet conditions on the left and bottom boundaries, unit flux at the top and a

natural boundary condition at the right. We begin by creating a structured `nelems` `nelems` Finite Element mesh using the built-in generator:

```
verts = numpy.linspace( 0, 1, nelems+1 )
domain, geom = mesh.rectilinear( [verts,verts] )
```

Here `domain` is topology representing an interconnected set of elements, and `geometry` is a mapping from the topology onto \mathbb{R}^2 , representing its placement in physical space. This strict separation of topological and geometric information is key design choice in Nutils.

Proceeding to specifying the problem, we create a second order spline basis `funcsp` which doubles as trial and test space (u resp. v). We build a matrix by integrating `laplace = $v \cdot u$` over the domain, and a `rhs` vector by integrating `v` over the top boundary. The Dirichlet constraints are projected over the left and bottom boundaries to find constrained coefficients `cons`. Remaining coefficients are found by solving the system in `lhs`. Finally these are contracted with the basis to form our solution function:

```
funcsp = domain.splinefunc( degree=2 )
laplace = function.outer( funcsp.grad(geom) ).sum()
matrix = domain.integrate( laplace, geometry=geom, ischeme='gauss2' )
rhs = domain.boundary['top'].integrate( funcsp, geometry=geom, ischeme='gauss1' )
cons = domain.boundary['left,bottom'].project( 0, ischeme='gauss1', geometry=geom,
↳ onto=funcsp )
lhs = matrix.solve( rhs, constrain=cons, tol=1e-8, symmetric=True )
solution = funcsp.dot(lhs)
```

The solution function is a mapping from the topology onto \mathbb{R}^2 . Sampling this together with the `geometry` generates arrays that we can use for plotting:

```
points, colors = domain.elem_eval( [ geom, solution ], ischeme='bezier4',
↳ separate=True )
with plot.PyPlot( 'solution', index=index ) as plt:
    plt.mesh( points, colors, triangulate='bezier' )
    plt.colorbar()
```

1.2 Wiki

This is a collection of technical notes.

1.2.1 Binary operations on Numpy/Nutils arrays

			Tensor	Einstein	Nutils
1	$\mathbf{a} \in \mathbb{R}^n$	$\mathbf{b} \in \mathbb{R}^n$	$c = \mathbf{a} \cdot \mathbf{b} \in \mathbb{R}$	$c = a_i b_i$	<code>c = (a*b).sum(-1)</code>
2	$\mathbf{a} \in \mathbb{R}^n$	$\mathbf{b} \in \mathbb{R}^m$	$\mathbf{C} = \mathbf{a} \otimes \mathbf{b} \in \mathbb{R}^{n \times m}$	$C_{ij} = a_i b_j$	<code>C = a[:,_]*b[_,:]</code> <code>C = function.outer(a,b)</code>
3	$\mathbf{A} \in \mathbb{R}^{m \times n}$	$\mathbf{b} \in \mathbb{R}^n$	$\mathbf{c} = \mathbf{A}\mathbf{b} \in \mathbb{R}^m$	$c_i = A_{ij} b_j$	<code>c = (A[:, :]*b[_,:]).sum(-1)</code>
4	$\mathbf{A} \in \mathbb{R}^{m \times n}$	$\mathbf{B} \in \mathbb{R}^{n \times p}$	$\mathbf{C} = \mathbf{A}\mathbf{B} \in \mathbb{R}^{m \times p}$	$c_{ij} = A_{ik} B_{kj}$	<code>c = (A[:, :,_]*B[_,,,]).sum(-2)</code>
5	$\mathbf{A} \in \mathbb{R}^{m \times n}$	$\mathbf{B} \in \mathbb{R}^{p \times n}$	$\mathbf{C} = \mathbf{A}\mathbf{B}^T \in \mathbb{R}^{m \times p}$	$C_{ij} = A_{ik} B_{jk}$	<code>C = (A[:, _, :]*B[_,,,]).sum(-1)</code> <code>C = function.outer(A,B).sum(-1)</code>
6	$\mathbf{A} \in \mathbb{R}^{m \times n}$	$\mathbf{B} \in \mathbb{R}^{m \times n}$	$c = \mathbf{A} : \mathbf{B} \in \mathbb{R}$	$c = A_{ij} B_{ij}$	<code>c = (A*B).sum([-2,-1])</code>

Notes:

1. In the above table the summation axes are numbered backward. For example, `sum(-1)` is used to sum over the last axis of an array. Although forward numbering is possible in many situations, backward numbering is generally preferred in Nutils code.
2. When a summation over multiple axes is performed (#6), these axes are to be listed. In the case of single-axis summations listing is optional (for example `sum(-1)` is equivalent to `sum([-1])`). The shorter notation `sum(-1)` is preferred.
3. When the number of dimensions of the two arguments of a binary operation mismatch, singleton axes are automatically prepended to the “shorter” argument. This property can be used to shorten notation. For example, #3 can be written as `(A*b).sum(-1)`. To avoid ambiguities, in general, such abbreviations are discouraged.

1.3 Library

The Nutils are separated in modules focussing on topics such as mesh generation, function manipulation, debugging, plotting, etc. They are designed to form relatively independent units, though some components such as output logging run through all. Others, such as topology and element, operate in tight connection, but are divided for reasons of scope and scale. A typical Nutils application uses methods from all modules, although, as seen above, very few modules require direct access for standard computations.

What follows is an automatically generated API reference.

1.3.1 Topology

The topology module defines the topology objects, notably the `StructuredTopology` and `UnstructuredTopology`. Maintaining strict separation of topological and geometrical information, the topology represents a set of elements and their interconnectivity, boundaries, refinements, subtopologies etc, but not their positioning in physical space. The dimension of the topology represents the dimension of its elements, not that of the the space they are embedded in.

The primary role of topologies is to form a domain for `nutils.function` objects, like the geometry function and function bases for analysis, as well as provide tools for their construction. It also offers methods for integration and sampling, thus providing a high level interface to operations otherwise written out in element loops. For lower level operations topologies can be used as `nutils.element` iterators.

```

class utils.topology.Topology (ndims)
    topology base class

    elem_eval (funcs, ischeme, separate=False, geometry=None, asfunction=False, edit=<function
                <lambda>>, *, arguments=None)
        element-wise evaluation

    elem_mean (funcs, geometry, ischeme, *, arguments=None)
        element-wise average

    integrate (funcs, ischeme='gauss', degree=None, geometry=None, force_dense=False,
                fcache=None, edit=<function <lambda>>, *, arguments=None)

    integral (func, ischeme='gauss', degree=None, geometry=None, edit=<function <lambda>>)

    projection (fun, onto, geometry, **kwargs)
        project and return as function

    project (fun, onto, geometry, tol=0, ischeme='gauss', degree=None, droptol=1e-12, ex-
                act_boundaries=False, constrain=None, verify=None, ptype='lsqr', precon='diag',
                edit=<function <lambda>>, *, arguments=None, **solverargs)
        L2 projection of function onto function space

    refined_by (refine)
        create refined space by refining dofs in existing one

    refine (n)
        refine entire topology n times

    trim (levelset, maxrefine, ndivisions=8, name='trimmed', leveltopo=None, *, arguments=None)
        trim element along levelset

    subset (elements, newboundary=None, strict=False)
        intersection

exception utils.topology.LocateError

class utils.topology.WithGroupsTopology (basetopo, vgroups={}, bgroups={}, igroups={},
                pgroups={})
    item topology

class utils.topology.OppositeTopology (basetopo)
    opposite topology

class utils.topology.EmptyTopology (ndims)
    empty topology

class utils.topology.Point (trans, opposite=None)
    point

class utils.topology.StructuredLine (root, i, j, periodic=False, bnames=None)
    structured topology

    basis_spline (degree, periodic=None, removedofs=None)
        spline from vertices

    basis_discont (degree)
        discontinuous shape functions

    basis_std (degree, periodic=None, removedofs=None)
        spline from vertices

class utils.topology.StructuredTopology (root, axes, nrefine=0, bnames=None)
    structured topology

```

basis_spline (*degree*, *knotvalues=None*, *knotmultiplicities=None*, *periodic=None*, *removedofs=None*)
 spline basis

basis_discont (*degree*)
 discontinuous shape functions

basis_std (*degree*, *removedofs=None*, *periodic=None*)
 spline from vertices

refined
 refine non-uniformly

class nutils.topology.**UnstructuredTopology** (*ndims*, *elements*)
 unstructured topology

basis_bubble ()
 bubble from vertices

basis_discont (*degree*)
 discontinuous shape functions

basis_lagrange (*degree*)
 lagrange shape functions

basis_bernstein (*degree*)
 bernstein shape functions

basis_std (*degree*)
 bernstein shape functions

class nutils.topology.**UnionTopology** (*topos*, *names=()*)
 grouped topology

class nutils.topology.**SubsetTopology** (*basetopo*, *refs*, *newboundary=None*)
 trimmed

class nutils.topology.**OrientedGroupsTopology** (*basetopo*, *elems*)
 unstructured topology with undirected semi-overlapping basetopology

class nutils.topology.**RefinedTopology** (*basetopo*)
 refinement

class nutils.topology.**TrimmedTopologyItem** (*basetopo*, *refdict*)
 trimmed topology item

class nutils.topology.**TrimmedTopologyBoundaryItem** (*btopo*, *trimmed*, *othertopo*)
 trimmed topology boundary item

class nutils.topology.**HierarchicalTopology** (*basetopo*, *allelements*, *precise*)
 collection of nested topology elements

basis (*name*, **args*, ***kwargs*)
 build hierarchical function space

class nutils.topology.**ProductTopology** (*topo1*, *topo2*)
 product topology

refine (*n*)
 refine entire topology n times

class nutils.topology.**RevolutionTopology**
 topology consisting of a single revolution element

```

class nutils.topology.MultipatchTopology (patches)
    multipatch topology

    class Patch (topo, verts, boundaries)

        boundaries
            Alias for field number 2

        topo
            Alias for field number 0

        verts
            Alias for field number 1

    static build_boundarydata (connectivity)
        build boundary data based on connectivity

    basis_spline (degree, patchcontinuous=True, knotvalues=None, knotmultiplicities=None)
        spline from vertices

        Create a spline basis with degree degree per patch. If patchcontinuous` is true the basis is  $C^0$ -
        continuous at patch interfaces.

    basis_discont (degree)
        discontinuous shape functions

    basis_patch ()
        degree zero patchwise discontinuous basis

class nutils.topology.DimAxis (i, j, isperiodic)

    i
        Alias for field number 0

    isperiodic
        Alias for field number 2

    j
        Alias for field number 1

class nutils.topology.BndAxis (i, j, ibound, side)

    i
        Alias for field number 0

    ibound
        Alias for field number 2

    j
        Alias for field number 1

    side
        Alias for field number 3

```

1.3.2 Function

The function module defines the *Evaluable* class and derived objects, commonly referred to as nutils functions. They represent mappings from a *nutils.topology* onto Python space. The notable class of *Array* objects map

onto the space of Numpy arrays of predefined dimension and shape. Most functions used in nutils applications are of this latter type, including the geometry and function bases for analysis.

Nutils functions are essentially postponed python functions, stored in a tree structure of input/output dependencies. Many *Array* objects have directly recognizable numpy equivalents, such as *Sin* or *Inverse*. By not evaluating directly but merely stacking operations, complex operations can be defined prior to entering a quadrature loop, allowing for a higher level style programming. It also allows for automatic differentiation and code optimization.

It is important to realize that nutils functions do not map for a physical xy-domain but from a topology, where a point is characterized by the combination of an element and its local coordinate. This is a natural fit for typical finite element operations such as quadrature. Evaluation from physical coordinates is possible only via inverting of the geometry function, which is a fundamentally expensive and currently unsupported operation.

class `nutils.function.Evaluable` (*args: tuple*)

Base class

asciitree (*seen=None*)

string representation

graphviz ()

create function graph

stackstr (*nlines=-1*)

print stack

exception `nutils.function.EvaluationError` (*etype, value, evaluable, values*)

evaluation error

class `nutils.function.Cache`

class `nutils.function.Points` (*opposite=False*)

class `nutils.function.Tuple` (*items: tuple*)

evalf (**items*)

evaluate

class `nutils.function.TransformChain` (*args: tuple, todims*)

class `nutils.function.SelectChain` (*n, todims=None*)

class `nutils.function.PopHead` (*todims: int, trans=<nutils.function.SelectChain object>*)

class `nutils.function.SelectBifurcation` (*trans, first: bool, todims=None*)

class `nutils.function.Promote` (*ndims: int, trans*)

class `nutils.function.TailOfTransform` (*trans, depth, todims: int*)

class `nutils.function.Array` (*args: tuple, shape: tuple, dtype: nutils.function.<lambda>*)

array function

class `nutils.function.Normal` (*lgrad: nutils.function.<lambda>*)

normal

class `nutils.function.ArrayFunc` (*args: tuple, shape: tuple*)

deprecated ArrayFunc alias

class `nutils.function.Constant` (*value: nutils.numeric.const*)

class `nutils.function.DofMap` (*dofs: tuple, index: nutils.function.<lambda>*)

class `nutils.function.InsertAxis` (*func: nutils.function.<lambda>, axis: int, length: nutils.function.<lambda>*)

```

class nutils.function.Transpose (func: nutils.function.<lambda>, axes: tuple)
class nutils.function.Get (func: nutils.function.<lambda>, axis: int, item: nutils.function.<lambda>)
class nutils.function.Product (func: nutils.function.<lambda>)
class nutils.function.ApplyTransforms (trans, points=<nutils.function.Points object>)
class nutils.function.LinearFrom (trans, fromdims: int)
class nutils.function.Inverse (func: nutils.function.<lambda>)
class nutils.function.Concatenate (funcs: tuple, axis: int = 0)
class nutils.function.Interpolate (x: nutils.function.<lambda>, xp: nutils.numeric.const, fp: nutils.numeric.const, left=None, right=None)
    interpolate uniformly spaced data; stepwise for now
class nutils.function.Cross (func1: nutils.function.<lambda>, func2: nutils.function.<lambda>, axis: int)
class nutils.function.Determinant (func: nutils.function.<lambda>)
class nutils.function.Multiply (funcs: nutils.util.frozenset)
class nutils.function.Add (funcs: nutils.util.frozenset)
class nutils.function.BlockAdd (funcs: nutils.util.frozenset)
    block addition (used for DG)
class nutils.function.Dot (funcs: nutils.util.frozenset, axes: tuple)
class nutils.function.Sum (func: nutils.function.<lambda>, axis: int)
class nutils.function.TakeDiag (func: nutils.function.<lambda>, axis: int, rmaxis: int)
class nutils.function.Take (func: nutils.function.<lambda>, indices: nutils.function.<lambda>, axis: int)
class nutils.function.Power (func: nutils.function.<lambda>, power: nutils.function.<lambda>)
class nutils.function.Pointwise (*args)
class nutils.function.Cos (*args)
class nutils.function.Sin (*args)
class nutils.function.Tan (*args)
class nutils.function.ArcSin (*args)
class nutils.function.ArcCos (*args)
class nutils.function.ArcTan (*args)
class nutils.function.Exp (*args)
class nutils.function.Log (*args)
class nutils.function.Mod (*args)
class nutils.function.ArcTan2 (*args)
class nutils.function.Greater (*args)
class nutils.function.Equal (*args)
class nutils.function.Less (*args)
class nutils.function.Minimum (*args)

```

```

class nutils.function.Maximum(*args)
class nutils.function.Int(*args)
class nutils.function.Sign(func: nutils.function.<lambda>)
class nutils.function.Sampled(data: nutils.util.frozendict, trans=<nutils.function.SelectChain
    object>)
    sampled
class nutils.function.Elemwise(data: tuple, index: nutils.function.<lambda>, dtype: nu-
    tils.function.<lambda>)
class nutils.function.Eig(func: nutils.function.<lambda>, symmetric: bool = False)
class nutils.function.ArrayFromTuple(arrays, index: int, shape: tuple, dtype: nu-
    tils.function.<lambda>)
class nutils.function.Zeros(shape: tuple, dtype: nutils.function.<lambda>)
    zero
class nutils.function.Inflate(func: nutils.function.<lambda>, dofmap: nu-
    tils.function.<lambda>, length: int, axis: int)
class nutils.function.Diagonalize(func: nutils.function.<lambda>, axis=<class 'int'>,
    newaxis=<class 'int'>)
class nutils.function.Guard(func: nutils.function.<lambda>)
    bar all simplifications
class nutils.function.TrigNormal(angle: nutils.function.<lambda>)
    cos, sin
class nutils.function.TrigTangent(angle: nutils.function.<lambda>)
    -sin, cos
class nutils.function.Find(when: nutils.function.<lambda>)
    indices of boolean index vector
class nutils.function.DerivativeTargetBase(args: tuple, shape: tuple, dtype: nu-
    tils.function.<lambda>)
    base class for derivative targets
class nutils.function.Argument(name, shape: tuple, nderiv: int = 0)
    Array argument, to be substituted before evaluation.

```

The *Argument* is an *Array* with a known shape, but whose values are to be defined later, before evaluation, e.g. using `replace_arguments()`.

It is possible to take the derivative of an *Array* to an *Argument*:

```

>>> from nutils import function
>>> a = function.Argument('x', [])
>>> b = function.Argument('y', [])
>>> f = a**3 + b**2
>>> function.derivative(f, a).simplified == (3.*a**2).simplified
True

```

Furthermore, derivatives to the local coordinates are remembered and applied to the replacement when using `replace_arguments()`:

```

>>> from nutils import mesh
>>> domain, x = mesh.rectilinear([2,2])
>>> basis = domain.basis('spline', degree=2)

```

(continues on next page)

(continued from previous page)

```
>>> c = function.Argument('c', basis.shape)
>>> replace_arguments(c.grad(x), dict(c=basis)) == basis.grad(x)
True
```

Parameters

- **name** (*str*) – The Identifier of this argument.
- **shape** (*tuple of ints*) – The shape of this argument.
- **nderiv** (*int*, non-negative) – Number of times a derivative to the local coordinates is taken. Default: 0.

```
class nutils.function.LocalCoords (ndims: int)
    local coords derivative target
```

```
class nutils.function.Ravel (func: nutils.function.<lambda>, axis: int)
```

```
class nutils.function.Unravel (func: nutils.function.<lambda>, axis: int, shape: tuple)
```

```
class nutils.function.Mask (func: nutils.function.<lambda>, mask: nutils.numeric.const, axis: int)
```

```
class nutils.function.FindTransform (transforms: tuple, trans)
```

```
class nutils.function.Range (length: nutils.function.<lambda>, offset: nutils.function.<lambda>
    = Array<>)
```

```
class nutils.function.Polyval (coeffs: nutils.function.<lambda>, points: nutils.function.<lambda>,
    ngrad: int = 0)
    Computes the k-dimensional array
```

$$j_0, \dots, j_{k-1} \mapsto \sum_{\substack{i_0, \dots, i_{n-1} \in \mathbb{N} \\ i_0 + \dots + i_{n-1} \leq d}} p_0^{i_0} \cdots p_{n-1}^{i_{n-1}} c_{j_0, \dots, j_{k-1}, i_0, \dots, i_{n-1}},$$

where p are the n -dimensional local coordinates and c is the argument `coeffs` and d is the degree of the polynomial, where d is the length of the last n axes of `coeffs`.

Warning: All coefficients with a (combined) degree larger than d should be zero. Failing to do so won't raise an `Exception`, but might give incorrect results.

```
nutils.function.chain (funcs)
```

```
nutils.function.bringforward (arg, axis)
    bring axis forward
```

```
nutils.function.matmat (arg0, *args)
    helper function, contracts last axis of arg0 with first axis of arg1, etc
```

```
nutils.function.derivative (func, var, seen=None)
```

```
nutils.function.localgradient (arg, ndims)
    local derivative
```

```
nutils.function.outer (arg1, arg2=None, axis=0)
    outer product
```

```
nutils.function.polyfunc (coeffs, dofs, ndofs, transforms, *, issorted=True)
    Create an inflated Polyval with coefficients coeffs and corresponding dofs. The arguments coeffs,
```


dofs and transforms are assumed to have matching order. In addition, if `issorted` is true, the transforms `argument` is assumed to be sorted.

`nutils.function.find(arg)`

`nutils.function.replace_arguments(value, arguments)`

Replace *Argument* objects in *value*.

Replace *Argument* objects in *value* according to the *arguments* map, taking into account derivatives to the local coordinates.

Parameters

- **value** (*Array*) – Array to be edited.
- **arguments** (`collections.abc.Mapping` with *Arrays* as values) – *Arguments* replacements. The key correspond to the name passed to an *Argument* and the value is the replacement.

Returns The edited value.

Return type *Array*

class `nutils.function.Namespace(*, default_geometry_name='x')`

Namespace for *Array* objects supporting assignments with tensor expressions.

The *Namespace* object is used to store *Array* objects.

```
>>> from nutils import function
>>> ns = function.Namespace()
>>> ns.A = function.zeros([3, 3])
>>> ns.x = function.zeros([3])
>>> ns.c = 2
```

In addition to the assignment of *Array* objects, it is also possible to specify an array using a tensor expression string — see `nutils.expression.parse()` for the syntax. All attributes defined in this namespace are available as variables in the expression. If the array defined by the expression has one or more dimensions the indices of the axes should be appended to the attribute name. Examples:

```
>>> ns.cAx_i = 'c A_ij x_j'
>>> ns.xAx = 'x_i A_ij x_j'
```

It is also possible to simply evaluate an expression without storing its value in the namespace by passing the expression to the method `eval_` suffixed with appropriate indices:

```
>>> ns.eval_('2 c')
Array<>
>>> ns.eval_i('c A_ij x_j')
Array<3>
>>> ns.eval_ij('A_ij + A_ji')
Array<3,3>
```

For zero and one dimensional expressions the following shorthand can be used:

```
>>> '2 c' @ ns
Array<>
>>> 'A_ij x_j' @ ns
Array<3>
```

When evaluating an expression through this namespace the following functions are available: `opposite`, `sin`, `cos`, `tan`, `sinh`, `cosh`, `tanh`, `arcsin`, `arccos`, `arctan2`, `arctanh`, `exp`, `abs`, `ln`, `log`, `log2`, `log10`, `sqrt` and `sign`.

Parameters `default_geometry_name` (*str*) – The name of the default geometry. This argument is passed to `utils.expression.parse()`. Default: `'x'`.

arg_shapes

`types.MappingProxyType` – A readonly map of argument names and shapes.

default_geometry_name

str – The name of the default geometry. See argument with the same name.

default_geometry

`utils.function.Array` – The default geometry, shorthand for `getattr(ns, ns.default_geometry_name)`.

copy_ (*, *default_geometry_name=None*)

Return a copy of this namespace.

1.3.3 Expression

This module defines the function `parse()`, which parses a tensor expression.

exception `utils.expression.ExpressionSyntaxError`

exception `utils.expression.AmbiguousAlignmentError`

`utils.expression.parse(expression, variables, functions, indices, arg_shapes={}, default_geometry_name='x')`

Parse expression and return AST.

This function parses a tensor expression with [Einstein Summation Convection](#) stored in a `str` and returns an Abstract Syntax Tree (AST). The syntax of `expression` is as follows:

- **Integers** or **decimal numbers** are denoted in the usual way. Examples: `1`, `1.2`, `.2`. A number may not start with a zero, except when followed by a dot: `0.1` is valid, but `01` is not.
- **Variables** are denoted with a string of alphanumeric characters. The first character may not be a numeral. Unlike Python variables, underscores are not allowed, as they have a special meaning. If the variable is an array with one or more axes, all those axes should be labeled with a latin character, the index, and appended to the variable with an underscore. For example an array `a` with two axes can be denoted with `a_ij`. Optionally, a single numeral may be used to select an item at the concerning axis. Example: in `a_i0` the first axis of `a` is labeled `i` and the first element of the second axis is selected. If the same index occurs twice, the trace is taken along the concerning axes. Example: the trace of the first and third axes of `b` is denoted by `b_ijj`. It is invalid to specify an index more than twice. The following names cannot be used as variables: `n`, `δ`, `ξ`. The variable named `x`, or the value of argument `default_geometry_name`, has a special meaning, detailed below.
- A term, the **product** of two or more arrays or scalars, is denoted by space-separated variables, constants or compound expressions. Example: `a b c` denotes the product of the scalars `a`, `b` and `c`. A term may start with a number, but a number is not allowed in other parts of the term. Example: `2 a` denotes two times `a`; `2 2 a` and `2 a 2`` are invalid. When two arrays in a term have the same index, this index is summed. Example: `a_i b_i` denotes the inner product of `a` and `b` and `A_ij b_j`` a matrix vector product. It is not allowed to use an index more than twice in a term.
- The operator `/` denotes a **fraction**. Example: in `a b / c d a b` is the numerator and `c d` the denominator. Both the numerator and the denominator may start with a number. Example: `2 a / 3 b`. The denominator must be a scalar. Example: `2 / a_i b_i` is valid, but `2 a_i / b_i` is not.

Warning: This syntax is different from the Python syntax. In Python $a*b / c*d$ is mathematically equivalent to $a*b*d/c$.

- The operators $+$ and $-$ denote **add** and **subtract**. Both operators should be surrounded by whitespace, e.g. $a + b$. Both operands should have the same shape. Example: $a_{ij} + b_{ij} c_j$ is a valid, provided that the lengths of the axes with the same indices match, but $a_{ij} + b_{ij}$ is invalid. At the beginning of an expression or a compound $-$ may be used to negate the following term. Example: in $-a b + c$ the term $a b$ is negated before adding c . It is not allowed to negate other terms: $a + -b$ is invalid, so is $-b$.
- An expression surrounded by parentheses is a **compound expression** and can be used as single entity in a term. Example: $(a_i + b_i) c_i$ denotes the inner product of $a_i + b_i$ with c_i .
- **Exponentiation** is denoted by $a ^$, where the left and right operands should be a number, variable or compound expression and the right operand should be a scalar. Example: a^2 denotes the square of a , a^{-2} denotes a to the power -2 and $a^{(1 / 2)}$ the square root of a .
- An **argument** is denoted by a name — following the same rules as a variable name — prefixed with a question mark. An argument is a scalar or array with a yet unknown value. Example: $basis_i ?coeffs_i$ denotes the inner product of a basis with unknown coefficient vector $?coeffs$. If possible the shape of the argument is deduced from the expression. In the previous example the shape of $?coeffs$ is equal to the shape of $basis$. If the shape cannot be deduced from the expression the shape should be defined manually (see `parse()`). Arguments and variables live in separate namespaces: $?x$ and x are different entities.
- An argument may be **substituted** by appending without whitespace ($arg = value$) to a variable of compound expression, where arg is an argument and $value$ the substitution. The substitution applies to the variable of compound expression only. The value may be an expression. Example: $2 ?x(x = 3 + y)$ is equivalent to $2 (3 + y)$ and $2 ?x(x=y) + 3$ is equivalent to $2 (y) + 3$. It is possible to apply multiple substitutions. Example: $(?x + ?y)(x = 1, y =)2$ is equivalent to $1 + 2$.
- The **gradient** of a variable to the default geometry — the default geometry is variable x unless overridden by the argument `default_geometry_name` — is denoted by an underscore, a comma and an index. If the variable is an array with more than one axis, the underscore is omitted. Example: $a_{,i}$ denotes the gradient of the scalar a to the geometry and $b_{i,j}$ the gradient of vector b . The gradient of a compound expression is denoted by an underscore, a comma and an index. Example: $(a_i + b_j)_{,k}$ denotes the gradient of $a_i + b_j$. The usual summation rules apply and it is allowed to use a numeral as index. The **surface gradient** is denoted with a semicolon instead of a comma, but follows the same rules as the gradient otherwise. Example: $a_i ; j$ is the surface gradient of a_i to the geometry. It is also possible to take the gradient to another geometry by appending the name of the geometry, which should exist as a variable, and an underscore directly after the comma or semicolon. Example: $a_i , altgeom_j$ denotes the gradient of a_i to `altgeom` and the gradient axis has index j . Furthermore, it is possible to take the **derivative** to an argument by adding the argument with appropriate indices after the comma. Example: $(?x^2)_{,?x}$ denotes the derivative of $?x^2$ to $?x$, which is equivalent to $2 ?x$, and $(?y_i ?y_i)_{,?y_j}$ is the derivative of $?y_i ?y_i$ to $?y_j$, which is equivalent to $2 ?y_j$.
- The **normal** of the default geometry is denoted by $n_{,i}$, where the index i may be replaced with an index of choice. The normal with respect to different geometry is denoted by appending an underscore with the name of the geometry right after n . Example: $n_{,altgeom_j}$ is the normal with respect to geometry `altgeom`.
- A **dirac** is denoted by δ or $\$$ and takes two indices. The shape of the dirac is deduced from the expression. Example: let A be a square matrix with three rows and columns, then δ_{ij} in $(A_{ij} - \lambda \delta_{ij}) x_j$ has three rows and columns as well.
- An expression surrounded by square brackets or curly braces denotes the **jump** or **mean**, respectively, of the enclosed expression. Example: $[a_i]$ denotes the jump of a_i and $\{ a_i + b_i \}$ denotes

the mean of $a_i + b_i$.

- A **function call** is denoted by a name — following the same rules as for a variable name — directly followed by the left parenthesis (, without a space. The arguments to the function are separated by a comma and at least one space. The function is applied pointwise to the arguments and all arguments should have the same shape. Example: $f(x_i, y_i)$ denotes the call to function f with arguments x_i and y_i . Functions and variables share a namespace: defining a variable with the same name as a function renders the function inaccessible.
- A **stack** of two or more arrays along an axis is denoted by a $<$ followed by comma and space separated arrays followed by $>$ and an index. If an argument does not have an axis with the specified stack index, the argument is expanded with an axis of length one. Beside the stack axis, all arguments should have the same shape. Example: $<1, x_i>_i$, with x a vector of length three, creates an array with components $1, x_0, x_1, x_2$.

Parameters

- **expression** (*str*) – The expression to parse. See *expression* for the expression syntax.
- **variables** (*dict* of *str* and *nutils.function.Array* pairs) – A *dict* of variable names and array pairs. All variables used in the expression should exist in variables.
- **functions** (*dict* of *str* and *int* pairs) – A *dict* of function names and number of arguments pairs. All functions used in the expression should exist in functions.
- **indices** (*str*) – The indices used for aligning the resulting array. For example, let expression be 'a_ij'. If indices is 'ij', then the returned array is simply variables['a'], but if indices is 'ji' the transpose of variables['a'] is returned. All indices of the expression should be listed precisely once.
- **arg_shapes** (*dict* of *str* and *tuple* or *ints* pairs) – A *dict* of argument names and shapes. If expression contains an argument not present in arg_shapes the shape will be deduced from the expression and added to a copy of arg_shapes.
- **default_geometry_name** (*str*) – The name of the default geometry variable. When computing a gradient or the normal, e.g. 'f_,i' or 'n_i', this variable is used as the geometry, unless the geometry is explicitly mentioned in the expression. Default: 'x'.

Returns

- **ast** (*tuple*) – The parsed expression as an abstract syntax tree (AST). The AST is a *tuple* of an opcode and arguments. The special opcode `None` indicates that the single argument is used verbatim. All other opcodes have AST as arguments. The following opcodes exist:

```
(None, const)
('group', group)
('arg', name, *shape)
('substitute', array, arg, value)
('call', func, arg)
('eye', length)
('normal', geom)
('getitem', array, dim, index)
('trace', array, n1, n2)
('sum', array, axis)
('concatenate', *args)
('grad', array, geom)
```

(continues on next page)

(continued from previous page)

```

('surfgrad', array, geom)
('derivative', func, target)
('append_axis', array, length)
('transpose', array, trans)
('jump', array)
('mean', array)
('neg', array)
('add', left, right)
('sub', left, right)
('mul', left, right)
('truediv', left, right)
('pow', left, right)

```

- **arg_shapes** (dict of str and tuple of ints pairs) – A copy of `arg_shapes` updated with shapes of arguments present in this expression.

1.3.4 Core

The core module provides a collection of low level constructs that have no dependencies on other nutils modules. Primarily for internal use.

```
nutils.core.open_in_outdir(file, *args, **kwargs)
    open a file relative to the outdirfd or outdir property
```

Wrapper around `open()` that opens a file relative to either the `outdirfd` property (if supported, see `os.supports_dir_fd()`) or `outdir`. Takes the same arguments as `open()`.

```
nutils.core.listoutdir()
    list files in outdirfd or outdir property
```

1.3.5 Config

This module holds the Nutils global configuration, stored as (immutable) attributes. To inspect the current configuration, use `print()` or `vars()` on this module. The configuration can be changed temporarily by calling this module with the new settings passed as keyword arguments and entering the returned context. The old settings are restored as soon as the context is exited. Example:

```

>>> from nutils import config
>>> config.verbose
4
>>> with config(verbose=2, nprocs=4):
...     # The configuration has been updated.
...     config.verbose
2
>>> # Exiting the context reverts the changes:
>>> config.verbose
4

```

Note: The default entry point for Nutils scripts `nutils.cli.run()` (and `nutils.cli.choose()`) will read user configuration from disk.

Important: The configuration is not thread-safe: changing the configuration inside a thread changes the process wide configuration.

1.3.6 Element

The element module defines reference elements such as the `QuadElement` and `TriangularElement`, but also more exotic objects like the `TrimmedElement`. A set of (interconnected) elements together form a `nutils.topology`. Elements have edges and children (for refinement), which are in turn elements and map onto self by an affine transformation. They also have a well defined reference coordinate system, and provide pointsets for purposes of integration and sampling.

class `nutils.element.Element` (*reference, trans, opptrans=None, oriented=False*)
element class

class `nutils.element.Reference` (*ndims: int*)
reference element

trim (*levels, maxrefine, ndivisions*)
trim element along levelset

class `nutils.element.EmptyReference` (*ndims: int*)
inverse reference element

trim (*levels, maxrefine, ndivisions*)
trim element along levelset

class `nutils.element.RevolutionReference`
modify gauss integration to always return a single point

class `nutils.element.SimplexReference` (*ndims: int*)
simplex reference

class `nutils.element.PointReference`
0D simplex

class `nutils.element.LineReference`
1D simplex

class `nutils.element.TriangleReference`
2D simplex

getischeme_gauss (*degree*)
get integration scheme <http://www.cs.rpi.edu/~flaherje/pdf/fea6.pdf>

class `nutils.element.TetrahedronReference`
3D simplex

getischeme_gauss (*degree*)
get integration scheme <http://www.cs.rpi.edu/~flaherje/pdf/fea6.pdf>

class `nutils.element.TensorReference` (*ref1, ref2*)
tensor reference

class `nutils.element.Cone` (*edgeref, etrans, tip: nutils.numeric.const*)
cone

class `nutils.element.NeighborhoodTensorReference` (*ref1, ref2, neighborhood, transf*)
product reference element

get_tri_bem_ischeme (*ischeme*)

Some cached quantities for the singularity quadrature scheme.

get_quad_bem_ischeme (*ischeme*)

Some cached quantities for the singularity quadrature scheme.

get_ischeme_singular (*n*)

get integration scheme

class `nutils.element.OwnChildReference` (*baseref*)

forward self as child

class `nutils.element.WithChildrenReference` (*baseref*, *child_refs: tuple*)

base reference with explicit children

permutation_transforms

tuple() -> empty tuple tuple(iterable) -> tuple initialized from iterable's items

If the argument is a tuple, the return value is the same object.

get_ischeme (*ischeme*)

get integration scheme

class `nutils.element.MosaicReference` (*baseref*, *edge_refs: tuple*, *midpoint: nutils.numeric.const*)

triangulation

get_ischeme (*ischeme*)

get integration scheme

1.3.7 Log

The log module provides print methods `debug`, `info`, `user`, `warning`, and `error`, in increasing order of priority. Output is sent to `stdout` as well as to an html formatted log file if so configured.

class `nutils.log.Log`

Base class for log objects. A subclass should define a `context()` method that returns a context manager which adds a contextual layer and a `write()` method.

context (*title*)

Return a context manager that adds a contextual layer named `title`.

Note: This function is abstract.

write (*level*, *text*)

Write `text` with log level `level` to the log.

Note: This function is abstract.

class `nutils.log.ContextLog`

Base class for loggers that keep track of the current list of contexts.

The base class implements `context()` which keeps the attribute `_context` up-to-date.

_context

A list of contexts (`strs`) that are currently active.

context (*title*)

Return a context manager that adds a contextual layer named *title*.

The list of currently active contexts is stored in `_context`.

class `nutils.log.ContextTreeLog`

Base class for loggers that display contexts as a tree.

`_print_push_context` (*title*)

Push a context to the log.

This method is called just before the first item of this context is added to the log. If no items are added to the log within this context or children of this context this method nor `_print_pop_context()` will be called.

Note: This function is abstract.

`_print_pop_context` ()

Pop a context from the log.

This method is called whenever a context is exited, but only if `_print_push_context()` has been called before for the same context.

Note: This function is abstract.

`_print_item` (*level, text*)

Add an item to the log.

Note: This function is abstract.

write (*level, text*)

Write *text* with log level *level* to the log.

This method makes sure the current context is printed and calls `_print_item()`.

class `nutils.log.StdoutLog` (*stream=None*)

Output plain text to stream.

write (*level, text, endl=True*)

Write *text* with log level *level* to the log.

Note: This function is abstract.

class `nutils.log.RichOutputLog` (*stream=None, *, progressinterval=None*)

Output rich (colored,unicode) text to stream.

class `nutils.log.HtmlInsertAnchor`

Mix-in class for HTML-based loggers that inserts anchor tags for paths.

`_insert_anchors` (*level, escaped_text*)

Insert anchors for all paths in *escaped_text*.

Note: *escaped_text* should be valid html (e.g. the result of `html.escape(text)`).

```
class nutils.log.HtmlLog (file, *, title='nutils', scriptname=None, funcname=None, funcargs=None)
```

Output html nested lists.

```
write (level, text)
```

Write *text* with log level *level* to the log.

This method makes sure the current context is printed and calls `_print_item()`.

```
write_post_mortem (etype, value, tb)
```

write exception info to html log

```
class nutils.log.IndentLog (file, *, progressfile=None, progressinterval=None)
```

Output indented html snippets.

```
class nutils.log.TeeLog (*logs)
```

Simultaneously interface multiple logs

```
context (title)
```

Return a context manager that adds a contextual layer named *title*.

Note: This function is abstract.

```
write (level, text)
```

Write *text* with log level *level* to the log.

Note: This function is abstract.

```
nutils.log.range (title, *args)
```

Progress logger identical to built in range

```
nutils.log.iter (title, iterable, length=None)
```

Progress logger identical to built in iter

```
nutils.log.enumerate (title, iterable)
```

Progress logger identical to built in enumerate

```
nutils.log.zip (title, *iterables)
```

Progress logger identical to built in enumerate

```
nutils.log.count (title, start=0, step=1)
```

Progress logger identical to itertools.count

```
nutils.log.title (f)
```

Decorator, adds title argument with default value equal to the name of the decorated function, unless argument already exists. The title value is used in a static log context that is destructed with the function frame.

1.3.8 Matrix

The matrix module defines a number of 2D matrix objects, notably the `ScipyMatrix()` and `NumpyMatrix()`. Matrix objects support basic addition and subtraction operations and provide a consistent interface for solving linear systems. Matrices can be converted to numpy arrays via `toarray` or scipy matrices via `toscopy`.

```
class nutils.matrix.Matrix (shape)
```

matrix base class

```
cond (constrain=None, lconstrain=None, rconstrain=None)
```

condition number

res (*x, b=0, constrain=None, lconstrain=None, rconstrain=None, scaled=True*)
 residual

class `nutils.matrix.ScipyMatrix` (*core*)
 matrix based on any of scipy's sparse matrices

rowsupp (*tol=0*)
 return row indices with nonzero/non-small entries

solve (*rhs=None, constrain=None, lconstrain=None, rconstrain=None, tol=0, lhs0=None, solver=None, symmetric=False, callback=None, precon=None, **solverargs*)

class `nutils.matrix.NumpyMatrix` (*core*)
 matrix based on numpy array

solve (*b=None, constrain=None, lconstrain=None, rconstrain=None, tol=0*)

`nutils.matrix.assemble` (*data, index, shape, force_dense=False*)
 create data from values and indices

`nutils.matrix.parsecons` (*constrain, lconstrain, rconstrain, shape*)
 parse constraints

1.3.9 Mesh

The mesh module provides mesh generators: methods that return a topology and an accompanying geometry function. Meshes can either be generated on the fly, e.g. `rectilinear()`, or read from external an externally prepared file, `gmsh()`, and converted to nutils format. Note that no mesh writers are provided at this point; output is handled by the `nutils.plot` module.

`nutils.mesh.rectilinear` (*richshape, periodic=(), name='rect'*)
 rectilinear mesh

`nutils.mesh.multipatch` (*patches, nelems, patchverts=None, name='multipatch'*)
 multipatch rectilinear mesh generator

Generator for a `MultipatchTopology` and geometry. The `MultipatchTopology` consists of a set patches, where each patch is a `StructuredTopology` and all patches have the same number of dimensions.

The `patches` argument, a `numpy.ndarray`-like with shape `(npatches, 2*ndims)` or `(npatches,)+(2,)*ndims`, defines the connectivity by labelling the patch vertices. For example, three one-dimensional patches can be connected at one edge by:

```
# connectivity:      3
#                   |
#                   1—0—2
patches=[[0,1], [0,2], [0,3]]
```

Or two two-dimensional patches along an edge by:

```
# connectivity:  3—4—5
#               | | |
#               0—1—2
patches=[[0,3], [1,4]], [[1,4], [2,5]]
```

The geometry is specified by the `patchverts` argument: a `numpy.ndarray`-like with shape `(nverts, geomdims)` specifying for each vertex a coordinate. Note that the dimension of the geometry may be higher

than the dimension of the patches. The created geometry is a patch-wise linear interpolation of the vertex coordinates. If the `patchverts` argument is omitted the geometry describes a unit hypercube per patch.

The `nelems` argument is either an `int` defining the number of elements per patch per dimension, or a `dict` with edges (a pair of vertex numbers) as keys and the number of elements (`int`) as values, with key `None` specifying the default number of elements. Example:

```
# connectivity:  3-----4-----5
#               | 4x3 | 8x3 |
#               0-----1-----2

patches=[[0,3],[1,4]], [[1,4],[2,5]]
nelems={None: 4, (1,2): 8, (4,5): 8, (0,3): 3, (1,4): 3, (2,5): 3}
```

Since the patches are structured topologies, the number of elements per patch per dimension should be unambiguous. In above example specifying `nelems={None: 4, (1,2): 8}` will raise an exception because the patch on the right has 8 elements along edge (1,2) and 4 along (4,5).

Example

An L-shaped domain can be generated by:

```
# connectivity:  2-----5
#               | |
#               1-4-----7      y
#               | |         |
#               0-3-----6      |
#                               |
#                               x

domain, geom = mesh.multipatch(
    patches=[[0,1,3,4], [1,2,4,5], [3,4,6,7]],
    patchverts=[[0,0], [0,1], [0,2], [1,0], [1,1], [1,2], [3,0], [3,1]],
    nelems={None: 4, (3,6): 8, (4,7): 8})
```

The number of elements is chosen such that all elements in the domain have the same size.

A topology and geometry describing the surface of a sphere can be generated by creating a multipatch cube surface and inflating the cube to a sphere:

```
# connectivity:  3-----7
#               | |
#               2-----6      y
#               | |         |
#               | 1-----5      z
#               | |         |
#               0-----4      *-----x

topo, cube = multipatch(
    patches=[
        # The order of the vertices is chosen such that normals point outward.
        [2,3,0,1],
        [4,5,6,7],
        [4,6,0,2],
        [1,3,5,7],
        [1,5,0,4],
        [2,6,3,7],
    ],
    patchverts=tuple(itertools.product(*([[ -1, 1]]*3))),
```

(continues on next page)

```

    nelems=10,
)
sphere = cube / function.sqrt((cube**2).sum(0))

```

Parameters

- **patches** – A `numpy.ndarray` with shape sequence of patches with each patch being a list of vertex indices.
- **patchverts** – A sequence of coordinates of the vertices.
- **nelems** – Either an `int` specifying the number of elements per patch per dimension, or a `dict` with edges (a pair of vertex numbers) as keys and the number of elements (`int`) as values, with key `None` specifying the default number of elements.

Returns

- `nutils.topology.MultipatchTopology` – The multipatch topology.
- `nutils.function.Array` – The geometry defined by the `patchverts` or a unit hypercube per patch if `patchverts` is not specified.

`nutils.mesh.gmsh` (*fname*, *name=None*)
Gmsh parser

Parser for Gmsh files in `.msh` format. Only files with physical groups are supported. See the [Gmsh manual](#) for details.

Parameters

- **fname** (*str*) – Path to mesh file
- **name** (*str*, *optional*) – Name of parsed topology, defaults to `None`

Returns Topology of parsed Gmsh file geom (`nutils.function.Array`): Isoparametric map

Return type topo (`nutils.topology.Topology`)

`nutils.mesh.fromfunc` (*func*, *nelems*, *ndims*, *degree=1*)
piecewise

`nutils.mesh.demo` (*xmin=0*, *xmax=1*, *ymin=0*, *ymax=1*)
demo triangulation of a rectangle

1.3.10 Numeric

The numeric module provides methods that are lacking from the `numpy` module.

`nutils.numeric.overlapping` (*arr*, *axis=-1*, *n=2*)
reinterpret data with overlaps

`nutils.numeric.normdim` (*ndim*, *n*)
check bounds and make positive

`nutils.numeric.get` (*arr*, *axis*, *item*)
take single item from array axis

`nutils.numeric.contract` (*A*, *B*, *axis=-1*)

`nutils.numeric.dot` (*A*, *B*, *axis=-1*)
 Transform axis of *A* by contraction with first axis of *B* and inserting remaining axes. Note: with default *axis=-1* this leads to multiplication of vectors and matrices following linear algebra conventions.

`nutils.numeric.meshgrid` (**args*)
 multi-dimensional meshgrid generalisation

`nutils.numeric.normalize` (*A*, *axis=-1*)
 divide by normal

`nutils.numeric.diagonalize` (*arg*, *axis=-1*, *newaxis=-1*)
 insert newaxis, place axis on diagonal of axis and newaxis

`nutils.numeric.ix` (*args*)
 version of `numpy.ix_()` that allows for scalars

`nutils.numeric.ext` (*A*)
 Exterior For array of shape (n,n-1) return n-vector *ex* such that *ex.array* = 0 and `det(arr;ex) = ex.ex`

class `nutils.numeric.const`

1.3.11 Parallel

The parallel module provides tools aimed at parallel computing. At this point all parallel solutions use the `fork` system call and are supported on limited platforms, notably excluding Windows. On unsupported platforms parallel features will disable and a warning is printed.

`nutils.parallel.shempty` (*shape*, *dtype=<class 'float'>*)
 create uninitialized array in shared memory

`nutils.parallel.shzeros` (*shape*, *dtype=<class 'float'>*)
 create zero-initialized array in shared memory

`nutils.parallel.pariter` (*iterable*, *nprocs*)
 iterate in parallel

Fork into *nprocs* subprocesses, then yield items from *iterable* such that all processes receive a nonoverlapping subset of the total. It is up to the user to prepare shared memory and/or locks for inter-process communication. The following creates a data vector containing the first four quadratics:

```
data = shzeros(shape=[4], dtype=int)
for i in pariter(range(4), 2):
    data[i] = i**2
data
```

As a safety measure nested pariters are blocked by setting the global `procid` variable; all secondary pariters will be treated like normal serial iterators.

Parameters

- **iterable** (*iterable*) – The collection of items to be distributed over processors
- **nprocs** (*int*) – Maximum number of processers to use

Yields *Items from iterable, distributed over at most nprocs processors.*

`nutils.parallel.parmap` (*func*, *iterable*, *nprocs*, *shape=()*, *dtype=<class 'float'>*)
 parallel equivalent to builtin map function

Produces an array of `func(item)` values for all items in *iterable*. Because of shared memory restrictions *func* must yield numpy arrays of predetermined shape and type.

Parameters

- **func** (*python function*) – Takes item from iterable, returns numpy array of shape and dtype
- **iterable** (*iterable*) – Collection of items
- **nprocs** (*int*) – Maximum number of processors to use
- **shape** (*tuple*) – Return shape of func, defaults to scalar
- **dtype** (*tuple*) – Return dtype of func, defaults to float

Returns

Return type Array of shape `len(iterable), +shape` and dtype `dtype`

1.3.12 Util

The util module provides a collection of general purpose methods. Most importantly it provides the `run()` method which is the preferred entry point of a numutils application, taking care of command line parsing, output dir creation and initiation of a log file.

```
numutils.util.sum()
    reduce(function, sequence[, initial]) -> value
```

Apply a function of two arguments cumulatively to the items of a sequence, from left to right, so as to reduce the sequence to a single value. For example, `reduce(lambda x, y: x+y, [1, 2, 3, 4, 5])` calculates $((((1+2)+3)+4)+5)$. If initial is present, it is placed before the items of the sequence in the calculation, and serves as a default when the sequence is empty.

```
numutils.util.product()
    reduce(function, sequence[, initial]) -> value
```

Apply a function of two arguments cumulatively to the items of a sequence, from left to right, so as to reduce the sequence to a single value. For example, `reduce(lambda x, y: x*y, [1, 2, 3, 4, 5])` calculates $((((1*2)*3)*4)*5)$. If initial is present, it is placed before the items of the sequence in the calculation, and serves as a default when the sequence is empty.

```
class numutils.util.NanVec(**attrs)
    nan-initialized vector
```

```
numutils.util.obj2str(obj)
    compact, lossy string representation of arbitrary object
```

```
numutils.util.single_or_multiple(f)
    Method wrapper, converts first positional argument to tuple: tuples/lists are passed on as tuples, other objects are turned into tuple singleton. Return values should match the length of the argument list, and are unpacked if the original argument was not a tuple/list.
```

```
>>> class Test:
...     @single_or_multiple
...     def square(self, args):
...         return [v**2 for v in args]
...
>>> T = Test()
>>> T.square(2)
4
>>> T.square([2,3])
[4, 9]
```

Parameters `f` (*method*) – Method that expects a tuple as first positional argument, and that returns a list/tuple of the same length.

Returns

Return type Wrapped method.

1.3.13 Plot

The plot module aims to provide a consistent interface to various plotting backends. At this point `matplotlib` and `vtk` are supported.

class `nutils.plot.BasePlot` (*name=None, ndigits=0, index=None*)
base class for plotting objects

class `nutils.plot.PyPlot` (*name=None, imgtype=None, ndigits=3, index=None, **kwargs*)
matplotlib figure

close ()
close figure

save (*name=None, index=None, **kwargs*)
save images

segments (*points, color='black', **kwargs*)
plot line

mesh (*points, values=None, edgcolors='k', edgewidth=0.1, mergetol=0, setxylim=True, aspect='equal', tight=True, **kwargs*)
plot elementwise mesh

polycol (*verts, facecolors='none', **kwargs*)
add polycollection

slope_marker (*x, y, slope=None, width=0.2, xoffset=0, yoffset=0.2, color='0.5'*)
slope marker

slope_triangle (*x, y, fillcolor='0.9', edgcolor='k', xoffset=0, yoffset=0.1, slopefmt='{0:.1f}'*)
Draw slope triangle for supplied $y(x)$ - x , y : coordinates - `xoffset`, `yoffset`: distance graph & triangle (points) - `fillcolor`, `edgcolor`: triangle style - `slopefmt`: format string for slope number

slope_trend (*x, y, lt='k-', xoffset=0.1, slopefmt='{0:.1f}'*)
Draw slope triangle for supplied $y(x)$ - x , y : coordinates - `slopefmt`: format string for slope number

rectangle (*x0, w, h, fc='none', ec='none', **kwargs*)

griddata (*xlim, ylim, data*)
plot griddata

cspy (*A, **kwargs*)
Like `pyplot.spy`, but coloring acc to 10^{\log} of absolute values, where $[0, \text{inf}, \text{nan}]$ show up in blue.

class `nutils.plot.PyPlotVideo` (*name, videotype=None, clearfigure=True, framerate=24*)
matplotlib based video generator

Video generator based on matplotlib figures. Follows the same syntax as `PyPlot`.

Parameters

- **clearfigure** (*bool, default: True*) – If `True` clears the matplotlib figure after writing each frame.

- **framerate** (*int*, *float*, *default: 24*) – Framerate in frames per second of the generated video.
- **videotype** (*str*, *default: 'webm'* unless overridden by property `videotype`) – Video type of the generated video. Note that not every video type supports playback before the video has been finalized, i.e. before `close` has been called.
- **properties** (*Nutils*) –
- -----
- **videotype** – see parameter with the same name
- **videoencoder** (*str*, *default: 'ffmpeg'*) – Name or path of the video encoder. The video encoder should take the same arguments as 'ffmpeg'.

Examples

Using a with-statement:

```
video = PyPlotVideo('video')
for timestep in timesteps:
    ...
    with video:
        video.plot(...)
        video.title('frame {:04d}'.format(video.frame))
video.close()
```

Using saveframe:

```
video = PyPlotVideo('video')
for timestep in timesteps:
    ...
    video.plot(...)
    video.title('frame {:04d}'.format(video.frame))
    video.saveframe()
video.close()
```

saveframe()
add a video frame

close()
finalize video

class `nutils.plot.DataFile` (*name=None*, *index=None*, *ext='txt'*, *ndigits=0*)
data file

class `nutils.plot.VTKFile` (*name=None*, *index=None*, *ndigits=0*, *ascii=False*)
vtk file

rectilineargrid (*coords*)
set rectilinear grid

unstructuredgrid (*cellpoints*, *npars=None*)
set unstructured grid

celldataarray (*name*, *data*)
add cell array

pointdataarray (*name*, *data*)
add cell array

`nutils.plot.writevtu`(*name, topo, coords, pointdata={}, celldata={}, ascii=False, superelements=False, maxrefine=3, ndigits=0, ischeme='gauss1', **kwargs*)
write vtu from coords function

1.3.14 Cache

The cache module.

class `nutils.cache.Wrapper` (*func*)
function decorator that caches results by arguments

class `nutils.cache.WrapperCache`
maintains a cache for Wrapper instances

class `nutils.cache.WrapperDummyCache`
placeholder object

class `nutils.cache.FileCache` (**args*)
cache

`nutils.cache.replace` (*func*)
decorator for deep object replacement

Generates a deep replacement method for Immutable objects based on a callable that is applied (recursively) on individual constructor arguments.

Parameters `func` (*callable which maps (obj, ..) onto replaced_obj*) -

Returns The method that searches the object to perform the replacements.

Return type callable

1.3.15 Cli

The cli (command line interface) module provides the `cli.run` function that can be used set up properties, initiate an output environment, and execute a python function based arguments specified on the command line.

`nutils.cli.run` (*func, *, skip=1, loaduserconfig=True*)
parse command line arguments and call function

`nutils.cli.choose` (**functions, loaduserconfig=True*)
parse command line arguments and call one of multiple functions

`nutils.cli.call` (*func, kwargs, scriptname, funcname=None*)
set up compute environment and call function

1.3.16 Solver

The solver module defines the `Integral` class, which represents an unevaluated integral. This is useful for fully automated solution procedures such as Newton, that require functional derivatives of an entire functional.

To demonstrate this consider the following setup:

```
>>> from nutils import mesh, function, solver
>>> ns = function.Namespace()
>>> domain, ns.x = mesh.rectilinear([4,4])
>>> ns.basis = domain.basis('spline', degree=2)
>>> cons = domain.boundary['left,top'].project(0, onto=ns.basis, geometry=ns.x,
-> ischeme='gauss4')
```

(continues on next page)

(continued from previous page)

```
project > constrained 11/36 dofs, error 0.00e+00/area
>>> ns.u = 'basis_n ?lhs_n'
```

Function `u` represents an element from the discrete space but cannot not evaluated yet as we did not yet establish values for `?lhs`. It can, however, be used to construct a residual functional `res`. Aiming to solve the Poisson problem $u_{,kk} = f$ we define the residual functional $res = v_{,k} u_{,k} + v f$ and solve for $res == 0$ using `solve_linear`:

```
>>> res = domain.integral('basis_n,i u_{,i} + basis_n' @ ns, geometry=ns.x, degree=2)
>>> lhs = solver.solve_linear('lhs', residual=res, constrain=cons)
solve > solving system using sparse direct solver
```

The coefficients `lhs` represent the solution to the Poisson problem.

In addition to `solve_linear` the solver module defines `newton` and `pseudotime` for solving nonlinear problems, as well as `impliciteuler` for time dependent problems.

class `nutils.solver.Integral` (*integrands*)

Postponed integral, used for derivative purposes

exception `nutils.solver.ModelError`

`nutils.solver.solve_linear` (*target, residual, constrain=None, *, arguments=None, **solveargs*)
solve linear problem

Parameters

- **target** (*str*) – Name of the target: a *nutils.function.Argument* in residual.
- **residual** (*Integral*) – Residual integral, depends on target
- **constrain** (*float vector*) – Defines the fixed entries of the coefficient vector
- **arguments** (*collections.abc.Mapping*) – Defines the values for *nutils.function.Argument* objects in *residual*. The target should not be present in arguments. Optional.

Returns Array of target values for which $residual == 0$

Return type vector

`nutils.solver.solve` (*gen_lhs_resnorm, tol=1e-10, maxiter=None*)
execute nonlinear solver

Iterates over nonlinear solver until tolerance is reached. Example:

```
lhs = solve(newton(target, residual), tol=1e-5)
```

Parameters

- **gen_lhs_resnorm** (*generator*) – Generates (lhs, resnorm) tuples
- **tol** (*float*) – Target residual norm
- **maxiter** (*int*) – Maximum number of iterations

Returns Coefficient vector that corresponds to a smaller than `tol` residual.

Return type vector

`nutils.solver.withsolve` (*f*)
add a `.solve` method to (lhs,resnorm) iterators

Introduces the convenient form:

```
newton(target, residual).solve(tol)
```

Shorthand for:

```
solve(newton(target, residual), tol)
```

class `nutils.solver.newton` (**args, **kwargs*)
iteratively solve nonlinear problem by gradient descent

Generates targets such that residual approaches 0 using Newton procedure with line search based on a residual integral. Suitable to be used inside `solve`.

An optimal relaxation value is computed based on the following cubic assumption:

$$| \text{res}(\text{lhs} + r * \text{dlhs}) |^2 = A + B * r + C * r^2 + D * r^3$$

where A, B, C and D are determined based on the current and updated residual and tangent.

Parameters

- **target** (*str*) – Name of the target: a `nutils.function.Argument` in residual.
- **residual** (*Integral*) –
- **lhs0** (*vector*) – Coefficient vector, starting point of the iterative procedure.
- **constrain** (*boolean or float vector*) – Equal length to `lhs0`, masks the free vector entries as `False` (boolean) or `NaN` (float). In the remaining positions the values of `lhs0` are returned unchanged (boolean) or overruled by the values in `constrain` (float).
- **nrelax** (*int*) – Maximum number of relaxation steps before proceeding with the updated coefficient vector (by default unlimited).
- **minrelax** (*float*) – Lower bound for the relaxation value, to force re-evaluating the functional in situation where the parabolic assumption would otherwise result in unreasonably small steps.
- **maxrelax** (*float*) – Relaxation value below which relaxation continues, unless `nrelax` is reached; should be a value less than or equal to 1.
- **rebound** (*float*) – Factor by which the relaxation value grows after every update until it reaches unity.
- **arguments** (*collections.abc.Mapping*) – Defines the values for `nutils.function.Argument` objects in `residual`. The target should not be present in `arguments`. Optional.

Yields *vector* – Coefficient vector that approximates `residual==0` with increasing accuracy

class `nutils.solver.pseudotime` (**args, **kwargs*)
iteratively solve nonlinear problem by pseudo time stepping

Generates targets such that residual approaches 0 using hybrid of Newton and time stepping. Requires an inertia term and initial timestep. Suitable to be used inside `solve`.

Parameters

- **target** (*str*) – Name of the target: a `nutils.function.Argument` in residual.

- **residual** (*Integral*) –
- **inertia** (*Integral*) –
- **timestep** (*float*) – Initial time step, will scale up as residual decreases
- **lhs0** (*vector*) – Coefficient vector, starting point of the iterative procedure.
- **constrain** (*boolean or float vector*) – Equal length to `lhs0`, masks the free vector entries as `False` (boolean) or `NaN` (float). In the remaining positions the values of `lhs0` are returned unchanged (boolean) or overruled by the values in `constrain` (float).
- **arguments** (*collections.abc.Mapping*) – Defines the values for `nutils.function.Argument` objects in `residual`. The `target` should not be present in `arguments`. Optional.

Yields *vector, float* – Tuple of coefficient vector and residual norm

```
nutils.solver.thetamethod(target, residual, inertia, timestep, lhs0, theta, target0='_thetamethod_target0', constrain=None, newtontol=1e-10, *, arguments=None, **newtonargs)
```

solve time dependent problem using the theta method

Parameters

- **target** (*str*) – Name of the target: a `nutils.function.Argument` in `residual`.
- **residual** (*Integral*) –
- **inertia** (*Integral*) –
- **timestep** (*float*) – Initial time step, will scale up as residual decreases
- **lhs0** (*vector*) – Coefficient vector, starting point of the iterative procedure.
- **theta** (*float*) – Theta value (theta=1 for implicit Euler, theta=0.5 for Crank-Nicolson)
- **residual0** (*Integral*) – Optional additional residual component evaluated in previous timestep
- **constrain** (*boolean or float vector*) – Equal length to `lhs0`, masks the free vector entries as `False` (boolean) or `NaN` (float). In the remaining positions the values of `lhs0` are returned unchanged (boolean) or overruled by the values in `constrain` (float).
- **newtontol** (*float*) – Residual tolerance of individual timesteps
- **arguments** (*collections.abc.Mapping*) – Defines the values for `nutils.function.Argument` objects in `residual`. The `target` should not be present in `arguments`. Optional.

Yields *vector* – Coefficient vector for all timesteps after the initial condition.

```
nutils.solver.implicit euler(target, residual, inertia, timestep, lhs0, *, theta=1, target0='_thetamethod_target0', constrain=None, newtontol=1e-10, arguments=None, **newtonargs)
```

solve time dependent problem using the theta method

Parameters

- **target** (*str*) – Name of the target: a `nutils.function.Argument` in `residual`.
- **residual** (*Integral*) –
- **inertia** (*Integral*) –
- **timestep** (*float*) – Initial time step, will scale up as residual decreases

- **lhs0** (*vector*) – Coefficient vector, starting point of the iterative procedure.
- **theta** (*float*) – Theta value (theta=1 for implicit Euler, theta=0.5 for Crank-Nicolson)
- **residual0** (*Integral*) – Optional additional residual component evaluated in previous timestep
- **constrain** (*boolean or float vector*) – Equal length to lhs0, masks the free vector entries as False (boolean) or NaN (float). In the remaining positions the values of lhs0 are returned unchanged (boolean) or overruled by the values in *constrain* (float).
- **newtontol** (*float*) – Residual tolerance of individual timesteps
- **arguments** (*collections.abc.Mapping*) – Defines the values for *nutils.function.Argument* objects in *residual*. The target should not be present in arguments. Optional.

Yields *vector* – Coefficient vector for all timesteps after the initial condition.

```
nutils.solver.cranknicolson(target, residual, inertia, timestep, lhs0, *, theta=0.5, target0='_thetamethod_target0', constrain=None, newtontol=1e-10, arguments=None, **newtonargs)
```

solve time dependent problem using the theta method

Parameters

- **target** (*str*) – Name of the target: a *nutils.function.Argument* in residual.
- **residual** (*Integral*) –
- **inertia** (*Integral*) –
- **timestep** (*float*) – Initial time step, will scale up as residual decreases
- **lhs0** (*vector*) – Coefficient vector, starting point of the iterative procedure.
- **theta** (*float*) – Theta value (theta=1 for implicit Euler, theta=0.5 for Crank-Nicolson)
- **residual0** (*Integral*) – Optional additional residual component evaluated in previous timestep
- **constrain** (*boolean or float vector*) – Equal length to lhs0, masks the free vector entries as False (boolean) or NaN (float). In the remaining positions the values of lhs0 are returned unchanged (boolean) or overruled by the values in *constrain* (float).
- **newtontol** (*float*) – Residual tolerance of individual timesteps
- **arguments** (*collections.abc.Mapping*) – Defines the values for *nutils.function.Argument* objects in *residual*. The target should not be present in arguments. Optional.

Yields *vector* – Coefficient vector for all timesteps after the initial condition.

```
nutils.solver.optimize(target, functional, droptol=None, lhs0=None, constrain=None, newtontol=None, *, arguments=None)
```

find the minimizer of a given functional

Parameters

- **target** (*str*) – Name of the target: a *nutils.function.Argument* in residual.
- **functional** (*scalar Integral*) – The functional the should be minimized by varying target
- **droptol** (*float*) – Threshold for leaving entries in the return value at NaN if they do not contribute to the value of the functional.

- **lhs0** (*vector*) – Coefficient vector, starting point of the iterative procedure (if applicable).
- **constrain** (*boolean or float vector*) – Equal length to `lhs0`, masks the free vector entries as `False` (boolean) or `NaN` (float). In the remaining positions the values of `lhs0` are returned unchanged (boolean) or overruled by the values in `constrain` (float).
- **newtontol** (*float*) – Residual tolerance of Newton procedure (if applicable)

Yields *vector* – Coefficient vector corresponding to the functional optimum

1.3.17 Transform

The transform module.

1.3.18 Warnings

exception `nutils.warnings.NutilsWarning`

exception `nutils.warnings.NutilsDeprecationWarning`

class `nutils.warnings.via` (*print*)
context manager to set/reset `warnings.showwarning`

CHAPTER 2

Indices and tables

- [genindex](#)
- [modindex](#)
- [search](#)

c

nutils.cache, 29
nutils.cli, 29
nutils.config, 17
nutils.core, 17

e

nutils.element, 18
nutils.expression, 14

f

nutils.function, 8

l

nutils.log, 19

m

nutils.matrix, 21
nutils.mesh, 22

n

nutils.numeric, 24

p

nutils.parallel, 25
nutils.plot, 27

s

nutils.solver, 29

t

nutils.topology, 5
nutils.transform, 34

u

nutils.util, 26

w

nutils.warnings, 34

Symbols

`_context` (nutils.log.ContextLog attribute), 19
`_insert_anchors()` (nutils.log.HtmlInsertAnchor method), 20
`_print_item()` (nutils.log.ContextTreeLog method), 20
`_print_pop_context()` (nutils.log.ContextTreeLog method), 20
`_print_push_context()` (nutils.log.ContextTreeLog method), 20

A

Add (class in nutils.function), 10
 AmbiguousAlignmentError, 14
 ApplyTransforms (class in nutils.function), 10
 ArcCos (class in nutils.function), 10
 ArcSin (class in nutils.function), 10
 ArcTan (class in nutils.function), 10
 ArcTan2 (class in nutils.function), 10
 arg_shapes (nutils.function.Namespace attribute), 14
 Argument (class in nutils.function), 11
 Array (class in nutils.function), 9
 ArrayFromTuple (class in nutils.function), 11
 ArrayFunc (class in nutils.function), 9
 asciitree() (nutils.function.Evaluable method), 9
 assemble() (in module nutils.matrix), 22

B

BasePlot (class in nutils.plot), 27
 basis() (nutils.topology.HierarchicalTopology method), 7
 basis_bernstein() (nutils.topology.UnstructuredTopology method), 7
 basis_bubble() (nutils.topology.UnstructuredTopology method), 7
 basis_discont() (nutils.topology.MultipatchTopology method), 8
 basis_discont() (nutils.topology.StructuredLine method), 6
 basis_discont() (nutils.topology.StructuredTopology method), 7

basis_discont() (nutils.topology.UnstructuredTopology method), 7
 basis_lagrange() (nutils.topology.UnstructuredTopology method), 7
 basis_patch() (nutils.topology.MultipatchTopology method), 8
 basis_spline() (nutils.topology.MultipatchTopology method), 8
 basis_spline() (nutils.topology.StructuredLine method), 6
 basis_spline() (nutils.topology.StructuredTopology method), 6
 basis_std() (nutils.topology.StructuredLine method), 6
 basis_std() (nutils.topology.StructuredTopology method), 7
 basis_std() (nutils.topology.UnstructuredTopology method), 7
 BlockAdd (class in nutils.function), 10
 BndAxis (class in nutils.topology), 8
 boundaries (nutils.topology.MultipatchTopology.Patch attribute), 8
 bringforward() (in module nutils.function), 12
 build_boundarydata() (nutils.topology.MultipatchTopology static method), 8

C

Cache (class in nutils.function), 9
 call() (in module nutils.cli), 29
 celldataarray() (nutils.plot.VTKFile method), 28
 chain() (in module nutils.function), 12
 choose() (in module nutils.cli), 29
 close() (nutils.plot.PyPlot method), 27
 close() (nutils.plot.PyPlotVideo method), 28
 Concatenate (class in nutils.function), 10
 cond() (nutils.matrix.Matrix method), 21
 Cone (class in nutils.element), 18
 const (class in nutils.numeric), 25
 Constant (class in nutils.function), 9
 context() (nutils.log.ContextLog method), 19
 context() (nutils.log.Log method), 19

context() (nutils.log.TeeLog method), 21
 ContextLog (class in nutils.log), 19
 ContextTreeLog (class in nutils.log), 20
 contract() (in module nutils.numeric), 24
 copy_() (nutils.function.Namespace method), 14
 Cos (class in nutils.function), 10
 count() (in module nutils.log), 21
 cranknicolson() (in module nutils.solver), 33
 Cross (class in nutils.function), 10
 cspy() (nutils.plot.PyPlot method), 27

D

DataFile (class in nutils.plot), 28
 default_geometry (nutils.function.Namespace attribute), 14
 default_geometry_name (nutils.function.Namespace attribute), 14
 demo() (in module nutils.mesh), 24
 derivative() (in module nutils.function), 12
 DerivativeTargetBase (class in nutils.function), 11
 Determinant (class in nutils.function), 10
 Diagonalize (class in nutils.function), 11
 diagonalize() (in module nutils.numeric), 25
 DimAxis (class in nutils.topology), 8
 DofMap (class in nutils.function), 9
 Dot (class in nutils.function), 10
 dot() (in module nutils.numeric), 24

E

Eig (class in nutils.function), 11
 elem_eval() (nutils.topology.Topology method), 6
 elem_mean() (nutils.topology.Topology method), 6
 Element (class in nutils.element), 18
 Elemwise (class in nutils.function), 11
 EmptyReference (class in nutils.element), 18
 EmptyTopology (class in nutils.topology), 6
 enumerate() (in module nutils.log), 21
 Equal (class in nutils.function), 10
 evalf() (nutils.function.Tuple method), 9
 Evaluable (class in nutils.function), 9
 EvaluationError, 9
 Exp (class in nutils.function), 10
 ExpressionSyntaxError, 14
 ext() (in module nutils.numeric), 25

F

FileCache (class in nutils.cache), 29
 Find (class in nutils.function), 11
 find() (in module nutils.function), 13
 FindTransform (class in nutils.function), 12
 fromfunc() (in module nutils.mesh), 24

G

Get (class in nutils.function), 10

get() (in module nutils.numeric), 24
 get_quad_bem_ischeme() (nutils.element.NeighborhoodTensorReference method), 19
 get_tri_bem_ischeme() (nutils.element.NeighborhoodTensorReference method), 18
 getscheme() (nutils.element.MosaicReference method), 19
 getscheme() (nutils.element.WithChildrenReference method), 19
 getscheme_gauss() (nutils.element.TetrahedronReference method), 18
 getscheme_gauss() (nutils.element.TriangleReference method), 18
 getscheme_singular() (nutils.element.NeighborhoodTensorReference method), 19
 gmsh() (in module nutils.mesh), 24
 graphviz() (nutils.function.Evaluable method), 9
 Greater (class in nutils.function), 10
 griddata() (nutils.plot.PyPlot method), 27
 Guard (class in nutils.function), 11

H

HierarchicalTopology (class in nutils.topology), 7
 HtmlInsertAnchor (class in nutils.log), 20
 HtmlLog (class in nutils.log), 20

I

i (nutils.topology.BndAxis attribute), 8
 i (nutils.topology.DimAxis attribute), 8
 ibound (nutils.topology.BndAxis attribute), 8
 impliciteuler() (in module nutils.solver), 32
 IndentLog (class in nutils.log), 21
 Inflate (class in nutils.function), 11
 InsertAxis (class in nutils.function), 9
 Int (class in nutils.function), 11
 Integral (class in nutils.solver), 30
 integral() (nutils.topology.Topology method), 6
 integrate() (nutils.topology.Topology method), 6
 Interpolate (class in nutils.function), 10
 Inverse (class in nutils.function), 10
 isperiodic (nutils.topology.DimAxis attribute), 8
 iter() (in module nutils.log), 21
 ix() (in module nutils.numeric), 25

J

j (nutils.topology.BndAxis attribute), 8
 j (nutils.topology.DimAxis attribute), 8

L

Less (class in nutils.function), 10

LinearFrom (class in nutils.function), 10
 LineReference (class in nutils.element), 18
 listoutdir() (in module nutils.core), 17
 LocalCoords (class in nutils.function), 12
 localgradient() (in module nutils.function), 12
 LocateError, 6
 Log (class in nutils.function), 10
 Log (class in nutils.log), 19

M

Mask (class in nutils.function), 12
 matmat() (in module nutils.function), 12
 Matrix (class in nutils.matrix), 21
 Maximum (class in nutils.function), 10
 mesh() (nutils.plot.PyPlot method), 27
 meshgrid() (in module nutils.numeric), 25
 Minimum (class in nutils.function), 10
 Mod (class in nutils.function), 10
 ModelError, 30
 MosaicReference (class in nutils.element), 19
 multipatch() (in module nutils.mesh), 22
 MultipatchTopology (class in nutils.topology), 7
 MultipatchTopology.Patch (class in nutils.topology), 8
 Multiply (class in nutils.function), 10

N

Namespace (class in nutils.function), 13
 NanVec (class in nutils.util), 26
 NeighborhoodTensorReference (class in nutils.element), 18
 newton (class in nutils.solver), 31
 Normal (class in nutils.function), 9
 normalize() (in module nutils.numeric), 25
 normdim() (in module nutils.numeric), 24
 NumpyMatrix (class in nutils.matrix), 22
 nutils.cache (module), 29
 nutils.cli (module), 29
 nutils.config (module), 17
 nutils.core (module), 17
 nutils.element (module), 18
 nutils.expression (module), 14
 nutils.function (module), 8
 nutils.log (module), 19
 nutils.matrix (module), 21
 nutils.mesh (module), 22
 nutils.numeric (module), 24
 nutils.parallel (module), 25
 nutils.plot (module), 27
 nutils.solver (module), 29
 nutils.topology (module), 5
 nutils.transform (module), 34
 nutils.util (module), 26
 nutils.warnings (module), 34
 NutilsDeprecationWarning, 34

NutilsWarning, 34

O

obj2str() (in module nutils.util), 26
 open_in_outdir() (in module nutils.core), 17
 OppositeTopology (class in nutils.topology), 6
 optimize() (in module nutils.solver), 33
 OrientedGroupsTopology (class in nutils.topology), 7
 outer() (in module nutils.function), 12
 overlapping() (in module nutils.numeric), 24
 OwnChildReference (class in nutils.element), 19

P

pariter() (in module nutils.parallel), 25
 parmap() (in module nutils.parallel), 25
 parse() (in module nutils.expression), 14
 parsecons() (in module nutils.matrix), 22
 permutation_transforms (nutils.element.WithChildrenReference attribute), 19
 Point (class in nutils.topology), 6
 pointdataarray() (nutils.plot.VTKFile method), 28
 PointReference (class in nutils.element), 18
 Points (class in nutils.function), 9
 Pointwise (class in nutils.function), 10
 polycol() (nutils.plot.PyPlot method), 27
 polyfunc() (in module nutils.function), 12
 Polyval (class in nutils.function), 12
 PopHead (class in nutils.function), 9
 Power (class in nutils.function), 10
 Product (class in nutils.function), 10
 product() (in module nutils.util), 26
 ProductTopology (class in nutils.topology), 7
 project() (nutils.topology.Topology method), 6
 projection() (nutils.topology.Topology method), 6
 Promote (class in nutils.function), 9
 pseudotime (class in nutils.solver), 31
 PyPlot (class in nutils.plot), 27
 PyPlotVideo (class in nutils.plot), 27

R

Range (class in nutils.function), 12
 range() (in module nutils.log), 21
 Ravel (class in nutils.function), 12
 rectangle() (nutils.plot.PyPlot method), 27
 rectilinear() (in module nutils.mesh), 22
 rectilineargrid() (nutils.plot.VTKFile method), 28
 Reference (class in nutils.element), 18
 refine() (nutils.topology.ProductTopology method), 7
 refine() (nutils.topology.Topology method), 6
 refined (nutils.topology.StructuredTopology attribute), 7
 refined_by() (nutils.topology.Topology method), 6
 RefinedTopology (class in nutils.topology), 7
 replace() (in module nutils.cache), 29

replace_arguments() (in module nutils.function), 13
 res() (nutils.matrix.Matrix method), 21
 RevolutionReference (class in nutils.element), 18
 RevolutionTopology (class in nutils.topology), 7
 RichOutputLog (class in nutils.log), 20
 rowsupp() (nutils.matrix.ScipyMatrix method), 22
 run() (in module nutils.cli), 29

S

Sampled (class in nutils.function), 11
 save() (nutils.plot.PyPlot method), 27
 saveframe() (nutils.plot.PyPlotVideo method), 28
 ScipyMatrix (class in nutils.matrix), 22
 segments() (nutils.plot.PyPlot method), 27
 SelectBifurcation (class in nutils.function), 9
 SelectChain (class in nutils.function), 9
 shempty() (in module nutils.parallel), 25
 shzeros() (in module nutils.parallel), 25
 side (nutils.topology.BndAxis attribute), 8
 Sign (class in nutils.function), 11
 SimplexReference (class in nutils.element), 18
 Sin (class in nutils.function), 10
 single_or_multiple() (in module nutils.util), 26
 slope_marker() (nutils.plot.PyPlot method), 27
 slope_trend() (nutils.plot.PyPlot method), 27
 slope_triangle() (nutils.plot.PyPlot method), 27
 solve() (in module nutils.solver), 30
 solve() (nutils.matrix.NumpyMatrix method), 22
 solve() (nutils.matrix.ScipyMatrix method), 22
 solve_linear() (in module nutils.solver), 30
 stackstr() (nutils.function.Evaluable method), 9
 StdoutLog (class in nutils.log), 20
 StructuredLine (class in nutils.topology), 6
 StructuredTopology (class in nutils.topology), 6
 subset() (nutils.topology.Topology method), 6
 SubsetTopology (class in nutils.topology), 7
 Sum (class in nutils.function), 10
 sum() (in module nutils.util), 26

T

TailOfTransform (class in nutils.function), 9
 Take (class in nutils.function), 10
 TakeDiag (class in nutils.function), 10
 Tan (class in nutils.function), 10
 TeeLog (class in nutils.log), 21
 TensorReference (class in nutils.element), 18
 TetrahedronReference (class in nutils.element), 18
 thetamethod() (in module nutils.solver), 32
 title() (in module nutils.log), 21
 topo (nutils.topology.MultipatchTopology.Patch attribute), 8
 Topology (class in nutils.topology), 5
 TransformChain (class in nutils.function), 9
 Transpose (class in nutils.function), 9

TriangleReference (class in nutils.element), 18
 TrigNormal (class in nutils.function), 11
 TrigTangent (class in nutils.function), 11
 trim() (nutils.element.EmptyReference method), 18
 trim() (nutils.element.Reference method), 18
 trim() (nutils.topology.Topology method), 6
 TrimmedTopologyBoundaryItem (class in nutils.topology), 7
 TrimmedTopologyItem (class in nutils.topology), 7
 Tuple (class in nutils.function), 9

U

UnionTopology (class in nutils.topology), 7
 Unravel (class in nutils.function), 12
 unstructuredgrid() (nutils.plot.VTKFile method), 28
 UnstructuredTopology (class in nutils.topology), 7

V

verts (nutils.topology.MultipatchTopology.Patch attribute), 8
 via (class in nutils.warnings), 34
 VTKFile (class in nutils.plot), 28

W

WithChildrenReference (class in nutils.element), 19
 WithGroupsTopology (class in nutils.topology), 6
 withsolve() (in module nutils.solver), 30
 Wrapper (class in nutils.cache), 29
 WrapperCache (class in nutils.cache), 29
 WrapperDummyCache (class in nutils.cache), 29
 write() (nutils.log.ContextTreeLog method), 20
 write() (nutils.log.HtmlLog method), 21
 write() (nutils.log.Log method), 19
 write() (nutils.log.StdoutLog method), 20
 write() (nutils.log.TeeLog method), 21
 write_post_mortem() (nutils.log.HtmlLog method), 21
 writetvu() (in module nutils.plot), 29

Z

Zeros (class in nutils.function), 11
 zip() (in module nutils.log), 21