nutils Documentation

Release 4.0a0

Evalf

Jul 21, 2018

Contents

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.

CHAPTER 1

Contents

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

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 numer 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 nutils.topology.**Topology**(**args*, ***kwargs*)
topology base class

edict
transform -> ielement mapping

sample(*self*, *ischeme*, *degree*)
Create sample.

elem_eval(*self*, *funcs*, *ischeme*, *separate=False*, *geometry=None*, *asfunction=False*, ***, *edit=None*, *title='elem_eval'*, ***kwargs*)
element-wise evaluation

integrate_elementwise(*self*, *funcs*, ***, *asfunction=False*, ***kwargs*)
element-wise integration

integrate(*self*, *funcs*, *ischeme='gauss'*, *degree=None*, *geometry=None*, *edit=None*, ***, *arguments=None*, *title='integrate'*)
integrate functions

integral(*self*, *func*, *ischeme='gauss'*, *degree=None*, *geometry=None*, *edit=None*)

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

project(*self*, *fun*, *onto*, *geometry*, *ischeme='gauss'*, *degree=None*, *droptol=1e-12*, *exact_boundaries=False*, *constrain=None*, *verify=None*, *ptype='lsqr'*, *edit=None*, ***, *arguments=None*, ***solverargs*)
L2 projection of function onto function space

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

refine(*self*, *n*)
refine entire topology n times

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

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

locate(*self*, *geom*, *coords*, *ischeme='vertex'*, *scale=1*, *tol=1e-12*, *eps=0*, *maxiter=100*, ***, *arguments=None*)
Create a sample based on physical coordinates.

In a finite element application, functions are commonly evaluated in points that are defined on the topology. The reverse, finding a point on the topology based on a function value, is often a nonlinear process and as such involves Newton iterations. The `locate` function facilitates this search process and produces a *nutils.sample.Sample* instance that can be used for the subsequent evaluation of any function in the given physical points.

Example:

```
>>> from . import mesh
>>> domain, geom = mesh.rectilinear([2,1])
>>> sample = domain.locate(geom, [[1.5, .5]])
>>> sample.eval(geom).tolist()
[[1.5, 0.5]]
```

Locate has a long list of arguments that can be used to steer the nonlinear search process, but the default values should be fine for reasonably standard situations.

Parameters

- **geom** (1-dimensional *nutils.function.Array*) – Geometry function of length ndims.
- **coords** (2-dimensional float array) – Array of coordinates with ndims columns.
- **ischeme** (str (default: "vertex")) – Sample points used to determine bounding boxes.
- **scale** (float (default: 1)) – Bounding box amplification factor, useful when element shapes are distorted. Setting this to >1 can increase computational effort but is otherwise harmless.
- **tol** (float (default: 1e-12)) – Newton tolerance.
- **eps** (float (default: 0)) – Epsilon radius around element within which a point is considered to be inside.
- **maxiter** (int (default: 100)) – Maximum allowed number of Newton iterations.
- **arguments** (dict (default: None)) – Arguments for function evaluation.

Returns located

Return type *nutils.sample.Sample*

basis_discont(*self*, *degree*)
discontinuous shape functions

basis_lagrange(*self*, *degree*)
lagrange shape functions

basis_bernstein(*self*, *degree*)
bernstein shape functions

basis_std(*self*, *degree*)
bernstein shape functions

exception nutils.topology.**LocateError**

class nutils.topology.**WithGroupsTopology**(**args*, ***kwargs*)
item topology

edict
transform -> ielement mapping

class nutils.topology.**OppositeTopology**(**args*, ***kwargs*)
opposite topology

class nutils.topology.**EmptyTopology**(**args*, ***kwargs*)
empty topology

class nutils.topology.**Point**(**args*, ***kwargs*)
point

class nutils.topology.**StructuredLine**(**args*, ***kwargs*)
structured topology

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

basis_discont(*self*, *degree*)
discontinuous shape functions

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

class nutils.topology.**Axis**(**args*, ***kwargs*)

class nutils.topology.**DimAxis**(**args*, ***kwargs*)

class nutils.topology.**BndAxis**(**args*, ***kwargs*)

class nutils.topology.**StructuredTopology**(**args*, ***kwargs*)
structured topology

boundary

interfaces

basis_spline(*self*, *degree*, *removedofs=None*, ***kwargs*)
spline basis

basis_discont(*self*, *degree*)
discontinuous shape functions

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

refined
refine non-uniformly

class nutils.topology.**UnstructuredTopology**(**args*, ***kwargs*)
unstructured topology

class nutils.topology.**ConnectedTopology**(**args*, ***kwargs*)
unstructured topology with connectivity

class nutils.topology.**SimplexTopology**(**args*, ***kwargs*)
simpex topology

basis_bubble(*self*)
bubble from vertices

class nutils.topology.**UnionTopology**(**args*, ***kwargs*)
grouped topology

class nutils.topology.**SubsetTopology**(**args*, ***kwargs*)
trimmed

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

class nutils.topology.**RefinedTopology**(**args*, ***kwargs*)
refinement

class nutils.topology.**HierarchicalTopology**(**args*, ***kwargs*)
collection of nested topology elments

boundary
boundary elements

interfaces

basis(*self*, *name*, **args*, *truncation_tolerance=1e-15*, ***kwargs*)
Create hierarchical basis.

A hierarchical basis is constructed from bases on different levels of uniform refinement. Two different types of hierarchical bases are supported:

1. Classical – Starting from the set of all basis functions originating from all levels of uniform refinement, only those basis functions are selected for which at least one supporting element is part of the hierarchical topology.

2. Truncated – Like classical, but with basis functions modified such that the area of support is reduced. An additional effect of this procedure is that it restores partition of unity. The spanned function space remains unchanged.

Truncation is based on linear combinations of basis functions, where fine level basis functions are used to reduce the support of coarser level basis functions. See Giannelli et al. 2012 for more information on truncated hierarchical refinement.

Parameters

- **name** (str) – Type of basis function as provided by the base topology, with prefix h- (h-std, h-spline) for a classical hierarchical basis and prefix th- (th-std, th-spline) for a truncated hierarchical basis. For backwards compatibility the h- prefix is optional, but omitting it triggers a deprecation warning as this behaviour will be removed in future.
- **truncation_tolerance** (float (default 1e-15)) – In order to benefit from the extra sparsity resulting from truncation, vanishing polynomials need to be actively identified and removed from the basis. The trunctation_tolerance offers control over this threshold.

Returns basis

Return type *nutils.function.Array*

class nutils.topology.**ProductTopology**(**args*, ***kwargs*)
product topology

refine(*self*, *n*)
refine entire topology n times

class nutils.topology.**RevolutionTopology**(**args*, ***kwargs*)
topology consisting of a single revolution element

class nutils.topology.**PatchBoundary**(**args*, ***kwargs*)

class nutils.topology.**Patch**(**args*, ***kwargs*)

class nutils.topology.**MultipatchTopology**(**args*, ***kwargs*)
multipatch topology

static build_boundarydata(*connectivity*)
build boundary data based on connectivity

basis_spline(*self*, *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(*self*, *degree*)
discontinuous shape functions

basis_patch(*self*)
degree zero patchwise discontinuous basis

boundary

interfaces
Return a topology with all element interfaces. The patch interfaces are accessible via the group `'interpatch'` and the interfaces *inside* a patch via `'intrapatch'`.

refined
refine

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 notabe class of *Array* objects map onto the space of Numpy arrays of predefined dimension and shape. Most functions used in nutils applicatons 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*, ***kwargs*)
Base class

dependencies
collection of all function arguments

ordereddeps
collection of all function arguments such that the arguments to dependencies[i] can be found in dependencies[:i]

dependencytree
lookup table of function arguments into ordereddeps, such that ordereddeps[i].__args[j] == ordereddeps[dependencytree[i][j]], and self.__args[j] == ordereddeps[dependencytree[-1][j]]

asciitree(*self*, *seen=None*)
string representation

graphviz(*self*)
create function graph

stackstr(*self*, *nlines=-1*)
print stack

exception nutils.function.**EvaluationError**(*etype*, *evalue*, *evaluable*, *values*)
evaluation error

class nutils.function.**Cache**(**args*, ***kwargs*)

class nutils.function.**Points**(**args*, ***kwargs*)

class nutils.function.**Tuple**(**args*, ***kwargs*)

evalf(*self*, **items*)
evaluate

class nutils.function.**TransformChain**(**args*, ***kwargs*)
Chain of affine transformations.

Evaluates to a tuple of *nutils.transform.TransformItem* objects.

class nutils.function.**SelectChain**(**args*, ***kwargs*)

class nutils.function.**PopHead**(**args*, ***kwargs*)

class nutils.function.**SelectBifurcation**(**args*, ***kwargs*)

class nutils.function.**Promote**(**args*, ***kwargs*)

class nutils.function.**TailOfTransform**(**args*, ***kwargs*)

class nutils.function.**Array**(**args*, ***kwargs*)
array function

class nutils.function.**Normal**(**args*, ***kwargs*)
normal

class nutils.function.**Constant**(**args*, ***kwargs*)

class nutils.function.**DofMap**(**args*, ***kwargs*)

class nutils.function.**InsertAxis**(**args*, ***kwargs*)

class nutils.function.**Transpose**(**args*, ***kwargs*)

class nutils.function.**Get**(**args*, ***kwargs*)

class nutils.function.**Product**(**args*, ***kwargs*)

class nutils.function.**ApplyTransforms**(**args*, ***kwargs*)

class nutils.function.**LinearFrom**(**args*, ***kwargs*)

class nutils.function.**Inverse**(**args*, ***kwargs*)
Matrix inverse of `func` over the last two axes. All other axes are treated element-wise.

class nutils.function.**Concatenate**(**args*, ***kwargs*)

class nutils.function.**Interpolate**(**args*, ***kwargs*)
interpolate uniformly spaced data; stepwise for now

class nutils.function.**Cross**(**args*, ***kwargs*)

class nutils.function.**Determinant**(**args*, ***kwargs*)

class nutils.function.**Multiply**(**args*, ***kwargs*)

class nutils.function.**Add**(**args*, ***kwargs*)

class nutils.function.**BlockAdd**(**args*, ***kwargs*)
block addition (used for DG)

class nutils.function.**Dot**(**args*, ***kwargs*)

class nutils.function.**Sum**(**args*, ***kwargs*)

class nutils.function.**TakeDiag**(**args*, ***kwargs*)

class nutils.function.**Take**(**args*, ***kwargs*)

class nutils.function.**Power**(**args*, ***kwargs*)

class nutils.function.**Pointwise**(**args*, ***kwargs*)

class nutils.function.**Cos**(**args*, ***kwargs*)
Cosine, element-wise.

class nutils.function.**Sin**(**args*, ***kwargs*)
Sine, element-wise.

class nutils.function.**Tan**(**args*, ***kwargs*)
Tangent, element-wise.

class nutils.function.**ArcSin**(**args*, ***kwargs*)
Inverse sine, element-wise.

class nutils.function.**ArcCos**(**args*, ***kwargs*)
Inverse cosine, element-wise.

class nutils.function.**ArcTan**(**args*, ***kwargs*)
Inverse tangent, element-wise.

class nutils.function.**Exp**(**args*, ***kwargs*)

class nutils.function.**Log**(**args*, ***kwargs*)

class nutils.function.**Mod**(**args*, ***kwargs*)

class nutils.function.**ArcTan2**(**args*, ***kwargs*)

class nutils.function.**Greater**(**args*, ***kwargs*)

class nutils.function.**Equal**(**args*, ***kwargs*)

class nutils.function.**Less**(**args*, ***kwargs*)

class nutils.function.**Minimum**(**args*, ***kwargs*)

class nutils.function.**Maximum**(**args*, ***kwargs*)

class nutils.function.**Int**(**args*, ***kwargs*)

class nutils.function.**Sign**(**args*, ***kwargs*)

class nutils.function.**OldSampled**(**args*, ***kwargs*)
sampled

class nutils.function.**Sampled**(**args*, ***kwargs*)
Convert sampled data to evaluable array.

Using the result of *nutils.sample.Sample.eval()*, create an evaluable array that upon evaluation recovers the original function in the set of points matching the original sampling.

Parameters

- **sample** (*nutils.sample.Sample*) – The set of points that the data was sampled on.
- **array** – The sampled data.
- **trans** (*TransformChain* (optional)) – The transformation chain that is used to locate the sample points.

class nutils.function.**Elemwise**(**args*, ***kwargs*)

class nutils.function.**Eig**(**args*, ***kwargs*)

class nutils.function.**ArrayFromTuple**(**args*, ***kwargs*)

class nutils.function.**Zeros**(**args*, ***kwargs*)
zero

class nutils.function.**Inflate**(**args*, ***kwargs*)

class nutils.function.**Diagonalize**(**args*, ***kwargs*)

class nutils.function.**Guard**(**args*, ***kwargs*)
bar all simplifications

class nutils.function.**TrigNormal**(**args*, ***kwargs*)
cos, sin

class nutils.function.**TrigTangent**(**args*, ***kwargs*)
-sin, cos

class nutils.function.**Find**(**args*, ***kwargs*)
indices of boolean index vector

class nutils.function.**DerivativeTargetBase**(**args*, ***kwargs*)
base class for derivative targets

class nutils.function.**Argument**(**args*, ***kwargs*)
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 cooardinates 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)
>>> 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**(**args*, ***kwargs*)
local coords derivative target

class nutils.function.**Ravel**(**args*, ***kwargs*)

class nutils.function.**Unravel**(**args*, ***kwargs*)

class nutils.function.**Mask**(**args*, ***kwargs*)

class nutils.function.**FindTransform**(**args*, ***kwargs*)

class nutils.function.**Range**(**args*, ***kwargs*)

class nutils.function.**Polyval**(**args*, ***kwargs*)
Computes the k-dimensional array

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

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.**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`

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 `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 *Array*s as values) – *Argument*s 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 *nutils.expression.parse()*. Default: `'x'`.

arg_shapes
view of `dict` – 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
nutils.function.Array – The default geometry, shorthand for `getattr(ns, ns.default_geometry_name)`.

copy_(*self*, *, *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 nutils.expression.**ExpressionSyntaxError**

exception nutils.expression.**AmbiguousAlignmentError**

nutils.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_iji`. It is invalid to specify an index more than twice. The following names cannot be used as variables: `n`, δ, `$`. 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_i c_j` is a valid, provided that the lengths of the axes with the same indices match, but `a_ij + b_i` 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 `a -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 overriden 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 sufrace 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 of semicolon. Example: `a_i,altgeom_j` denotes the gradient of `a_i` to `altgeom` and the gradient axis has index `j`. Futhermore, 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 - λ δ_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 decuded 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)
('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 is deprecated.

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.

The following configuration properties are used in Nutils.

`nutils.config.`**`nprocs`**
: Controls the number of processes to use for computing integrals (*nutils.topology.Topology.integrate()*) and a few other expensive and parallelizable functions.

  Defaults to `1`.

`nutils.config.`**`verbose`**
: Controls the level of verbosity of loggers. Log entries with a level higher than *verbose* are omitted. The levels are `1`: error, `2`: warning, `3`: user, `4`: info and `5`: debug.

  Defaults to `4`: info.

`nutils.config.`**`dot`**
: If `True`, *nutils.topology.Topology.elem_eval()* and *nutils.topology.Topology.integrate()* log a visualization of the function tree that is being evaluated or integrated.

  Defaults to `False`.

`nutils.config.`**`imagetype`**
: A comma-separated list of (file extensions of) image types. *nutils.plot.PyPlot* generates figures for all listed types.

  Defaults to `'png'`.

Deprecated since version 4.0: Instead of *nutils.plot.PyPlot* the new *nutils.export.mplfigure()* helper should be used. The latter requires a filename with extension and ignores the *imagetype* attribute.

The following properties are only used in *nutils.cli.run()* and *nutils.cli.choose()*.

nutils.config.**outrootdir**
Defines the root directory for general output.

Defaults to `'~/public_html'`

nutils.config.**outdir**
Defines the output directory for the HTML log (*nutils.log.HtmlLog*) and plots. Relative paths are relative with respect to the current working directory (see `os.getcwd()`).

Defaults to `'<outrootdir>/<scriptname>/<YY/MM/DD/HH-MM-SS>'`

nutils.config.**cache**
Controls on-disk caching. If `True`, functions decorated with *nutils.cache.function()* (e.g. *nutils.topology.Topology.integrate()*) and subclasses of *nutils.cache.Recursion* (e.g. *nutils.solver.thetamethod*) are automatically cached.

Defaults to `False`.

nutils.config.**cachedir**
Defines the location of the on-disk cache (see *cache*) relative to `<outrootdir>/<scriptname>`.

Defaults to `'cache'`.

nutils.config.**symlink**
If not empty, the symlinks `'<outrootdir>/<symlink>'` and `'<outrootdir>/<scriptname>/<symlink>'` will be created, both pointing to `'<outrootdir>/<scriptname>/<YY/MM/DD/HH-MM-SS>'`.

Defaults to `''`.

nutils.config.**richoutput**
Controls whether or not the console logger should output rich text (*nutils.log.RichOutputLog*) or plain text (*nutils.log.StdoutLog*).

Defaults to `True` if `sys.stdout` is attached to a terminal (i.e. `sys.stdout.isatty()` returns true), otherwise `False`.

nutils.config.**htmloutput**
If `True` the HTML logger (*nutils.log.HtmlLog*) is enabled and written to `'<outrootdir>/<scriptname>/<YY/MM/DD/HH-MM-SS>/log.html'`

Defaults to `True`.

nutils.config.**pdb**
If `True` the debugger will be invoked when an exception reaches *nutils.cli.run()* or *nutils.cli.choose()*.

Defaults to `False`.

nutils.config.**matrix**
A comma-separated list of matrix backends. The first one available is activated. The names — the case is irrelevant – correspond to subclasses of *nutils.matrix.Backend*. Use `nutils.matrix.Backend.__subclasses__()` to list the available backends.

Defauls to `'mkl,scipy,numpy'`.

1.3.6 Element

The element module defines reference elements such as the *LineReference* and *TriangleReference*, but also more exotic objects like the *MosaicReference*. A set of (interconnected) elements together form a *nutils.topology.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.**Reference**(**args*, ***kwargs*)
reference element

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

class nutils.element.**EmptyLike**(**args*, ***kwargs*)
inverse reference element

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

class nutils.element.**RevolutionReference**(**args*, ***kwargs*)
modify gauss integration to always return a single point

class nutils.element.**SimplexReference**(**args*, ***kwargs*)
simplex reference

class nutils.element.**PointReference**(**args*, ***kwargs*)
0D simplex

class nutils.element.**LineReference**(**args*, ***kwargs*)
1D simplex

class nutils.element.**TriangleReference**(**args*, ***kwargs*)
2D simplex

class nutils.element.**TetrahedronReference**(**args*, ***kwargs*)
3D simplex

class nutils.element.**TensorReference**(**args*, ***kwargs*)
tensor reference

class nutils.element.**Cone**(**args*, ***kwargs*)
cone

class nutils.element.**OwnChildReference**(**args*, ***kwargs*)
forward self as child

class nutils.element.**WithChildrenReference**(**args*, ***kwargs*)
base reference with explicit children

class nutils.element.**MosaicReference**(**args*, ***kwargs*)
triangulation

class nutils.element.**Element**(**args*, ***kwargs*)
element class

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(*self*, *title*, *mayskip=False*)
Return a context manager that adds a contextual layer named `title`.

Note: This function is abstract.

---

write(*self*, *level*, *text*)
Write `text` with log level `level` to the log.

Note: This function is abstract.

---

open(*self*, *filename*, *mode*, *level*, *exists*)
Create file object.

class nutils.log.**DataLog**(*outdir*)
Output only data.

context(*self*, *title*, *mayskip=False*)
Return a context manager that adds a contextual layer named `title`.

Note: This function is abstract.

---

write(*self*, *level*, *text*)
Write `text` with log level `level` to the log.

Note: This function is abstract.

---

open(*self*, *filename*, *mode*, *level*, *exists*)
Create file object.

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 (`str`s) that are currently active.

context(*self*, *title*, *mayskip=False*)
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(*self*, *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(*self*)
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(*self*, *level*, *text*)
Add an item to the log.

Note: This function is abstract.

write(*self*, *level*, *text*, ***kwargs*)
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(*self*, *level*, *text*, *endl=True*)
Write `text` with log level `level` to the log.

Note: This function is abstract.

open(*self*, *filename*, *mode*, *level*, *exists*)
Create file object.

class nutils.log.**RichOutputLog**(*stream=None*, ***, *progressinterval=None*)
Output rich (colored,unicode) text to stream.

class nutils.log.**HtmlLog**(*outdir*, ***, *title='nutils'*, *scriptname=None*, *funcname=None*, *funcargs=None*)
Output html nested lists.

write_post_mortem(*self*, *etype*, *value*, *tb*)
write exception nfo to html log

open(*self*, *filename*, *mode*, *level*, *exists*)
Create file object.

class nutils.log.**IndentLog**(*outdir*, ***, *progressinterval=None*)
Output indented html snippets.

open(*self*, *filename*, *mode*, *level*, *exists*)
Create file object.

class nutils.log.**TeeLog**(**logs*)
Simultaneously interface multiple logs

context(*self*, *title*, *mayskip=False*)
Return a context manager that adds a contextual layer named `title`.

Note: This function is abstract.

write (*self*, *level*, *text*)
Write `text` with log level `level` to the log.

Note: This function is abstract.

open (*self*, *filename*, *mode*, *level*, *exists*)
Create file object.

class nutils.log.**RecordLog**
Log object that records log messages. All messages are forwarded to the log that whas active before activating this log (e.g. by `with RecordLog() as record:`). The recorded messages can be replayed to the log that's currently active by *replay()*.

Typical usage is caching expensive operations:

```
# compute
with RecordLog() as record:
  result = compute_something_expensive()
raw = pickle.dumps((record, result))
# reuse
record, result = pickle.loads(raw)
record.replay()
```

Note: Instead of using *RecordLog* and `pickle` manually, as in above example, we advice to use *nutils.cache.FileCache* instead.

Note: Exceptions raised while in a *Log.context()* are not recorded.

Note: Messages dispatched from forks (e.g. inside *nutils.parallel.pariter()*) are not recorded.

context (*self*, *title*, *mayskip=False*)
Return a context manager that adds a contextual layer named `title`.

Note: This function is abstract.

write (*self*, *level*, *text*)
Write `text` with log level `level` to the log.

Note: This function is abstract.

open (*self*, *filename*, *mode*, *level*, *exists*)
Create file object.

replay (*self*)
Replay this recorded log in the log that's currently active.

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.

nutils.log.**open**(*filename*, *mode*, ***, *level='user'*, *exists='rename'*)
Open file in logger-controlled directory.

Parameters

- **filename** (str) –
- **mode** (str) – Should be either 'w' (text) or 'wb' (binary data).
- **level** (str) – Log level in which the filename is displayed. Default: 'user'.
- **exists** (str) – Determines how existence of filename in the output directory should be handled. Valid values are:
  - 'overwrite': open the file and remove current contents.
  - 'rename': change the filename by adding the smallest positive suffix n for which filename-n.ext does not exist.
  - 'skip': return a dummy file object, which tests as False to allow content creation to be skipped altogether.

1.3.8 Matrix

The matrix module defines an abstract *Matrix* object and several implementations. Matrix objects support basic addition and subtraction operations and provide a consistent insterface for solving linear systems. Matrices can be converted into other forms suitable for external processing via the export method.

class nutils.matrix.**Backend**
backend base class

assemble(*self*, *data*, *index*, *shape*)
Assemble a (sparse) tensor based on index-value pairs.

Note: This function is abstract.

class nutils.matrix.**Matrix**(*shape*)
matrix base class

T
transpose matrix

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

solve (*self*, *rhs=None*, *, *lhs0=None*, *constrain=None*, *rconstrain=None*, ***solverargs*)
Solve system given right hand side vector and/or constraints.

Parameters

- **rhs** (`float` vector or `None`) – Right hand side vector. *None* implies all zeros.
- **lhs0** (class:*float* vector or `None`) – Initial values. *None* implies all zeros.
- **constrain** (`float` or `bool` array, or `None`) – Column constraints. For float values, a number signifies a constraint, NaN signifies a free dof. For boolean, a True value signifies a constraint to the value in *lhs0*, a False value signifies a free dof. *None* implies no constraints.
- **rconstrain** (`bool` array or `None`) – Row constrains. A True value signifies a constrains, a False value a free dof. *None* implies that the constraints follow those defined in *constrain* (by implication the matrix must be square).

Returns Left hand side vector.

Return type `numpy.ndarray`

submatrix (*self*, *rows*, *cols*)
Create submatrix from selected rows, columns.

Parameters

- **rows** (`bool`/`int` array selecting rows for keeping) –
- **cols** (`bool`/`int` array selecting columns for keeping) –

Returns Matrix instance of reduced dimensions

Return type *Matrix*

export (*self*, *form*)
Export matrix data to any of supported forms.

Parameters **form** (`str`) –

- “dense” : return matrix as a single dense array
- ”csr” : return matrix as 3-tuple of (data, indices, indptr)
- ”coo” : return matrix as 2-tuple of (data, (row, col))

nutils.matrix.**preparesolvearguments** (*wrapped*)
Make rhs optional, add lhs0, constrain, rconstrain arguments.

See Matrix.solve.

class nutils.matrix.**Numpy**
matrix backend based on numpy array

assemble (*self*, *data*, *index*, *shape*)
Assemble a (sparse) tensor based on index-value pairs.

Note: This function is abstract.

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

T
transpose matrix

export (*self*, *form*)
Export matrix data to any of supported forms.

Parameters **form** (str) –

- “dense” : return matrix as a single dense array
- ”csr” : return matrix as 3-tuple of (data, indices, indptr)
- ”coo” : return matrix as 2-tuple of (data, (row, col))

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

submatrix (*self*, *rows*, *cols*)
Create submatrix from selected rows, columns.

Parameters

- **rows** (bool/int array selecting rows for keeping) –
- **cols** (bool/int array selecting columns for keeping) –

Returns Matrix instance of reduced dimensions

Return type *Matrix*

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
```

(continues on next page)

(continued from previous page)

```
patches=[[[0,3],[1,4]], [[1,4],[2,5]]]
```

The geometry is specified by the `patchverts` argument: a `numpy.ndarray`-like with shape `(nverts, ngeomdims)` 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],
```

(continues on next page)

(continued from previous page)

```
      [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))),
    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='gmsh'*)
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` or `None`) – Name of parsed topology, defaults to 'gmsh'.

Returns

- **topo** (*nutils.topology.SimplexTopology*) – Topology of parsed Gmsh file.
- **geom** (*nutils.function.Array*) – Isoparametric map.

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*)
devide by normal

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

nutils.numeric.**inv**(*A*)
Matrix inverse.

Fully equivalent to `numpy.linalg.inv()`, with the exception that upon singular systems *`inv()`* does not raise a `LinAlgError`, but rather issues a `RuntimeWarning` and returns NaN (not a number) values. For arguments of dimension >2 the return array contains NaN values only for those entries that correspond to singular matrices.

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

nutils.numeric.**unpack**(*n*, *atol*, *rtol*)
Convert packed representation to floating point data.

The packed binary form is a floating point interpretation of signed integer data, such that any integer `n` maps onto float `a` as follows:

```
a = nan                         if n = -N-1
a = -inf                        if n = -N
a = sinh(n*rtol)*atol/rtol      if -N < n < N
a = +inf                        if n = N,
```

where `N = 2**(nbits-1)-1` is the largest representable signed integer.

Note that packing is both order and zero preserving. The transformation is designed such that the spacing around zero equals `atol`, while the relative spacing for most of the data range is approximately constant at `rtol`. Precisely, the spacing between a value `a` and the adjacent value is `sqrt(atol**2 + (a*rtol)**2)`. Note that the truncation error equals half the spacing.

The representable data range depends on the values of `atol` and `rtol` and the bitsize of `n`. Useful values for different data types are:

dtype	rtol	atol	range
int8	2e-1	2e-06	4e+05
int16	2e-3	2e-15	1e+16
int32	2e-7	2e-96	2e+97

Parameters

- **n** (int array) – Integer data.
- **atol** (float) – Absolute tolerance.
- **rtol** (float) – Relative tolerance.

Returns

Return type float array

nutils.numeric.**pack**(*a*, *atol*, *rtol*, *dtype*)

Lossy compression of floating point data.

See *unpack()* for the definition of the packed binary form. The converse transformation uses rounding in packed domain to determine the closest matching value. In particular this may lead to values falling outside the representable data range to be clipped to infinity. Some examples of packed truncation:

```
>>> def truncate(a, dtype, **tol):
...   return unpack(pack(a, dtype=dtype, **tol), **tol)
>>> truncate(0.5, dtype='int16', atol=2e-15, rtol=2e-3)
0.5004...
>>> truncate(1, dtype='int16', atol=2e-15, rtol=2e-3)
0.9998...
>>> truncate(2, dtype='int16', atol=2e-15, rtol=2e-3)
2.0013...
>>> truncate(2, dtype='int16', atol=2e-15, rtol=2e-4)
inf
>>> truncate(2, dtype='int32', atol=2e-15, rtol=2e-4)
2.00013...
```

Parameters

- **a** (float array) – Input data.
- **atol** (float) – Absolute tolerance.
- **rtol** (float) – Relative tolerance.
- **dtype** (str or numpy dtype) – Target dtype for packed data.

Returns

Return type int array.

nutils.numeric.**accumulate**(*data*, *index*, *shape*)

accumulate scattered data in dense array.

Accumulates values from data in an array of shape shape at positions index, equivalent with:

```
>>> def accumulate(data, index, shape):
...   array = numpy.zeros(shape, data.dtype)
...   for v, *ij in zip(data, *index):
...     array[ij] += v
...   return array
```

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 secundary pariters will be treated like normal serial iterators.

Parameters

- **`iterable`** (`collections.abc.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`** (`callable`) – Takes item from iterable, returns numpy array of `shape` and `dtype`
- **`iterable`** (`collections.abc.Iterable`) – Collection of items
- **`nprocs`** (`int`) – Maximum number of processers 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.

class nutils.util.**NanVec**(*length*)
nan-initialized vector

nutils.util.**tri_merge**(*tri*, *x*, *mergetol=0*)
Create connected triangulation by connecting (near) identical points.

Based on a set of coordinates `x`, create a modified copy of `tri` with any occurrence of `j` replaced by `i` if `x[i]` equals `x[j]` within specified tolerance. The result is a triangulation that remains valid for any associated data vector that follows the same equality relations.

Example:

```
>>> x = [0,0], [1,0], [0,1], [1,0], [1,1] # note: x[1] == x[3])
>>> tri = [0,1,2], [2,3,4]
>>> tri_merge(tri, x)
array([[0, 1, 2],
       [2, 1, 4]])
```

Parameters

- **x** (`float` array) – Vertex coordinates.
- **tri** (`int` array) – Triangulation.
- **mergetol** (`float` (optional, default 0)) – Distance within which two points are considered equal. If mergetol == 0 then points are considered equal if and only if their coordinates are identical. If mergetol > 0 (required scipy) then points are considered equal if they are within euclidian distance < mergetol. If mergetol < 0 then tri is returned unchanged.

Returns **merged_tri**

Return type `int` array

class nutils.util.**tri_interpolator**(*tri*, *x*, *mergetol=0*)
Interpolate function values defined in triangulation vertices.

Convenience object that implements 2D interpolation on top of matplotlib's triangulation routines. Unlike matplotlib's own `LinearTriInterpolator`, the `tri_interpolator` allows for interpolation of multi-dimensional arrays, as well as repeated interpolations of different vertex values.

The arguments are identical to *tri_merge()*.

After instantiation of the interpolator object, interpolation coordinates are specified via the object's getitem operator. The resulting callable performs the interpolation:

```
>>> trix = [0,0], [1,0], [0,1], [1,1] # vertex coordinates
>>> triu = 0, 0, 10, 0 # vertex values
>>> interpolate = tri_interpolator([[0,1,2],[1,3,2]], trix)
>>> x = [.1,.1], [.1,.9], [.9,.9] # interpolation coordinates
>>> u = interpolate[x](triu) # interpolated values
```

nutils.util.**obj2str**(*obj*)
compact, lossy string representation of arbitrary object

nutils.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** (callable) – Method that expects a tuple as first positional argument, and that returns a list/tuple of the same length.

Returns Wrapped method.

nutils.util.**positional_only**(**names*, *keep_varpositional=False*)

Add var-positional arguments to function signature.

Python has no explicit syntax for defining positional-only parameters, but the effect can be achieved by using a var-positional argument and unpacking it inside the function body. The *positional_only()* decorator adds a check for the number of positional arguments provided, and updates the function signature to reflect this design. It requires that the first argument is var-positional, precluding positional-or-keyword arguments.

Example:

```
>>> @positional_only('x')
... def f(*args, **kwargs):
...   x, = args
>>> inspect.signature(f)
<Signature (x, /, **kwargs)>
```

```
>>> @positional_only('x', keep_varpositional=True)
... def f(*args, **kwargs):
...   x, *args = args
>>> inspect.signature(f)
<Signature (x, /, *args, **kwargs)>
```

Parameters

- **names** (variable argument list of str) – Names of the positional-only arguments, in order.
- **keep_varpositional** (bool (default: False)) – If True, retain the var_positional argument.

1.3.13 Types

Module with general purpose types.

nutils.types.**aspreprocessor**(*apply*)

Convert apply into a preprocessor decorator. When applied to a function, wrapped, the returned decorator preprocesses the arguments with apply before calling wrapped. The apply function should return a tuple of args (*tuple* or list) and kwargs (dict). The decorated function wrapped will be called with wrapped(*args, **kwargs). The apply function is allowed to change the signature of the decorated function.

Examples

The following preprocessor swaps two arguments.

```
>>> @aspreprocessor
... def swapargs(a, b):
...   return (b, a), {}
```

Decorating a function with `swapargs` will cause the arguments to be swapped before the wrapped function is called.

```
>>> @swapargs
... def func(a, b):
...   return a, b
>>> func(1, 2)
(2, 1)
```

`nutils.types.`**`apply_annotations`**(*wrapped*)

Decorator that applies annotations to arguments. All annotations should be `callable`s taking one argument and returning a transformed argument. All annotations are strongly recommended to be idempotent.

If a parameter of the decorated function has a default value `None` and the value of this parameter is `None` as well, the annotation is omitted.

Examples

Consider the following function.

```
>>> @apply_annotations
... def f(a:tuple, b:int):
...   return a + (b,)
```

When calling `f` with a `list` and `str` as arguments, the *`apply_annotations()`* decorator first applies *`tuple`* and `int` to the arguments before passing them to the decorated function.

```
>>> f([1, 2], '3')
(1, 2, 3)
```

The following example illustrates the behavior of parameters with default value `None`.

```
>>> addone = lambda x: x+1
>>> @apply_annotations
... def g(a:addone=None):
...   return a
```

When calling `g` without arguments or with argument `None`, the annotation `addone` is not applied. Note that `None + 1` would raise an exception.

```
>>> g() is None
True
>>> g(None) is None
True
```

When passing a different value, the annotation is applied:

```
>>> g(1)
2
```

nutils.types.**argument_canonicalizer**(*signature*)

Returns a function that converts arguments matching `signature` to canonical positional and keyword arguments. If possible, an argument is added to the list of positional arguments, otherwise to the keyword arguments dictionary. The returned arguments include default values.

Parameters **signature** (`inspect.Signature`) – The signature of a function to generate canonical arguments for.

Returns A function that returns a *`tuple`* of a *`tuple`* of positional arguments and a `dict` of keyword arguments.

Return type `callable`

Examples

Consider the following function.

```
>>> def f(a, b=4, *, c): pass
```

The `argument_canonicalizer` for `f` is generated as follows:

```
>>> canon = argument_canonicalizer(inspect.signature(f))
```

Calling `canon` with parameter `b` passed as keyword returns arguments with parameter `b` as positional argument:

```
>>> canon(1, c=3, b=2)
((1, 2), {'c': 3})
```

When calling `canon` without parameter `b` the default value is added to the positional arguments:

```
>>> canon(1, c=3)
((1, 4), {'c': 3})
```

nutils.types.**nutils_hash**(*data*)

Compute a stable hash of immutable object `data`. The hash is not affected by Python's hash randomization (see `object.__hash__()`).

Parameters **data** – An immutable object of type `bool`, `int`, `float`, `complex`, `str`, `bytes`, *`tuple`*, `frozenset`, or `Ellipsis` or `None`, or the type itself, or an object with a `__nutils_hash__` attribute.

Returns The hash of `data`.

Return type 40 `bytes`

class nutils.types.**CacheMeta**(*name*, *bases*, *namespace*, ***kwargs*)

Metaclass that adds caching functionality to properties and methods listed in the special attribute `__cache__`. If an attribute is of type `property`, the value of the property will be computed at the first attribute access and served from cache subsequently. If an attribute is a method, the arguments and return value are cached and the cached value will be used if a subsequent call is made with the same arguments; if not, the cache will be overwritten. The cache lives in private attributes in the instance. The metaclass supports the use of `__slots__`. If a subclass redefines a cached property or method (in the sense of this metaclass) of a base class, the property or method of the subclass is *not* automatically cached; `__cache__` should be used in the subclass explicitly.

Examples

An example of a class with a cached property:

```
>>> class T(metaclass=CacheMeta):
...   __cache__ = 'x',
...   @property
...   def x(self):
...     print('uncached')
...     return 1
```

The print statement is added to illustrate when method `x` (as defined above) is called:

```
>>> t = T()
>>> t.x
uncached
1
>>> t.x
1
```

An example of a class with a cached method:

```
>>> class U(metaclass=CacheMeta):
...   __cache__ = 'y',
...   def y(self, a):
...     print('uncached')
...     return a
```

Again, the print statement is added to illustrate when the method `y` (as defined above) is called:

```
>>> u = U()
>>> u.y(1)
uncached
1
>>> u.y(1)
1
>>> u.y(2)
uncached
2
>>> u.y(2)
2
```

class nutils.types.**ImmutableMeta**(*name*, *bases*, *namespace*, *, *version=0*, ***kwargs*)

class nutils.types.**Immutable**(**args*, ***kwargs*)

Base class for immutable types. This class adds equality tests, traditional hashing (`hash()`), nutils hashing (*`nutils_hash()`*) and pickling, all based solely on the (positional) intialization arguments, `args` for future reference. Keyword-only arguments are not supported. All arguments should be hashable by *`nutils_hash()`*.

Positional and keyword initialization arguments are canonicalized automatically (by *`argument_canonicalizer()`*). If the `__init__` method of a subclass is decorated with preprocessors (see *`aspreprocessor()`*), the preprocessors are applied to the initialization arguments and `args` becomes the preprocessed positional part.

Examples

Consider the following class.

```
>>> class Plain(Immutable):
...   def __init__(self, a, b):
...     pass
```

Calling `Plain` with equivalent positional or keyword arguments produces equal instances:

```
>>> Plain(1, 2) == Plain(a=1, b=2)
True
```

Passing unhashable values to `Plain` will fail:

```
>>> Plain([1, 2], [3, 4])
Traceback (most recent call last):
    ...
TypeError: unhashable type: 'list'
```

This can be solved by adding and applying annotations to `__init__`. The following class converts its initialization arguments to *`tuple`* automaticaly:

```
>>> class Annotated(Immutable):
...   @apply_annotations
...   def __init__(self, a:tuple, b:tuple):
...     pass
```

Calling `Annotated` with either `lists` of `1, 2` and `3, 4` or *`tuple`*s gives equal instances:

```
>>> Annotated([1, 2], [3, 4]) == Annotated((1, 2), (3, 4))
True
```

class nutils.types.**SingletonMeta**(*name*, *bases*, *namespace*, ***kwargs*)

class nutils.types.**Singleton**(**args*, ***kwargs*)

Subclass of *`Immutable`* that creates a single instance per unique set of initialization arguments.

Examples

Consider the following class.

```
>>> class Plain(Singleton):
...   def __init__(self, a, b):
...     pass
```

Calling `Plain` with equivalent positional or keyword arguments produces one instance:

```
>>> Plain(1, 2) is Plain(a=1, b=2)
True
```

Consider the folling class with annotations.

```
>>> class Annotated(Singleton):
...   @apply_annotations
...   def __init__(self, a:tuple, b:tuple):
...     pass
```

Calling `Annotated` with either `list`s of `1, 2` and `3, 4` or *`tuple`*s gives a single instance:

```
>>> Annotated([1, 2], [3, 4]) is Annotated((1, 2), (3, 4))
True
```

nutils.types.**strictint**(*value*)

Converts any type that is a subclass of `numbers.Integral` (e.g. `int` and `numpy.int64`) to `int`, and fails otherwise. Notable differences with the behavior of `int`:

- *`strictint()`* does not convert a `str` to an `int`.
- *`strictint()`* does not truncate `float` to an `int`.

Examples

```
>>> strictint(1), type(strictint(1))
(1, <class 'int'>)
>>> strictint(numpy.int64(1)), type(strictint(numpy.int64(1)))
(1, <class 'int'>)
>>> strictint(1.0)
Traceback (most recent call last):
    ...
ValueError: not an integer: 1.0
>>> strictint('1')
Traceback (most recent call last):
    ...
ValueError: not an integer: '1'
```

nutils.types.**strictfloat**(*value*)

Converts any type that is a subclass of `numbers.Real` (e.g. `float` and `numpy.float64`) to `float`, and fails otherwise. Notable difference with the behavior of `float`:

- *`strictfloat()`* does not convert a `str` to an `float`.

Examples

```
>>> strictfloat(1), type(strictfloat(1))
(1.0, <class 'float'>)
>>> strictfloat(numpy.float64(1.2)), type(strictfloat(numpy.float64(1.2)))
(1.2, <class 'float'>)
>>> strictfloat(1.2+3.4j)
Traceback (most recent call last):
    ...
ValueError: not a real number: (1.2+3.4j)
>>> strictfloat('1.2')
Traceback (most recent call last):
    ...
ValueError: not a real number: '1.2'
```

nutils.types.**strictstr**(*value*)

Returns `value` unmodified if it is a `str`, and fails otherwise. Notable difference with the behavior of `str`:

- *`strictstr()`* does not call `__str__` methods of objects to automatically convert objects to `str`s.

Examples

Passing a `str` to *`strictstr()`* works:

```
>>> strictstr('spam')
'spam'
```

Passing an `int` will fail:

```
>>> strictstr(1)
Traceback (most recent call last):
    ...
ValueError: not a 'str': 1
```

class nutils.types.**strict**(**args*, ***kwargs*)

Type checker. The function `strict[cls](value)` returns `value` unmodified if `value` is an instance of `cls`, otherwise a `ValueError` is raised.

Examples

The `strict[int]` function passes integers unmodified:

```
>>> strict[int](1)
1
```

Other types fail:

```
>>> strict[int]('1')
Traceback (most recent call last):
    ...
ValueError: expected an object of type 'int' but got '1' with type 'str'
```

class nutils.types.**tuple**(**args*, ***kwargs*)

Wrapper of *`tuple`* that supports a user-defined item constructor via the notation `tuple[I]`, with `I` the item constructor. This is shorthand for `lambda items: tuple(map(I, items))`. The item constructor should be any callable that takes one argument.

Examples

A tuple with items processed with *`strictint()`*:

```
>>> tuple[strictint]((False, 1, 2, numpy.int64(3)))
(0, 1, 2, 3)
```

If the item constructor raises an exception, the construction of the *`tuple`* failes accordingly:

```
>>> tuple[strictint]((1, 2, 3.4))
Traceback (most recent call last):
    ...
ValueError: not an integer: 3.4
```

class nutils.types.**frozendict**(*base*)

An immutable version of `dict`. The *`frozendict`* is hashable and both the keys and values should be hashable as well.

Custom key and value constructors can be supplied via the `frozendict[K,V]` notation, with `K` the key constructor and `V` the value constructor, which is roughly equivalent to `lambda *args, **kwargs: {K(k): V(v) for k, v in dict(*args, **kwargs).items()}`.

Examples

A *frozendict* with *strictstr()* as key constructor and *strictfloat()* as value constructor:

```
>>> frozendict[strictstr,strictfloat]({'spam': False})
frozendict({'spam': 0.0})
```

Similar but with non-strict constructors:

```
>>> frozendict[str,float]({None: '1.2'})
frozendict({'None': 1.2})
```

Applying the strict constructors to invalid data raises an exception:

```
>>> frozendict[strictstr,strictfloat]({None: '1.2'})
Traceback (most recent call last):
    ...
ValueError: not a 'str': None
```

class nutils.types.**frozenmultiset**(*items*)

An immutable multiset. A multiset is a generalization of a set: items may occur more than once. Two mutlisets are equal if they have the same set of items and the same item multiplicities.

A custom item constructor can be supplied via the notation `frozenmultiset[I]`, with `I` the item constructor. This is shorthand for `lambda items: frozenmultiset(map(I, items))`. The item constructor should be any callable that takes one argument.

Examples

```
>>> a = frozenmultiset(['spam', 'bacon', 'spam'])
>>> b = frozenmultiset(['sausage', 'spam'])
```

The *frozenmultiset* objects support +, − and & operators:

```
>>> a + b
frozenmultiset(['spam', 'bacon', 'spam', 'sausage', 'spam'])
>>> a - b
frozenmultiset(['bacon', 'spam'])
>>> a & b
frozenmultiset(['spam'])
```

The order of the items is irrelevant:

```
>>> frozenmultiset(['spam', 'spam', 'eggs']) == frozenmultiset(['spam', 'eggs',
↪'spam'])
True
```

The multiplicities, however, are not:

```
>>> frozenmultiset(['spam', 'spam', 'eggs']) == frozenmultiset(['spam', 'eggs'])
False
```

__and__ (*self*, *other*)
Return a *frozenmultiset* with elements from the left and right hand sides with strict positive multiplicity, where the multiplicity is the minimum of multiplicitie of the left and right hand side.

__add__ (*self*, *other*)
Return a *frozenmultiset* with elements from the left and right hand sides with a multiplicity equal to the sum of the left and right hand sides.

__sub__ (*self*, *other*)
Return a *frozenmultiset* with elements from the left hand sides with a multiplicity equal to the difference of the multiplicity of the left and right hand sides, truncated to zero. Elements with multiplicity zero are omitted.

class nutils.types.**frozenarray**(*base*, *dtype=None*, *copy=True*)
An immutable version (and drop-in replacement) of `numpy.ndarray`.

Besides being immutable, the *frozenarray* differs from `numpy.ndarray` in (in)equality tests. Given two *frozenarray* objects `a` and `b`, the test `a == b` returns `True` if both arrays are equal in its entirety, including dtype and shape, while the same test with `numpy.ndarray` objects would give a boolean array with element-wise thruth values.

The constructor with predefined `dtype` argument can generated via the notation `frozenarray[dtype]`. This is shorthand for `lambda base: frozenarray(base, dtype=dtype)`.

Parameters

- **base** (`numpy.ndarray` or array-like) – The array data.
- **dtype** – The dtype of the array or `None`.
- **copy** (`bool`) – If `base` is a *frozenarray* and the `dtype` matches or is `None`, this argument is ignored. If `base` is a `numpy.ndarray` and the `dtype` matches or is `None` and `copy` is `False`, `base` is stored as is. Otherwise `base` is copied.

class nutils.types.**c_array**(**args*, ***kwargs*)
Converts an array-like object to a ctypes array with a specific dtype. The function `c_array[dtype](array)` returns `array` unmodified if `array` is already a ctypes array. If `array` is a `numpy.ndarray`, the array is converted if the `dtype` is correct and the array is contiguous; otherwise `ValueError` is raised. Otherwise, `array` is first converted to a contiguous `numpy.ndarray` and then converted to ctypes array. In the first two cases changes made to the ctypes array are reflected by the `array` argument: both are essentially views of the same data. In the third case, changes to either `array` or the returned ctypes array are not reflected by the other.

class nutils.types.**attributes**(***args*)
Dictionary-like container with attributes instead of keys, instantiated using keyword arguments:

```
>>> A = attributes(foo=10, bar=True)
>>> A
attributes(bar=True, foo=10)
>>> A.foo
10
```

1.3.14 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(*self*)
close figure

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

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

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

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

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

slope_triangle(*self*, *x*, *y*, *fillcolor='0.9'*, *edgecolor='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, edgecolor: triangle style - slopefmt: format string for slope number

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

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

cspy(*self*, *A*, ***kwargs*)
Like pyplot.spy, but coloring acc to 10^log of absolute values, where [0, inf, 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 overriden 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.

Examples

Using a `with`-statement:

```
video = PyPlotVideo('video')
for timestep in timesteps:
  ...
```

(continues on next page)

(continued from previous page)

```
  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(*self*)
add a video frame

close(*self*)
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(*self*, *coords*)
set rectilinear grid

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

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

pointdataarray(*self*, *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

1.3.15 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.**enable**(*cachedir*)
Enable cacheing and set the cache directory to `cachedir`. Affects functions decorated with *function()* and subclasses of *Recursion*.

nutils.cache.**disable**()
Disable cacheing. Affects functions decorated with *function()* and subclasses of *Recursion*.

nutils.cache.**function**(*func=None*, *, *version=0*)
Decorator to wrap a function `func` with a memoizing callable. It is assumed that `func` computes its return value based strictly on the arguments. In other words: calling `func` with the same arguments repeatedly, should produce the same return value. All arguments passed to the decorator should be hashable (by *nutils.types.nutils_hash()*).

Memoization is controlled by the context managers *enable()* and *disable()*. If inside an *enable()* context, memoization is enabled: The first time the decorator is called with a unique set of arguments, the decorator calls `func` and stores the result on disk in the directory specified by the argument to *enable()*; when the decorator is called with the same arguments, the result is retrieved from the cache. If inside a *disable()* context, the decorator calls `func` directly, bypassing the cache. Note that memoization is off by default.

Parameters

- **func** (`callable`) – The function to be memoized.
- **version** (`int`) – Optional version number of `func`. Increment this if the behavior of `func` is changed. The decorator can be applied as follows:

```
>>> @function(version=1)
... def f(x):
...   return x
```

Returns A memoized version of `func`.

Return type `callable`

class nutils.cache.**Recursion**(**args*, ***kwargs*)
Base class for memoized iterators with fixed recursion. This class describes iterators of the form

$$x_i = f(x_{i-1}, x_{i-2}, \ldots, x_{i-n})$$

where n is the recursion length. The iterator is defined by the abstract *resume()* method. The method takes a single parameter `history`: a `list` of the last `length` items, or less if the iteration is resumed after less than `length` iterations. The method should proceed with yielding the remaining items. The *resume()* method should follow above definition of the recursion and the generator f should be based strictly on the initialization arguments of the subclass. Failing to do so will lead to unpredictable behavior if memoization is enabled. As this class bases *nutils.types.Immutable*, all initialization arguments should be hashable (by *nutils.types.nutils_hash()*).

The recursion length should be passed as keyword argument when defining the class. For example:

```
class Subclass(Recursion, length=1):
  def resume(self, history):
    ...
```

Memoization is controlled by the context managers *enable()* and *disable()*. If inside an *enable()* context, memoization is enabled: All cached iterations are retrieved from disk and are yielded; if iteration continues, the *resume()* method is called to produce the remaining iterations. If inside a *disable()* context, the memoization is disabled and the *resume()* method is called immediately with empty history.

Note that this class is iterable, but is not an iterator. Calling `iter()` on an instance of this class, e.g. implicitly in a `for` statement, the returned iterator always starts from scratch.

Examples

The Fibonacc sequence

$$f(x_{i-1}, x_{i-2}) := x_{i-1} + x_{i-2},$$

with variable seed values x_0 and x_1 can be implemented as follows.

```
>>> class Fibonacci(Recursion, length=2):
...   def __init__(self, x0, x1):
...     self.x0 = x0
...     self.x1 = x1
...   def resume(self, history):
...     if len(history) == 0:
...       yield self.x0
...       history.append(self.x0)
...     if len(history) == 1:
...       yield self.x1
...       history.append(self.x1)
...     while True:
...       value = history[-2] + history[-1]
...       yield value
...       history = history[-1], value
...
>>> f = iter(Fibonacci(1, 1))
>>> for i in range(6):
...   next(f)
1
1
2
3
5
8
```

resume (*self*, *history*)
Resume recursion from `history`.

Note: This function is abstract.

1.3.16 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

1.3.17 Solver

The solver module defines solvers for problems of the kind `res = 0` or `inertia/t + res = 0`, where `res` is a *nutils.sample.Integral*. 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')
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 > solver returned with residual ...
```

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.

exception nutils.solver.**ModelError**

nutils.solver.**solve_linear**(*target*, *residual*, *constrain=None*, *, *arguments={}*, *solveargs={}*)

solve linear problem

Parameters

- **target** (str) – Name of the target: a *nutils.function.Argument* in `residual`.
- **residual** (*nutils.sample.Integral*) – Residual integral, depends on `target`
- **constrain** (numpy.ndarray with dtype float) – 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 numpy.ndarray

nutils.solver.**solve**(*gen_lhs_resnorm*, *tol=0.0*, *maxiter=inf*)

execute nonlinear solver, return lhs

Iterates over nonlinear solver until tolerance is reached. Example:

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

Parameters

- **gen_lhs_resnorm** (collections.abc.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 numpy.ndarray

nutils.solver.**solve_withinfo**(*gen_lhs_resnorm*, *tol=0.0*, *maxiter=inf*)
execute nonlinear solver, return lhs and info

Like *solve()*, but return a 2-tuple of the solution and the corresponding info object which holds information about the final residual norm and other generator-dependent information.

class nutils.solver.**RecursionWithSolve**(**args*, ***kwargs*)
add a .solve method to (lhs,resnorm) iterators

Introduces the convenient form:

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

Shorthand for:

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

solve_withinfo(*gen_lhs_resnorm*, *tol=0.0*, *maxiter=inf*)
execute nonlinear solver, return lhs and info

Like *solve()*, but return a 2-tuple of the solution and the corresponding info object which holds information about the final residual norm and other generator-dependent information.

solve(*gen_lhs_resnorm*, *tol=0.0*, *maxiter=inf*)
execute nonlinear solver, return lhs

Iterates over nonlinear solver until tolerance is reached. Example:

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

Parameters

- **gen_lhs_resnorm** (collections.abc.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 numpy.ndarray

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 the residual norm. Suitable to be used inside solve.

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

```
|res(lhs + r * dlhs)|^2 = A + B * r + C * r^2 + D * r^3
```

where A, B, C and D are determined based on the current residual and tangent, the new residual, and the new tangent. If this value is found to be close to 1 (> maxrelax) then the newton update is accepted.

Parameters

- **target** (str) – Name of the target: a *nutils.function.Argument* in residual.
- **residual** (*nutils.sample.Integral*) –
- **lhs0** (numpy.ndarray) – Coefficient vector, starting point of the iterative procedure.
- **constrain** (numpy.ndarray with dtype bool or float) – 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 proceding 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.
- **droptol** (float) – Threshold for leaving entries in the return value at NaN if they do not contribute to the value of the functional.
- **arguments** (collections.abc.Mapping) – Defines the values for *nutils.function.Argument* objects in *residual*. The target should not be present in arguments. Optional.

Yields numpy.ndarray – Coefficient vector that approximates residual==0 with increasing accuracy

resume(*self*, *history*)

Resume recursion from history.

Note: This function is abstract.

---

class nutils.solver.**minimize**(**args*, ***kwargs*)

iteratively minimize nonlinear functional by gradient descent

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

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

```
energy(lhs + r * dlhs) = A + B * r + C * r^2 + D * r^3 + E * r^4 + F * r^5
```

where A, B, C, D, E and F are determined based on the current and new energy, residual and tangent. If this value is found to be close to 1 (> maxrelax) then the newton update is accepted.

Parameters

- **target** (str) – Name of the target: a *nutils.function.Argument* in residual.
- **residual** (*nutils.sample.Integral*) –
- **lhs0** (numpy.ndarray) – Coefficient vector, starting point of the iterative procedure.

- **constrain** (numpy.ndarray with dtype bool or float) – 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 proceding 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.
- **droptol** (float) – Threshold for leaving entries in the return value at NaN if they do not contribute to the value of the functional.
- **arguments** (collections.abc.Mapping) – Defines the values for *nutils.function.Argument* objects in *residual*. The target should not be present in arguments. Optional.

Yields numpy.ndarray – Coefficient vector that approximates residual==0 with increasing accuracy

resume (*self*, *history*)
Resume recursion from history.

Note: This function is abstract.

---

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** (*nutils.sample.Integral*) –
- **inertia** (*nutils.sample.Integral*) –
- **timestep** (float) – Initial time step, will scale up as residual decreases
- **lhs0** (numpy.ndarray) – Coefficient vector, starting point of the iterative procedure.
- **constrain** (numpy.ndarray with dtype bool or float) – 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 numpy.ndarray with dtype float – Tuple of coefficient vector and residual norm

resume (*self*, *history*)
Resume recursion from `history`.

Note: This function is abstract.

class nutils.solver.**thetamethod**(**args*, ***kwargs*)
solve time dependent problem using the theta method

Parameters

- **target** (`str`) – Name of the target: a *nutils.function.Argument* in `residual`.
- **residual** (*nutils.sample.Integral*) –
- **inertia** (*nutils.sample.Integral*) –
- **timestep** (`float`) – Initial time step, will scale up as residual decreases
- **lhs0** (`numpy.ndarray`) – Coefficient vector, starting point of the iterative procedure.
- **theta** (`float`) – Theta value (theta=1 for implicit Euler, theta=0.5 for Crank-Nicolson)
- **residual0** (*nutils.sample.Integral*) – Optional additional residual component evaluated in previous timestep
- **constrain** (`numpy.ndarray` with dtype `bool` or `float`) – 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 `numpy.ndarray` – Coefficient vector for all timesteps after the initial condition.

resume (*self*, *history*)
Resume recursion from `history`.

Note: This function is abstract.

nutils.solver.**optimize**(*target*, *functional*, *, *newtontol=0.0*, ***kwargs*)
find the minimizer of a given functional

Parameters

- **target** (`str`) – Name of the target: a *nutils.function.Argument* in `residual`.
- **functional** (scalar *nutils.sample.Integral*) – The functional the should be minimized by varying target
- **newtontol** (`float`) – Residual tolerance of Newton procedure (if applicable)
- ****kwargs** – Additional arguments for *minimize*

Yields `numpy.ndarray` – Coefficient vector corresponding to the functional optimum

1.3.18 Transform

The transform module.

class nutils.transform.**TransformItem**(**args*, ***kwargs*)
Affine transformation.

Base class for transformations of the type `A x + b`.

Parameters

- **todims** (int) – Dimension of the affine transformation domain.
- **fromdims** (int) – Dimension of the affine transformation range.

class nutils.transform.**Bifurcate**(**args*, ***kwargs*)

class nutils.transform.**Matrix**(**args*, ***kwargs*)

class nutils.transform.**Square**(**args*, ***kwargs*)

class nutils.transform.**Simplex**(**args*, ***kwargs*)

class nutils.transform.**Shift**(**args*, ***kwargs*)

class nutils.transform.**Identity**(**args*, ***kwargs*)

class nutils.transform.**Scale**(**args*, ***kwargs*)

class nutils.transform.**Updim**(**args*, ***kwargs*)

class nutils.transform.**SimplexEdge**(**args*, ***kwargs*)

class nutils.transform.**SimplexChild**(**args*, ***kwargs*)

class nutils.transform.**Slice**(**args*, ***kwargs*)

class nutils.transform.**ScaledUpdim**(**args*, ***kwargs*)

class nutils.transform.**TensorEdge1**(**args*, ***kwargs*)

class nutils.transform.**TensorEdge2**(**args*, ***kwargs*)

class nutils.transform.**TensorChild**(**args*, ***kwargs*)

class nutils.transform.**Identifier**(**args*, ***kwargs*)

1.3.19 Warnings

exception nutils.warnings.**NutilsWarning**

exception nutils.warnings.**NutilsDeprecationWarning**

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

1.3.20 Points

The points module defines the *Points* base class, which bundles point coordinates, point weights, a local triangulation and a hull triangulation. The module provides several different implementations such as *TensorPoints* and *SimplexGaussPoints* that reflect the variety of elements in the *nutils.element* module.

class nutils.points.**Points**(**args*, ***kwargs*)
Collection of points on an element.

The *Points* base class bundles point coordinates, point weights, a local triangulation and hull triangulation. Of these only the coordinates are mandatory, and should be provided by the derived class in the form of the `coords` attribute. Of the remaining properties only *hull()* has a functional base implementation that relies on the availability of `tri`.

coords
Coordinates of the points as a `float` array.

weights
Weights of the points as a `float` array.

Parameters

- **npoints** (`int`) – Number of discrete points.
- **ndims** (`int`) – Number of spatial dimensions.

tri
Triangulation of interior.

A two-dimensional integer array with `ndims+1` columns, of which every row defines a simplex by mapping vertices into the list of points.

hull
Triangulation of the exterior hull.

A two-dimensional integer array with `ndims` columns, of which every row defines a simplex by mapping vertices into the list of points.

onhull
Boolean mask marking boundary points.

The array of length `npoints` is `True` where the corresponding point is part of the *hull*, and `False` where it is not.

class nutils.points.**CoordsPoints**(**args*, ***kwargs*)
Manually supplied points.

class nutils.points.**CoordsWeightsPoints**(**args*, ***kwargs*)
Manually supplied points and weights.

class nutils.points.**CoordsUniformPoints**(**args*, ***kwargs*)
Manually supplied points with uniform weights.

class nutils.points.**TensorPoints**(**args*, ***kwargs*)
Tensor product of two Points instances.

class nutils.points.**SimplexGaussPoints**(**args*, ***kwargs*)
Gauss quadrature points on a simplex.

class nutils.points.**SimplexBezierPoints**(**args*, ***kwargs*)
Bezier points on a simplex.

class nutils.points.**TransformPoints**(**args*, ***kwargs*)
Affinely transformed Points.

tri
Triangulation of interior.

A two-dimensional integer array with `ndims+1` columns, of which every row defines a simplex by mapping vertices into the list of points.

hull
Triangulation of the exterior hull.

A two-dimensional integer array with `ndims` columns, of which every row defines a simplex by mapping vertices into the list of points.

class nutils.points.**ConcatPoints**(**args*, ***kwargs*)
Concatenation of several Points objects.

An optional `duplicates` argument lists all points that are equal, triggering deduplication and resulting in a smaller total point count.

class nutils.points.**ConePoints**(**args*, ***kwargs*)
Affinely transformed lower-dimensional points plus tip.

The point count is incremented by one regardless of the nature of the point set; no effort is made to introduce extra points between base plane and tip. Likewise, the simplex count stays equal, with all simplices obtaining an extra vertex in tip.

nutils.points.**gauss1**(*degree*)
Gauss quadrature for line.

nutils.points.**gauss2**
Gauss quadrature for triangle.

Reference: http://www.cs.rpi.edu/~flaherje/pdf/fea6.pdf

nutils.points.**gauss3**
Gauss quadrature for tetrahedron.

Reference http://www.cs.rpi.edu/~flaherje/pdf/fea6.pdf

1.3.21 Sample

The sample module defines the *Sample* class, which represents a collection of discrete points on a topology and is typically formed via *nutils.topology.Topology.sample()*. Any function evaluation starts from this sampling step, which drops element information and other topological properties such as boundaries and groups, but retains point positions and (optionally) integration weights. Evaluation is performed by subsequent calls to *Sample.integrate()*, *Sample.integral()* or *Sample.eval()*.

Besides the location of points, *Sample* also keeps track of point connectivity through its *Sample.tri* and *Sample.hull* properties, representing a (n-dimensional) triangulation of the interior and boundary, respectively. Availability of these properties depends on the selected sample points, and is typically used in combination with the "bezier" set.

In addition to *Sample*, the sample module defines the *Integral* class which represents postponed integration. Integrals are internally represented as pairs of *Sample* and *nutils.function.Array* objects. Evaluation proceeds via either the *Integral.eval()* method, or the *eval_integrals()* function. The latter can also be used to evaluate multiple integrals simultaneously, which has the advantage that it can efficiently combine common substructures.

class nutils.sample.**Sample**(**args*, ***kwargs*)
Collection of points on a topology.

The *Sample* class represents a collection of discrete points on a topology and is typically formed via *nutils.topology.Topology.sample()*. Any function evaluation starts from this sampling step, which drops

element information and other topological properties such as boundaries and groups, but retains point positions and (optionally) integration weights. Evaluation is performed by subsequent calls to *integrate()*, *integral()* or *eval()*.

Besides the location of points, Sample also keeps track of point connectivity through its *tri* and *hull* properties, representing a (n-dimensional) triangulation of the interior and boundary, respectively. Availability of these properties depends on the selected sample points, and is typically used in combination with the "bezier" set.

Parameters

- **transforms** (tuple or transformation chains) – List of transformation chains leading to local coordinate systems that contain points.
- **points** (tuple of point sets) – List of point sets matching transforms.
- **index** (tuple of integer arrays) – List of indices matching transforms, defining the order on which points show up in the evaluation.

integrate (*self*, *funcs*, *, *arguments=None*)
Integrate functions.

Parameters

- **funcs** (*nutils.function.Array* object or tuple thereof.) – The integrand(s).
- **arguments** (dict (default: None)) – Optional arguments for function evaluation.

integral (*self*, *func*)
Create Integral object for postponed integration.

Parameters **func** (*nutils.function.Array*) – Integrand.

eval (*self*, *funcs*, *, *arguments=None*)
Evaluate function.

Parameters

- **funcs** (*nutils.function.Array* object or tuple thereof.) – The integrand(s).
- **arguments** (dict (default: None)) – Optional arguments for function evaluation.

asfunction (*self*, *array*)
Convert sampled data to evaluable array.

Using the result of *Sample.eval()*, create a *nutils.function.Sampled* array that upon evaluation recovers the original function in the set of points matching the original sampling.

```
>>> from nutils import mesh
>>> domain, geom = mesh.rectilinear([1,2])
>>> gauss = domain.sample('gauss', 2)
>>> data = gauss.eval(geom)
>>> sampled = gauss.asfunction(data)
>>> domain.integrate(sampled, degree=2)
array([ 1.,  2.])
```

Parameters **array** – The sampled data.

tri
Triangulation of interior.

A two-dimensional integer array with ndims+1 columns, of which every row defines a simplex by mapping vertices into the list of points.

hull
Triangulation of the exterior hull.

A two-dimensional integer array with `ndims` columns, of which every row defines a simplex by mapping vertices into the list of points. Note that the hull often does contain internal element boundaries as the triangulations originating from separate elements are disconnected.

subset(*self*, *mask*)
Reduce the number of points.

Simple selection mechanism that returns a reduced Sample based on a selection mask. Points that are marked True will still be part of the new subset; points marked False may be dropped but this is not guaranteed. The point order of the original Sample is preserved.

Parameters **mask** (`bool` array.) – Boolean mask that selects all points that should remain. The resulting Sample may contain more points than this, but not less.

Returns **subset**

Return type *Sample*

class nutils.sample.**Integral**(**args*, ***kwargs*)
Postponed integration.

The *Integral* class represents postponed integration. Integrals are internally represented as pairs of *Sample* and *nutils.function.Array* objects. Evaluation proceeds via either the *eval()* method, or the *eval_integrals()* function. The latter can also be used to evaluate multiple integrals simultaneously, which has the advantage that it can efficiently combine common substructures.

Integrals support basic arithmetic such as summation, subtraction, and scalar multiplication and division. It also supports differentiation via the *derivative()* method. This makes Integral particularly well suited for use in combination with the *nutils.solver* module which provides linear and non-linear solvers.

Parameters **integrands** (`dict`) – Dictionary representing a sum of integrals, where every key-value pair binds together the sample set and the integrand.

eval(*self*, ***kwargs*)
Evaluate integral.

Equivalent to *eval_integrals()* (self, ...).

derivative(*self*, *target*)
Differentiate integral.

Return an Integral in which all integrands are differentiated with respect to a target. This is typically used in combination with *nutils.function.Namespace*, in which targets are denoted with a question mark (e.g. `'?dofs_n'` corresponds to target `'dofs'`).

Parameters **target** (`str`) – Name of the derivative target.

Returns **derivative**

Return type *Integral*

replace(*self*, *arguments*)
Return copy with arguments applied.

Return a copy of self in which all all arguments are edited into the integrands. The effect is that `self.eval(..., arguments=args)` is equivalent to `self.replace(args).eval(...)`. Note, however, that after the replacement it is no longer possible to take derivatives against any of the targets in `arguments`.

Parameters **arguments** (`dict`) – Arguments for function evaluation.

Returns replaced

Return type *Integral*

contains(*self*, *name*)
Test if target occurs in any of the integrands.

Parameters **name** (str) – Target name.

Returns iscontained

Return type bool

nutils.sample.**eval_integrals**(**integrals*, *arguments=None*)
Evaluate integrals.

Evaluate one or several postponed integrals. By evaluating them simultaneously, rather than using *Integral.eval()* on each integral individually, integrations will be grouped per Sample and jointly executed, potentially increasing efficiency.

Parameters

- **integrals** (tuple of integrals) – Integrals to be evaluated.
- **arguments** (dict (default: None)) – Optional arguments for function evaluation.

Returns results

Return type tuple of arrays and/or *nutils.matrix.Matrix* objects.

1.3.22 Export

nutils.export.**mplfigure**(*name*, */*, ***kwargs*)
Matplotlib figure context, convenience function.

Returns a matplotlib.figure.Figure object suitable for object-oriented plotting. Upon exit the result is saved using the agg-backend in all formats configured via *nutils.config.imagetype*, and the resulting filenames written to log.

Parameters

- **name** (str) – The filename (without extension) of the resulting figure(s)
- ****kwargs** – Keyword arguments are passed on unchanged to the constructor of the matplotlib.figure.Figure object.

nutils.export.**vtk**(*name*, *tri*, *x*, */*, ***kwargs*)
Export data to a VTK file.

This method provides a simple interface to the VTK file format with a number of important restrictions:

- Simplex-only. This makes it possible to define the mesh by a combination of vertex coordinates and a connectivity table.
- Legacy mode. The newer XML based format is more complex and does not provide benefits within the constraints set by this method.
- Binary mode. This allows for direct output of binary data, which aids speed, accuracy and file size.

Beyond the mandatory file name, connectivity table, and vertex coordinates, any additional data sets can be provided as keyword arguments, where the keys are the names by which the arrays are stored. The data can be either vertex or point data, with the distinction made based on the length of the array.

Parameters

- **name** (str) – Destination file name (without vtk extension).
- **tri** (int array) – Triangulation.
- **x** (float array) – Vertex coordinates.
- ****kwargs** – Cell and/or point data

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index

Symbols

A

B

C

D

E

F

G

H

P

R

S

T

U

V

W

Z