nutils Documentation

Release 1.0

Gertjan van Zwieten

August 28, 2014

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 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.

1.2.1 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`**(*ndims*)
  topology base class

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

  `elem_eval`(*funcs*, *ischeme*, *separate=False*, *title='evaluating'*)
    element-wise evaluation

  `elem_mean`(*funcs*, *geometry*, *ischeme*, *title='computing mean values'*)
    element-wise integration

grid_eval (*funcs*, *geometry*, *C*, *title='grid-evaluating'*)
evaluate grid points

build_graph (*func*)
get matrix sparsity

integrate (*funcs*, *ischeme*, *geometry=None*, *iweights=None*, *force_dense=False*, *title='integrating'*)

integrate_symm (*funcs*, *ischeme*, *geometry=None*, *iweights=None*, *force_dense=False*, *title='integrating'*)
integrate a symmetric integrand on a product domain

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

project (*fun*, *onto*, *geometry*, *tol=0*, *ischeme=None*, *title='projecting'*, *droptol=1e-08*, *exact_boundaries=False*, *constrain=None*, *verify=None*, *maxiter=0*, *ptype='lsqr'*)
L2 projection of function onto function space

refinedfunc (*dofaxis*, *refine*, *degree*, *title='refining'*)
create refined space by refining dofs in existing one

refine (*n*)
refine entire topology n times

get_simplices (*maxrefine*, *title='getting simplices'*)
Getting simplices

get_trimmededges (*maxrefine*, *title='getting trimmededges'*)
Getting trimmed edges

class nutils.topology.**StructuredTopology** (*structure*, *periodic=()*)
structured topology

make_periodic (*periodic*)
add periodicity

linearfunc ()
linears

rectilinearfunc (*gridvertices*)
rectilinear func

refined
refine entire topology

trim (*levelset*, *maxrefine*, *lscheme='bezier3'*, *finestscheme='uniform2'*, *evalrefine=0*, *title='trimming'*, *log=<module 'nutils.log' from '/var/build/user_builds/nutils/checkouts/v1.0/nutils/log.pyc'>*)
trim element along levelset

neighbor (*elem0*, *elem1*)
Neighbor detection, returns codimension of interface, -1 for non-neighboring elements.

class nutils.topology.**IndexedTopology** (*topo*, *elements*)
trimmed topology

splinefunc (*degree*)
create spline function space

class nutils.topology.**UnstructuredTopology** (*elements*, *ndims*, *namedfuncs={}*)
externally defined topology

splinefunc (*degree*)
spline func

linearfunc()
linear func

bubblefunc()
linear func + bubble

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

class nutils.topology.**ElemMap**(*mapping*, *ndims*)
dictionary-like element mapping

nutils.topology.**glue**(*master*, *slave*, *geometry*, *tol=1e-10*, *verbose=False*)
Glue topologies along boundary group __glue__.

1.2.2 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 `ArrayFunc` 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 `ArrayFunc` objects have directly recognizable numpy equivalents, such as `Sin` or `Inverse`. By not evaluating directly but merely stacking operations, coplex operations can be defined prior to entering a quadrature loop, allowing for a higher lever 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*, *evalf*)
Base class

recurse_index(*data*, *operations*, *cbuild*)
compile

compile()

graphviz(*title='graphviz'*)
create function graph

stackstr(*values=None*)
print stack

asciitree()
string representation

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

class nutils.function.**Tuple**(*items*)
combine

static vartuple(**f*)
evaluate

class nutils.function.**PointShape**
shape of integration points

static pointshape (*points*)
evaluate

class nutils.function.**Cascade** (*ndims*, *side=0*)
point cascade: list of (elem,points) tuples

static cascade (*elem*, *points*, *ndims*, *side*)
evaluate

class nutils.function.**ArrayFunc** (*evalf*, *args*, *shape*, *dtype=<type 'float'>*)
array function

vector (*ndims*)
vectorize

dot (*weights*, *axis=0*)
array contraction

find (*elem*, *C*)
iteratively find x for f(x) = target, starting at x=start

normalized ()
normalize last axis

normal (*ndims=-1*)

curvature (*ndims=-1*)

swapaxes (*n1*, *n2*)
swap axes

transpose (*trans=None*)

grad (*coords*, *ndims=0*)
gradient

laplace (*coords*, *ndims=0*)
laplacian

add_T (*axes=(-2, -1)*)
add transposed

symgrad (*coords*, *ndims=0*)
gradient

div (*coords*, *ndims=0*)
gradient

dotnorm (*coords*, *ndims=0*)
normal component

ngrad (*coords*, *ndims=0*)
normal gradient

nsymgrad (*coords*, *ndims=0*)
normal gradient

T
transpose

class nutils.function.**ElemArea** (*weights*)
element area

static elemarea (*weights*)
evaluate

class `nutils.function.`**`ElemInt`** (*func*, *weights*)
elementwise integration

static **`elemint`** (*w*, *f*, *ndim*)
evaluate

class `nutils.function.`**`Align`** (*func*, *axes*, *ndim*)
align axes

static **`align`** (*arr*, *trans*, *ndim*)

class `nutils.function.`**`Get`** (*func*, *axis*, *item*)
get

class `nutils.function.`**`Product`** (*func*, *axis*)
product

class `nutils.function.`**`IWeights`**
integration weights

static **`iweights`** (*elem*, *weights*)
evaluate

class `nutils.function.`**`OrientationHack`** (*side=0*)
orientation hack for 1d elements; VERY dirty

static **`orientation`** (*elem*, *side*)
evaluate

class `nutils.function.`**`Transform`** (*fromcascade*, *tocascade*, *side=0*)
transform

static **`transform`** (*fromcascade*, *tocascade*, *side*)

class `nutils.function.`**`Function`** (*cascade*, *stdmap*, *igrad*, *axis*)
function

static **`function`** (*cascade*, *stdmap*, *igrad*)
evaluate

class `nutils.function.`**`Choose`** (*level*, *choices*)
piecewise function

static **`choose`** (*level*, **choices*)

class `nutils.function.`**`Choose2D`** (*coords*, *contour*, *fin*, *fout*)
piecewise function

static **`choose2d`** (*xy*, *contour*, *fin*, *fout*)
evaluate

class `nutils.function.`**`Inverse`** (*func*)
inverse

class `nutils.function.`**`DofMap`** (*cascade*, *dofmap*, *axis*)
dof axis

static **`evalmap`** (*cascade*, *dofmap*)
evaluate

class `nutils.function.`**`Concatenate`** (*funcs*, *axis=0*)
concatenate

static **`concatenate`** (*iax*, **arrays*)
evaluate

class `nutils.function.`**`Interpolate`**(*x*, *xp*, *fp*, *left=None*, *right=None*)
interpolate uniformly spaced data; stepwise for now

class `nutils.function.`**`Cross`**(*func1*, *func2*, *axis*)
cross product

class `nutils.function.`**`Determinant`**(*func*)
normal

class `nutils.function.`**`DofIndex`**(*array*, *iax*, *index*)
element-based indexing

static **`dofindex`**(*arr*, **item*)
evaluate

class `nutils.function.`**`Multiply`**(*func1*, *func2*)
multiply

`cdef`()
generate C code

class `nutils.function.`**`Negative`**(*func*)
negate

class `nutils.function.`**`Add`**(*func1*, *func2*)
add

`cdef`()
generate C code

class `nutils.function.`**`BlockAdd`**(*func1*, *func2*)
block addition (used for DG)

class `nutils.function.`**`Dot`**(*func1*, *func2*, *naxes*)
dot

`cdef`()
generate C code

class `nutils.function.`**`Sum`**(*func*, *axis*)
sum

class `nutils.function.`**`Debug`**(*func*)
debug

static **`debug`**(*arr*)

class `nutils.function.`**`TakeDiag`**(*func*)
extract diagonal

class `nutils.function.`**`Take`**(*func*, *indices*, *axis*)
generalization of numpy.take(), to accept lists, slices, arrays

class `nutils.function.`**`Power`**(*func*, *power*)
power

class `nutils.function.`**`ElemFunc`**(*domainelem*, *side=0*)
trivial func

static **`elemfunc`**(*cascade*, *domainelem*)
evaluate

`find`(*elem*, *C*)
find coordinates

class `nutils.function.`**`Pointwise`**(*args*, *evalf*, *deriv*)
pointwise transformation

class `nutils.function.`**`Sign`**(*func*)
sign

class `nutils.function.`**`Eig`**(*func*, *symmetric=False*, *sort=False*)
Eig

class `nutils.function.`**`Zeros`**(*shape*)
zero

static **`zeros`**(*points*, *shape*)
prepend point axes

class `nutils.function.`**`Inflate`**(*func*, *dofmap*, *length*, *axis*)
inflate

static **`inflate`**(*array*, *indices*, *length*, *axis*)

class `nutils.function.`**`Diagonalize`**(*func*)
diagonal matrix

class `nutils.function.`**`Repeat`**(*func*, *length*, *axis*)
repeat singleton axis

class `nutils.function.`**`Const`**(*func*)
pointwise transformation

static **`const`**(*points*, *arr*)
prepend point axes

`nutils.function.`**`asarray`**(*arg*)
convert to ArrayFunc or numpy.ndarray

`nutils.function.`**`insert`**(*arg*, *n*)
insert axis

`nutils.function.`**`stack`**(*args*, *axis=0*)
stack functions along new axis

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

`nutils.function.`**`merge`**(*funcs*)
Combines unchained funcs into one function object.

`nutils.function.`**`vectorize`**(*args*)

`nutils.function.`**`expand`**(*arg*, *shape*)

`nutils.function.`**`repeat`**(*arg*, *length*, *axis*)

`nutils.function.`**`get`**(*arg*, *iax*, *item*)
get item

`nutils.function.`**`align`**(*arg*, *axes*, *ndim*)

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

`nutils.function.`**`elemint`**(*arg*, *weights*)
elementwise integration

`nutils.function.`**`grad`**(*arg*, *coords*, *ndims=0*)
local derivative

`nutils.function.`**`symgrad`**(*arg*, *coords*, *ndims=0*)
    symmetric gradient

`nutils.function.`**`div`**(*arg*, *coords*, *ndims=0*)
    gradient

`nutils.function.`**`sum`**(*arg*, *axes=-1*)
    sum over multiply axes

`nutils.function.`**`dot`**(*arg1*, *arg2*, *axes*)
    dot product

`nutils.function.`**`determinant`**(*arg*, *axes=(-2*, *-1)*)

`nutils.function.`**`inverse`**(*arg*, *axes=(-2*, *-1)*)

`nutils.function.`**`takediag`**(*arg*, *ax1=-2*, *ax2=-1*)

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

`nutils.function.`**`dotnorm`**(*arg*, *coords*, *ndims=0*)
    normal component

`nutils.function.`**`kronecker`**(*arg*, *axis*, *length*, *pos*)

`nutils.function.`**`diagonalize`**(*arg*)

`nutils.function.`**`concatenate`**(*args*, *axis=0*)

`nutils.function.`**`transpose`**(*arg*, *trans=None*)

`nutils.function.`**`product`**(*arg*, *axis*)

`nutils.function.`**`choose`**(*level*, *choices*)

`nutils.function.`**`cross`**(*arg1*, *arg2*, *axis*)
    cross product

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

`nutils.function.`**`pointwise`**(*args*, *evalf*, *deriv*)
    general pointwise operation

`nutils.function.`**`multiply`**(*arg1*, *arg2*)

`nutils.function.`**`add`**(*arg1*, *arg2*)

`nutils.function.`**`negative`**(*arg*)
    make negative

`nutils.function.`**`power`**(*arg*, *n*)

`nutils.function.`**`sign`**(*arg*)

`nutils.function.`**`eig`**(*arg, axes [symmetric]*)
    Compute the eigenvalues and vectors of a matrix. The eigenvalues and vectors are positioned on the last axes.

- tuple axes The axis on which the eigenvalues and vectors are calculated
- bool symmetric Is the matrix symmetric
- int sort Sort the eigenvalues and vectors (-1=descending, 0=unsorted, 1=ascending)

`nutils.function.`**`swapaxes`**(*arg*, *axes=(-2*, *-1)*)
    swap axes

`nutils.function.`**`opposite`**(*arg*)
evaluate jump over interface

`nutils.function.`**`function`**(*fmap*, *nmap*, *ndofs*, *ndims*)
create function on ndims-element

`nutils.function.`**`take`**(*arg*, *index*, *axis*)
take index

`nutils.function.`**`inflate`**(*arg*, *dofmap*, *length*, *axis*)

`nutils.function.`**`pointdata`**(*topo*, *ischeme*, *func=None*, *shape=None*, *value=None*)
point data

`nutils.function.`**`fdapprox`**(*func*, *w*, *dofs*, *delta=1e-05*)
Finite difference approximation of the variation of func in directions w around dofs. Input arguments: * func, the functional to differentiate * dofs, DOF vector of linearization point * w, the function space or a tuple of chained spaces * delta, finite difference step scaling of ||dofs||_inf

`nutils.function.`**`iwscale`**(*coords*, *ndims*)
integration weights scale

`nutils.function.`**`supp`**(*funcsp*, *indices*)
find support of selection of basis functions

class `nutils.function.`**`CBuilder`**(*cachedir='/tmp/nutils'*)
cbuilder

1.2.3 Core

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

1.2.4 Debug

The debug module provides code inspection tools and the "traceback explorer" interactive shell environment. Access to these components is primarily via `breakpoint()` and an exception handler in `nutils.util.run()`.

class `nutils.debug.`**`Frame`**(*frame*, *lineno=None*)
frame info

class `nutils.debug.`**`Explorer`**(*exc*, *frames*, *intro*)
traceback explorer

`show_context`()
show traceback up to index

`do_s`(*arg*)
Show source code of the currently focussed frame.

`do_l`(*arg*)
List the stack and exception type

`do_q`(*arg*)
Quit traceback exploror.

`do_u`(*arg*)
Shift focus to the frame above the current one.

do_d (*arg*)
Shift focus to the frame below the current one.

do_w (*arg*)
Show overview of local variables.

do_p (*arg*)
Print local of global variable, or function evaluation.

onecmd (*text*)
wrap command handling to avoid a second death

do_pp (*arg*)
Pretty-print local of global variable, or function evaluation.

completedefault (*text*, *line*, *begidx*, *endidx*)
complete object names

nutils.debug.**exception** ()
constructor

nutils.debug.**write_html** (*out*, *exc*, *frames*)
write exception info to html file

nutils.debug.**breakpoint** ()

1.2.5 Element

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

class nutils.element.**TrimmedIScheme** (*levelset*, *ischeme*, *maxrefine*, *finestscheme='uniform1'*, *degree=3*, *retain=None*)
integration scheme for truncated elements

generate_ischeme (*elem*, *maxrefine*)
generate integration scheme

class nutils.element.**Transformation** (*fromdim*, *todim*)
transform points

eval (*points*)
evaluate

class nutils.element.**SliceTransformation** (*fromdim*, *start=None*, *stop=None*, *step=None*)
take slice

class nutils.element.**AffineTransformation** (*offset*, *transform*)
affine transformation

invtrans
inverse transformation

det
determinant

nest (*other*)
merge transformations

get_transform()
get transformation copy

invapply(*coords*)
apply inverse transformation

class nutils.element.**Element**(*ndims*, *vertices*, *index=None*, *parent=None*, *context=None*, *interface=None*)
Element base class.

Represents the topological shape.

neighbor(*other*)
level of neighborhood; 0=self

eval(*where*)
get points

zoom(*elemset*, *points*)
zoom points

intersected(*levelset*, *lscheme*, *evalrefine=0*)
check levelset intersection:

+1 for levelset > 0 everywhere -1 for levelset < 0 everywhere =0 for intersected element

trim(*levelset*, *maxrefine*, *lscheme*, *finestscheme*, *evalrefine*)
trim element along levelset

get_simplices(*maxrefine*)
divide in simple elements

class nutils.element.**ProductElement**(*elem1*, *elem2*)
element product

orientation
Neighborhood of elem1 and elem2 and transformations to get mutual overlap in right location. Returns 3-element tuple: * neighborhood, as given by Element.neighbor(), * transf1, required rotation of elem1 map: {0:0, 1:pi/2, 2:pi, 3:3*pi/2}, * transf2, required rotation of elem2 map (is indep of transf1 in UnstructuredTopology.

eval(*where*)
get integration scheme

class nutils.element.**TrimmedElement**(*elem*, *levelset*, *maxrefine*, *lscheme*, *finestscheme*, *evalrefine*, *parent*, *vertices*)
trimmed element

children
all 1x refined elements

edge(*iedge*)

get_simplices(*maxrefine*)
divide in simple elements

class nutils.element.**QuadElement**(*ndims*, *vertices*, *index=None*, *parent=None*, *context=None*, *interface=None*)
quadrilateral element

children_by(*N*)
divide element by n

children
all 1x refined elements

ribbons
ribbons

edge (*iedge*)

refine (*n*)
refine n times

select_contained (*points*, *eps=0*)
select points contained in element

class nutils.element.**TriangularElement** (*vertices*, *index=None*, *parent=None*, *context=None*)
Triangular element. Conventions: * reference elem: unit simplex {(x,y) | x>0, y>0, x+y<1} * vertex numbering: {(1,0):0, (0,1):1, (0,0):2} * edge numbering: {bottom:0, slanted:1, left:2} * edge local coords run counter-clockwise.

children
all 1x refined elements

edge (*iedge*)

refined (*n*)
refine

select_contained (*points*, *eps=0*)
select points contained in element

class nutils.element.**TetrahedronElement** (*vertices*, *index=None*, *parent=None*, *context=None*)
tetrahedron element

children
all 1x refined elements

edge (*iedge*)

refined (*n*)
refine

select_contained (*points*, *eps=0*)
select points contained in element

class nutils.element.**StdElem**
stdelem base class

extract (*extraction*)
apply extraction matrix

class nutils.element.**PolyProduct**
multiply standard elements

class nutils.element.**PolyLine** (*poly*)
polynomial on a line

classmethod bernstein_poly (*degree*)
bernstein polynomial coefficients

classmethod spline_poly (*p*, *n*)
spline polynomial coefficients

classmethod spline (*degree*, *nelems*, *periodic=False*, *neumann=0*, *curvature=False*)
spline elements, any amount

extract (*extraction*)
apply extraction

class nutils.element.**PolyTriangle**
poly triangle (linear for now) conventions: dof numbering as vertices, see TriangularElement docstring.

class nutils.element.**BubbleTriangle**
linear triangle + bubble function conventions: dof numbering as vertices (see TriangularElement docstring), then barycenter.

class nutils.element.**ExtractionWrapper** (*stdelem*, *extraction*)
extraction wrapper

1.2.6 Library

The library module provides a collection of application specific functions, that nevertheless have a wide enough range of applicability to be useful as generic building blocks.

1.2.7 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.**ContextLog** (*depth=1*)
base class

disable ()
disable this logger

class nutils.log.**HtmlLog** (*fileobj*, *title*, *depth=1*)
html log

class nutils.log.**HtmlStream** (*chunks*, *attr*, *html*)
html line stream

write (*text*)
write to out and buffer for html

class nutils.log.**ProgressContextLog** (*text*, *iterable=None*, *target=None*, *showpct=True*, *depth=1*)
progress bar

text
get text

update (*current*)
update progress

nutils.log.**SimpleLog** (*chunks=('',)*, *attr=None*)
just write to stdout

class nutils.log.**StaticContextLog** (*text*, *depth=1*)
simple text logger

nutils.log.**context**
alias of `StaticContextLog`

nutils.log.**iterate**
alias of `ProgressContextLog`

`nutils.log.`**`progress`**
alias of `ProgressContextLog`

`nutils.log.`**`setup_html`**
alias of `HtmlLog`

`nutils.log.`**`stack`**(*msg*, *frames=None*)
print stack trace

1.2.8 Matrix

The matrix module defines a number of 2D matrix objects, notably the `SparseMatrix()` and `DenseMatrix()`. Matrix objects support indexed addition, basic addition and subtraction operations, and provide a consistent insterface for solving linear systems. Matrices can be converted to numpy arrays via `asarray`.

`nutils.matrix.`**`krylov`**(*matvec*, *b*, *x0=None*, *tol=1e-05*, *restart=None*, *maxiter=0*, *precon=None*, *callback=None*)
solve linear system iteratively

restart=None: CG restart=integer: GMRES

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

class `nutils.matrix.`**`Matrix`**(*(nrows*, *ncols)*)
matrix base class

`cond`(*constrain=None*, *lconstrain=None*, *rconstrain=None*)
condition number

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

class `nutils.matrix.`**`DenseSubMatrix`**(*data*, *indices*)
dense but non-contiguous data

class `nutils.matrix.`**`SparseMatrix`**(*graph*, *ncols=None*)
sparse matrix

`reshape`(*(nrows*, *ncols)*)
reshape matrix

`clone`()
clone matrix

`matvec`(*other*)
matrix-vector multiplication

`T`
transpose

`toarray`()
convert to numpy array

`todense`()
convert to dense matrix

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

`solve`(*b=0*, *constrain=None*, *lconstrain=None*, *rconstrain=None*, *tol=0*, *x0=None*, *symmetric=False*, *maxiter=0*, *restart=999*, *title='solving system'*, *callback=None*, *precon=None*)

class `nutils.matrix.`**`DenseMatrix`**(*shape*)
matrix wrapper class

`clone`()
clone matrix

`addblock`(*rows*, *cols*, *vals*)
add matrix data

`toarray`()
convert to numpy array

`matvec`(*other*)
matrix-vector multiplication

`T`
transpose

`solve`(*b=0*, *constrain=None*, *lconstrain=None*, *rconstrain=None*, *title='solving system'*, ***dummy*)

1.2.9 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, `gmesh()`, `igatool()`, 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`**(*vertices*, *periodic=()*, *name='rect'*)
rectilinear mesh

`nutils.mesh.`**`revolve`**(*topo*, *coords*, *nelems*, *degree=3*, *axis=0*)
revolve coordinates

`nutils.mesh.`**`gmesh`**(*path*, *btags={}*, *name=None*)

`nutils.mesh.`**`triangulation`**(*vertices*, *nvertices*)

`nutils.mesh.`**`igatool`**(*path*, *name=None*)
igatool mesh

`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

1.2.10 Numeric

The numeric module provides methods that are lacking from the numpy module. An accompanying extension module _numeric.c should be compiled to benefit from extra performance, although a Python-only implementation is provided as fallback. A warning message is printed if the extension module is not found.

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

`nutils.numeric.`**`align`**(*arr*, *trans*, *ndim*)
create new array of ndim from arr with axes moved accordin to trans

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

`nutils.numeric.`**`expand`**`(`*arr*, **shape*`)`

`nutils.numeric.`**`linspace2d`**`(`*start*, *stop*, *steps*`)`
: linspace & meshgrid combined

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

`nutils.numeric.`**`contract_fast`**`(`*A*, *B*, *naxes*`)`
: contract last n axes

`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.`**`fastrepeat`**`(`*A*, *nrepeat*, *axis=-1*`)`
: repeat axis by 0stride

`nutils.numeric.`**`fastmeshgrid`**`(`*X*, *Y*`)`
: mesh grid based on fastrepeat

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

`nutils.numeric.`**`appendaxes`**`(`*A*, *shape*`)`
: append axes by 0stride

`nutils.numeric.`**`inverse`**`(`*A*`)`
: linearized inverse

`nutils.numeric.`**`determinant`**`(`*A*`)`

`nutils.numeric.`**`eig`**`(`*A*, *sort=False*`)`
: Compute the eigenvalues and vectors of a hermitian matrix sort -1/0/1 -> descending / unsorted / ascending

`nutils.numeric.`**`eigh`**`(`*A*, *sort=False*`)`
: Compute the eigenvalues and vectors of a hermitian matrix sort -1/0/1 -> descending / unsorted / ascending

`nutils.numeric.`**`reshape`**`(`*A*, **shape*`)`
: more useful reshape

`nutils.numeric.`**`mean`**`(`*A*, *weights=None*, *axis=-1*`)`
: generalized mean

`nutils.numeric.`**`norm2`**`(`*A*, *axis=-1*`)`
: L2 norm over specified axis

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

`nutils.numeric.`**`cross`**`(`*v1*, *v2*, *axis*`)`
: cross product

`nutils.numeric.`**`stack`**`(`*arrays*, *axis=0*`)`
: powerful array stacker with singleton expansion

`nutils.numeric.`**`bringforward`**`(`*arg*, *axis*`)`
: bring axis forward

`nutils.numeric.`**`diagonalize`**`(`*arg*`)`
: append axis, place last axis on diagonal of self and new

1.2.11 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.

class `nutils.parallel.`**`Fork`**(*nprocs*)
nested fork context, unwinds at exit

class `nutils.parallel.`**`AlternativeFork`**(*nprocs*)
single master, multiple slave fork context, unwinds at exit

`nutils.parallel.`**`fork`**(*func*, *nice=19*)
fork and run (return value is lost)

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

`nutils.parallel.`**`pariter`**(*iterable*)
iterate parallel

1.2.12 Util

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

class `nutils.util.`**`ImmutableArray`**
immutable array

`nutils.util.`**`delaunay`**(*points*)
delaunay triangulation

`nutils.util.`**`withrepr`**(*f*)
add string representation to generated function

class `nutils.util.`**`Cache`**(**args*)
cache

`nutils.util.`**`getpath`**(*pattern*)
create file in dumpdir

`nutils.util.`**`sum`**(*seq*)
a better sum

`nutils.util.`**`product`**(*seq*)
multiply items in sequence

`nutils.util.`**`clone`**(*obj*)
clone object

`nutils.util.`**`iterate`**(*context='iter'*, *nmax=-1*)
iterate forever

class `nutils.util.`**`NanVec`**
nan-initialized vector

`where`
find non-nan items

class `nutils.util.`**`Clock`**(*interval*)
simple interval timer

`nutils.util.`**`tensorial`**(*args*)
create n-dimensional array containing tensorial combinations of n args

`nutils.util.`**`arraymap`**(*f*, *dtype*, **args*)
call f for sequence of arguments and cast to dtype

`nutils.util.`**`objmap`**(*func*, **arrays*)
map numpy arrays

`nutils.util.`**`fail`**(*msg*, **args*)
generate exception

class `nutils.util.`**`Locals`**
local namespace as object

`nutils.util.`**`getkwargdefaults`**(*func*)
helper for run

class `nutils.util.`**`Statm`**(*rusage=None*)
memory statistics on systems that support it

`nutils.util.`**`run`**(**functions*)
call function specified on command line

1.2.13 Plot

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

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

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

`save`(*name*)
save images

`mesh`(*points*, *colors=None*, *edgecolors='k'*, *edgewidth=None*, *triangulate='delaunay'*, *setxylim=True*, ***kwargs*)
plot elemtwise mesh

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

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

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

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

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

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

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

`unstructuredgrid`(*points*, *npars=None*)
add unstructured grid

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

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

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

class `nutils.plot.`**`Pylab`**(*title*, *name='graph{0:03x}'*)
matplotlib figure

class `nutils.plot.`**`PylabAxis`**(*ax*, *title*)
matplotlib axis augmented with nutils-specific functions

`add_mesh`(*coords*, *topology*, *deform=0*, *color=None*, *edgecolors='none'*, *linewidth=1*, *xmargin=0*, *ymargin=0*, *aspect='equal'*, *cbar='vertical'*, *title=None*, *ischeme='gauss2'*, *cscheme='contour3'*, *clim=None*, *frame=True*, *colormap=None*)
plot mesh

`add_quiver`(*coords*, *topology*, *quiver*, *sample='uniform3'*, *scale=None*)
quiver builder

`add_graph`(*xfun*, *yfun*, *topology*, *sample='contour10'*, *logx=False*, *logy=False*, ***kwargs*)
plot graph of function on 1d topology

`add_convplot`(*x*, *y*, *drop=0.8*, *shift=1.1*, *slope=True*, ***kwargs*)
Convergence plot including slope triangle (below graph) for supplied y(x), drop = distance graph & triangle, shift = distance triangle & text.

`nutils.plot.`**`preview`**(*coords*, *topology*, *cscheme='contour8'*)
preview function

CHAPTER 2

Indices and tables

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