nutils Documentation

Release 1.0

Gertjan van Zwieten

February 18, 2016

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

1.3.2 Function

1.3.3 Core

1.3.4 Debug

1.3.5 Element

1.3.6 Library

1.3.7 Log

1.3.8 Matrix

1.3.9 Mesh

1.3.10 Numeric

1.3.11 Parallel

1.3.12 Util

1.3.13 Plot

CHAPTER 2

Indices and tables

- genindex
- modindex
- search