dolfin_navier_scipy Documentation

Release 1.0

highlando

May 04, 2017

Contents

The package *dolfin_navier_scipy (dns)* provides an interface between *scipy* and *FEniCS* in view of solving Navier-Stokes Equations. *FEniCS* is used to perform a Finite Element discretization of the equations. The assembled coefficients are exported as sparse matrices for use in *scipy*. Nonlinear and time-dependent parts are evaluated and assembled on demand. Visualization is done via the *FEniCS* interface to *paraview*.

Contents:

CHAPTER 1

Code Reference

dolfin_to_sparrays

`dolfin_to_sparrays.`**`ass_convmat_asmatquad`**(*W=None*, *invindsw=None*)

assemble the convection matrix H, so that N(v)v = H[v.v]

for the inner nodes.

Notes

Implemented only for 2D problems

`dolfin_to_sparrays.`**`get_stokessysmats`**(*V*, *Q*, *nu=None*, *bccontrol=False*, *cbclist=None*, *cbshapefuns=None*)

Assembles the system matrices for Stokes equation

in mixed FEM formulation, namely

$$\begin{bmatrix} A & -J' \\ J & 0 \end{bmatrix} : V \times Q \to V' \times Q'$$

as the discrete representation of

$$\begin{bmatrix} -\Delta & \text{grad} \\ \text{div} & 0 \end{bmatrix}$$

plus the velocity and pressure mass matrices

for a given trial and test space W = V * Q not considering boundary conditions.

Parameters **V** : dolfin.VectorFunctionSpace

Fenics VectorFunctionSpace for the velocity

Q : dolfin.FunctionSpace

Fenics FunctionSpace for the pressure

nu : float, optional

viscosity parameter - defaults to 1

bccontrol : boolean, optional

whether boundary control (via penalized Robin conditions) is applied, defaults to *False*

cbclist : list, optional

list of dolfin's Subdomain classes describing the control boundaries

cbshapefuns : list, optional

list of spatial shape functions of the control boundaries

Returns stokesmats, dictionary :

a dictionary with the following keys:

- `M`: the mass matrix of the velocity space,
- `A`: the stiffness matrix $\nu \int_\Omega (\nabla \phi_i, \nabla \phi_j)$
- `JT`: the gradient matrix,
- `J`: the divergence matrix, and
- `MP`: the mass matrix of the pressure space
- `Apbc`: (N, N) sparse matrix, the contribution of the Robin conditions to *A* $\nu \int_\Gamma (\phi_i, \phi_j)$
- `Bpbc`: (N, k) sparse matrix, the input matrix of the Robin conditions $\nu \int_\Gamma (\phi_i, g_k)$, where g_k is the shape function associated with the j-th control boundary segment

`dolfin_to_sparrays.`**`get_convmats`**(*u0_dolfun=None*, *u0_vec=None*, *V=None*, *invinds=None*, *diribcs=None*)

returns the matrices related to the linearized convection

where u_0 is the linearization point

Returns N1 : (N,N) sparse matrix

representing $(u_0 \cdot \nabla)u$

N2 : (N,N) sparse matrix

representing $(u \cdot \nabla)u_0$

fv : (N,1) array

representing $(u_0 \cdot \nabla)u_0$

See also:

`stokes_navier_utils.get_v_conv_conts` the convection contributions reduced to the inner nodes

`dolfin_to_sparrays.`**`get_curfv`**(*V*, *fv*, *invinds*, *tcur*)

get the fv at innernotes at t=tcur

`dolfin_to_sparrays.`**`get_convvec`**(*u0_dolfun=None*, *V=None*, *u0_vec=None*, *femp=None*, *diribcs=None*, *invinds=None*)

return the convection vector e.g. for explicit schemes

given a dolfin function or the coefficient vector `dolfin_to_sparrays.`**`condense_sysmatsbybcs`**(*stms*, *velbcs*)

resolve the Dirichlet BCs and condense the system matrices

to the inner nodes

Parameters **stms: dict** :

of the stokes matrices with the keys

- `M`: the mass matrix of the velocity space,
- `A`: the stiffness matrix,
- `JT`: the gradient matrix,
- `J`: the divergence matrix, and
- `MP`: the mass matrix of the pressure space

velbcs : list

of dolfin Dirichlet boundary conditions for the velocity

Returns **stokesmatsc** : dict

a dictionary of the condensed matrices:

- `M`: the mass matrix of the velocity space,
- `A`: the stiffness matrix,
- `JT`: the gradient matrix, and
- `J`: the divergence matrix
- `MP`: the mass matrix of the pressure space

rhsvecsb : dict

a dictionary of the contributions of the boundary data to the rhs:

- `fv`: contribution to momentum equation,
- `fp`: contribution to continuity equation

invinds : (N,) array

vector of indices of the inner nodes

bcinds : (K,) array

vector of indices of the boundary nodes

bcvals : (K,) array

vector of the values of the boundary nodes

`dolfin_to_sparrays.`**`condense_velmatsbybcs`**(*A*, *velbcs*, *return_bcinfo=False*)

resolve the Dirichlet BCs, condense velocity related matrices

to the inner nodes, and compute the rhs contribution This is necessary when, e.g., the convection matrix changes with time

Parameters **A** : (N,N) sparse matrix

coefficient matrix for the velocity

velbcs : list

of dolfin *dolfin* Dirichlet boundary conditions for the velocity

return_bcinfo : boolean, optional

if *True* a dict with the inner and the boundary indices is returned, defaults to *False*

Returns **Ac** : (K, K) sparse matrix

the condensed velocity matrix

fvbc : (K, 1) array

the contribution to the rhs of the momentum equation

dict, on demand :

with the keys

- `ininds`: indices of the inner nodes
- `bcinds`: indices of the boundary nodes

`dolfin_to_sparrays.`**`expand_vp_dolfunc`**(*V=None*, *Q=None*, *invinds=None*, *diribcs=None*, *vp=None*, *vc=None*, *pc=None*, *ppin=-1*, ***kwargs*)

expand v [and p] to the dolfin function representation

Parameters **V** : dolfin.VectorFunctionSpace

FEM space of the velocity

Q : dolfin.FunctionSpace

FEM space of the pressure

invinds : (N,) array

vector of indices of the velocity nodes

diribcs : list, optional

of the (Dirichlet) velocity boundary conditions, if *None* it is assumed that *vc* already contains the bc, defaults to *None*

vp : (N+M,1) array, optional

solution vector of velocity and pressure

vc : (N,1) array, optional

solution vector of velocity

pc : (M,1) array, optional

solution vector of pressure

ppin : {int, None}, optional

which dof of *p* is used to pin the pressure, defaults to *-1*

Returns **v** : dolfin.Function(V)

velocity as function

p : dolfin.Function(Q), optional

pressure as function

See also:

`expand_vecnbc_dolfunc` for a scalar function with multiple bcs dolfin_to_sparrays.**expand_vecnbc_dolfunc**(*V=None*, *vec=None*, *bcindsl=None*, *bcvalsl=None*, *diribcs=None*, *bcsfaclist=None*, *invinds=None*)

expand a function vector with changing boundary conditions

the boundary conditions may not be disjoint, what is used to model spatial dependencies of a control at the boundary.

Parameters **V** : dolfin.FunctionSpace

FEM space of the scalar

invinds : (N,) array

vector of indices of the velocity nodes

vec : (N,1) array

solution vector

diribcs : list

of boundary conditions

bcsfaclist : list, optional

of factors for the boundary conditions

Returns **dolfin.function** :

of the vector values and the bcs

dolfin_to_sparrays.**append_bcs_vec**(*vvec*, *V=None*, *vdim=None*, *invinds=None*, *diribcs=None*, ***kwargs*)

append given boundary conditions to a vector representing inner nodes

dolfin_to_sparrays.**mat_dolfin2sparse**(*A*)

get the csr matrix representing an assembled linear dolfin form

stokes_navier_utils

stokes_navier_utils.**get_v_conv_conts**(*prev_v=None*, *V=None*, *invinds=None*, *diribcs=None*, *Picard=False*)

get and condense the linearized convection

to be used in a Newton scheme

$$(u \cdot \nabla)u \to (u_0 \cdot \nabla)u + (u \cdot \nabla)u_0 - (u_0 \cdot \nabla)u_0$$

or in a Picard scheme

$$(u \cdot \nabla)u \to (u_0 \cdot \nabla)u$$

Parameters **prev_v** : (N,1) ndarray

convection velocity

V : dolfin.VectorFunctionSpace

FEM space of the velocity

invinds : (N,) ndarray or list

indices of the inner nodes

diribcs : list

of dolfin Dirichlet boundary conditons

Picard : boolean

whether Picard linearization is applied, defaults to *False*

Returns **convc_mat** : (N,N) sparse matrix

representing the linearized convection at the inner nodes

rhs_con : (N,1) array

representing $(u_0 \cdot \nabla)u_0$ at the inner nodes

rhsv_conbc : (N,1) ndarray

representing the boundary conditions

stokes_navier_utils.**solve_nse**(*A=None*, *M=None*, *J=None*, *JT=None*, *fv=None*, *fp=None*, *fvc=None*, *fpc=None*, *fv_tmdp=None*, *fv_tmdp_params={}*, *fv_tmdp_memory=None*, *iniv=None*, *lin_vel_point=None*, *stokes_flow=False*, *trange=None*, *t0=None*, *tE=None*, *Nts=None*, *V=None*, *Q=None*, *invinds=None*, *diribcs=None*, *output_includes_bcs=False*, *N=None*, *nu=None*, *ppin=-1*, *closed_loop=False*, *static_feedback=False*, *feedbackthroughdict=None*, *return_vp=False*, *tb_mat=None*, *c_mat=None*, *vel_nwtn_stps=20*, *vel_nwtn_tol=5e-15*, *vel_pcrd_stps=4*, *krylov=None*, *krpslvprms={}*, *krplsprms={}*, *clearprvdata=False*, *get_datastring=None*, *data_prfx=''*, *paraviewoutput=False*, *vfileprfx=''*, *pfileprfx=''*, *return_dictofvelstrs=False*, *return_dictofpstrs=False*, *dictkeysstr=False*, *comp_nonl_semexp=False*, *return_as_list=False*, *start_ssstokes=False*, ***kw*)

solution of the time-dependent nonlinear Navier-Stokes equation

$$\begin{aligned} M\dot{v} + Av + N(v)v + J^T p &= f \\ Jv &= g \end{aligned}$$

using a Newton scheme in function space, i.e. given v_k, we solve for the update like

$$M\dot{v} + Av + N(v_k)v + N(v)v_k + J^T p = N(v_k)v_k + f,$$

and trapezoidal rule in time. To solve an *Oseen* system (linearization about a steady state) or a *Stokes* system, set the number of Newton steps to one and provide a linearization point and an initial value.

Parameters **lin_vel_point** : dictionary, optional

contains the linearization point for the first Newton iteration

- Steady State: {{*None*: 'path_to_nparray'}, {'None': nparray}}
- Newton: {*t*: 'path_to_nparray'}

defaults to *None*

dictkeysstr : boolean, optional

whether the *keys* of the result dictionaries are strings instead of floats, defaults to *False*

fv_tmdp : callable f(t, v, dict), optional

time-dependent part of the right-hand side, set to zero if None

fv_tmdp_params : dictionary, optional

dictionary of parameters to be passed to *fv_tmdp*, defaults to *{}*

fv_tmdp_memory : dictionary, optional

memory of the function

output_includes_bcs : boolean, optional

whether append the boundary nodes to the computed and stored velocities, defaults to *False*

krylov : {None, 'gmres'}, optional

whether or not to use an iterative solver, defaults to *None*

krpslvprms : dictionary, optional

to specify parameters of the linear solver for use in Krypy, e.g.,

- initial guess
- tolerance
- number of iterations

defaults to *None*

krplsprms : dictionary, optional

parameters to define the linear system like

- preconditioner

ppin : {int, None}, optional

which dof of *p* is used to pin the pressure, defaults to *-1*

stokes_flow : boolean, optional

whether to consider the Stokes linearization, defaults to *False*

start_ssstokes : boolean, optional

for your convenience, compute and use the steady state stokes solution as initial value, defaults to *False*

Returns **dictofvelstrs** : dictionary, on demand

dictionary with time *t* as keys and path to velocity files as values

dictofpstrs : dictionary, on demand

dictionary with time *t* as keys and path to pressure files as values

vellist : list, on demand

list of the velocity solutions

stokes_navier_utils.**solve_steadystate_nse**(*A=None, J=None, JT=None, M=None, fv=None, fp=None, V=None, Q=None, invinds=None, diribcs=None, return_vp=False, ppin=-1, N=None, nu=None, vel_pcrd_stps=10, vel_pcrd_tol=0.0001, vel_nwtn_stps=20, vel_nwtn_tol=5e-15, clearprvdata=False, vel_start_nwtn=None, get_datastring=None, data_prfx='', paraviewoutput=False, save_intermediate_steps=False, vfileprfx='', pfileprfx='', **kw*)

Solution of the steady state nonlinear NSE Problem

using Newton's scheme. If no starting value is provided, the iteration is started with the steady state Stokes solution.

Parameters **A** : (N,N) sparse matrix

stiffness matrix aka discrete Laplacian, note the sign!

M : (N,N) sparse matrix

mass matrix

J : (M,N) sparse matrix

discrete divergence operator

JT : (N,M) sparse matrix, optional

discrete gradient operator, set to J.T if not provided

fv, fp : (N,1), (M,1) ndarrays

right hand sides restricted via removing the boundary nodes in the momentum and the pressure freedom in the continuity equation

ppin : {int, None}, optional

which dof of *p* is used to pin the pressure, defaults to *-1*

return_vp : boolean, optional

whether to return also the pressure, defaults to *False*

vel_pcrd_stps : int, optional

Number of Picard iterations when computing a starting value for the Newton scheme, cf. Elman, Silvester, Wathen: *FEM and fast iterative solvers*, 2005, defaults to *100*

vel_pcrd_tol : real, optional

tolerance for the size of the Picard update, defaults to *1e-4*

vel_nwtn_stps : int, optional

Number of Newton iterations, defaults to *20*

vel_nwtn_tol : real, optional

tolerance for the size of the Newton update, defaults to *5e-15*

stokes_navier_utils.**get_pfromv**(*v=None, V=None, M=None, A=None, J=None, fv=None, fp=None, diribcs=None, invinds=None, **kwargs*)

for a velocity *v*, get the corresponding *p*

Notes

Formula is only valid for constant rhs in the continuity equation

problem_setups

`problem_setups.get_sysmats`(*problem='drivencavity'*, *N=10*, *scheme=None*, *ppin=None*, *Re=None*, *nu=None*, *bccontrol=False*, *mergerhs=False*, *onlymesh=False*)

retrieve the system matrices for stokes flow

Parameters **problem** : {'drivencavity', 'cylinderwake'}

problem class

N : int

mesh parameter

nu : real, optional

kinematic viscosity, is set to *L/Re* if *Re* is provided

Re : real, optional

Reynoldsnumber, is set to *L/nu* if *nu* is provided

bccontrol : boolean, optional

whether to consider boundary control via penalized Robin defaults to *False*

mergerhs : boolean, optional

whether to merge the actual rhs and the contribution from the boundary conditions into one rhs

onlymesh : boolean, optional

whether to only return *femp*, containing the mesh and FEM spaces, defaults to *False*

Returns **femp** : dict

with the keys:

- *V*: FEM space of the velocity
- *Q*: FEM space of the pressure
- *diribcs*: list of the (Dirichlet) boundary conditions
- *bcinds*: indices of the boundary nodes
- *bcvals*: values of the boundary nodes
- *invinds*: indices of the inner nodes
- *fv*: right hand side of the momentum equation
- *fp*: right hand side of the continuity equation
- *charlen*: characteristic length of the setup
- *nu*: the kinematic viscosity
- *Re*: the Reynolds number
- *odcoo*: dictionary with the coordinates of the domain of observation

- ***cdcoo*: dictionary with the coordinates of the domain of * *ppin*** [{int, None}]

  which dof of *p* is used to pin the pressure, typically *-1* for internal flows, and *None* for flows with outfl
  control

stokesmatsc : dict

a dictionary of the condensed matrices:

- *M*: the mass matrix of the velocity space,
- *A*: the stiffness matrix,
- *JT*: the gradient matrix, and
- *J*: the divergence matrix
- *Jfull*: the uncondensed divergence matrix

and, if *bccontrol=True*, the boundary control matrices that weakly impose *Arob*v = Brob*u*, where

- *Arob*: contribution to *A*
- *Brob*: input operator

'if mergerhs' :

rhsd : dict

rhsd_vfrc and *rhsd_stbc* merged

'else' :

rhsd_vfrc : dict

of the dirichlet and pressure fix reduced right hand sides

rhsd_stbc : dict

of the contributions of the boundary data to the rhs:

- *fv*: contribution to momentum equation,
- *fp*: contribution to continuity equation

Examples

femp, stokesmatsc, rhsd_vfrc, rhsd_stbc = get_sysmats(problem='drivencavity', N=10, nu=1e-2)

`problem_setups.`**`drivcav_fems`**(*N*, *vdgree=2*, *pdgree=1*, *scheme=None*, *bccontrol=None*)

dictionary for the fem items of the (unit) driven cavity

Parameters **N** : int

mesh parameter for the unitsquare (N gives 2*N*N triangles)

vdgree : int, optional

polynomial degree of the velocity basis functions, defaults to 2

pdgree : int, optional

polynomial degree of the pressure basis functions, defaults to 1

scheme : {None, 'CR', 'TH'}

the finite element scheme to be applied, 'CR' for Crouzieux-Raviart, 'TH' for Taylor-Hood, overrides *pdgree*, *vdgree*, defaults to *None*

bccontrol : boolean, optional

whether to consider boundary control via penalized Robin defaults to false. TODO: not implemented yet but we need it here for consistency

Returns femp : a dict

of problem FEM description with the keys:

- *V*: FEM space of the velocity
- *Q*: FEM space of the pressure
- *diribcs*: list of the (Dirichlet) boundary conditions
- *fv*: right hand side of the momentum equation
- *fp*: right hand side of the continuity equation
- *charlen*: characteristic length of the setup
- *odcoo*: dictionary with the coordinates of the domain of observation
- *cdcoo*: dictionary with the coordinates of the domain of control

`problem_setups.`**`cyl_fems`**(*refinement_level=2*, *vdgree=2*, *pdgree=1*, *scheme=None*, *bccontrol=False*, *verbose=False*)

dictionary for the fem items for the cylinder wake

Parameters N : mesh parameter for the unitsquare (N gives 2*N*N triangles)

vdgree : polynomial degree of the velocity basis functions,

defaults to 2

pdgree : polynomial degree of the pressure basis functions,

defaults to 1

scheme : {None, 'CR', 'TH'}

the finite element scheme to be applied, 'CR' for Crouzieux-Raviart, 'TH' for Taylor-Hood, overrides *pdgree*, *vdgree*, defaults to *None*

bccontrol : boolean, optional

whether to consider boundary control via penalized Robin defaults to *False*

Returns femp : a dictionary with the keys:

- *V*: FEM space of the velocity
- *Q*: FEM space of the pressure
- *diribcs*: list of the (Dirichlet) boundary conditions
- *dirip*: list of the (Dirichlet) boundary conditions for the pressure
- *fv*: right hand side of the momentum equation
- *fp*: right hand side of the continuity equation
- *charlen*: characteristic length of the setup
- *odcoo*: dictionary with the coordinates of the domain of observation
- *cdcoo*: dictionary with the coordinates of the domain of control

- *uspacedep*: int that specifies in what spatial direction Bu changes. The remaining is constant
- *bcsubdoms*: list of subdomains that define the segments where the boundary control is applied

Notes

parts of the code were taken from the NSbench collection https://launchpad.net/nsbench

__author__ = "Kristian Valen-Sendstad <kvs@simula.no>"
__date__ = "2009-10-01"
__copyright__ = "Copyright (C) 2009-2010 " + __author__
__license__ = "GNU GPL version 3 or any later version"

data_output_utils

`data_output_utils.`**`output_paraview`**(*V=None*, *Q=None*, *fstring='nn'*, *invinds=None*, *diribcs=None*, *vp=None*, *vc=None*, *pc=None*, *ppin=-1*, *t=None*, *writeoutput=True*, *vfile=None*, *pfile=None*)

write the paraview output for a solution vector vp

`data_output_utils.`**`load_or_comp`**(*filestr=None*, *comprtn=None*, *comprtnargs={}*, *arraytype=None*, *debug=False*, *loadrtn=None*, *loadmsg='loaded '*, *savertn=None*, *savemsg='saved '*, *itsadict=False*, *numthings=1*)

routine for caching computation results on disc

Parameters **filestr: {string, list of strings, 'None'}** :

where to load/store the computed things, if *None* nothing is loaded or stored

arraytype: {'None', 'sparse', 'dense'} :

if not None, then it sets the default routines to save/load dense or sparse arrays

itsadict: boolean, optional :

whether it is *python dictionary* that can be JSON serialized, overrides all other options concerning arrays

savertn: fun(), optional :

routine for saving the computed results, defaults to None, i.e. no saving here

debug: boolean, optional :

no saving or loading, defaults to *False*

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

d

p

s

Index

A

C

D

E

G

L

M

O

P

S